A Theory of Secure Control Flow

Martín Abadi
Computer Science Department
University of California, Santa Cruz

Mihai Budiu
Microsoft Research
Silicon Valley

Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Computer Science Department
Princeton University

February, 2005
Revised June, 2005

MSR-TR-05-17

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A Theory of Secure Control Flow

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

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

Abstract. Control-Flow Integrity (CFI) means that the execution of a program dynamically follows only certain paths, in accordance with a static policy. CFI can prevent attacks that, by exploiting buffer overflows and other vulnerabilities, attempt to control program behavior. This paper develops the basic theory that underlies two practical techniques for CFI enforcement, with precise formulations of hypotheses and guarantees.

1 Introduction

Many modern attacks against computers take advantage of software flaws, such as buffer-overflow or integer-overflow vulnerabilities. The abundance of software flaws, and the corresponding success of the attacks, has motivated substantial defensive efforts. These efforts include systematic attempts to eliminate those flaws from legacy software and to avoid them in new software, relying on programmer education and security reviews. Although these attempts have been at least partly fruitful, one might be concerned about their cost, and also about the possibility that they will not remove all flaws. Therefore, complementary approaches have also been considered and sometimes adopted.

One such approach is the use of various mitigation tools. These tools can be applied to code, more or less automatically, in order to reduce or eliminate the effects of certain vulnerabilities. The goals of these tools include runtime detection of buffer overflows [4, 12], randomization and artificial heterogeneity [10, 19], and tainting of suspect data [16]. Unfortunately, these tools often target only specific classes of vulnerabilities. For example, stack canaries [4] address only certain buffer overflows in the stack (and none in the heap). Moreover, these tools offer imperfect, hard-to-define safeguards, which determined attackers can defeat or circumvent [11, 13, 18].

Another approach is the adoption of high-level, type-safe languages, such as Java and C#. These languages aim to guarantee general, fundamental properties that can be defined precisely and proved rigorously, in particular memory safety. These properties contribute greatly to program security. Unfortunately, implementation flaws and interoperation with low-level code can weaken the guarantees. Moreover, it is questionable whether every piece of software will be written or rewritten in these languages.

A third approach, which we advocate, is the enforcement of *Control-Flow Integrity* (CFI). CFI means that program execution dynamically follows only certain paths, in accordance with a statically specified policy given as a control-flow graph (CFG). Many attacks aim to subvert execution and control 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 [11]. Since these attacks invariably affect control flow, CFI can prevent them. Like various mitigation tools, CFI enforcement can be applied to existing source code and binaries. At the same time, CFI has much in common with the properties guaranteed by high-level, type-safe languages. In particular, as we demonstrate, CFI can be defined precisely and proved rigorously.

In a companion paper [2], we motivate CFI and present an implementation. The implementation relies on machine-code rewriting that instruments software with runtime checks; it applies to legacy systems (e.g., code compiled from C and C++ on x86 Windows) with a modest performance overhead. We also validate, experimentally, that CFI thwarts many types of exploits and several documented past attacks. Finally, we show that CFI can help in the enforcement of additional security properties.

This paper is concerned with the foundations of CFI. It develops the basic theory that underlies our strategy for CFI enforcement. It includes a detailed semantics for programs, definitions for program instrumentation, and theorems about the executions of instrumented programs. We regard this basic theory as central to our approach. The precise formulation of hypotheses, guarantees, and proofs is a major difference between our approach and those based on previous mitigation tools, and an important similarity with research on high-level, type-safe languages. Furthermore, a formal approach is useful not only for elucidating hypotheses and guarantees, but also as a guide in the design and development of techniques. Indeed, in the course of our work, we rejected several alternatives that made unclear assumptions or that offered protection only in hard-to-define circumstances.

The main theorems of the paper establish that CFI holds for programs processed according to either of two enforcement techniques, even with respect to a powerful attacker that controls data memory. Although both techniques employ machine-code rewriting, they differ in their specifics and their assumptions. Most noticeably, one technique requires that data memory not be executable. This assumption, which we call NXD, thwarts some attacks on its own, but not those that exploit unintended control transfers in pre-existing code, such as "jump-to-`libc`" attacks [11]. Some architectures support NXD, and recent versions of Windows use it [8]. NXD can also be implemented in software, with support from the underlying operating system [10]. The second technique is a refinement of the first with a built-in, inline implementation of NXD.

The second technique relies on a generalization of *Software Fault Isolation* (SFI) [17] that we call *Software Memory Access Control* (SMAC). SFI provides multiple domains of memory protection within a single address space. For SFI, code inserted before each memory access ensures that the target memory address is within a certain range. For SMAC, more generally, each instruction that may perform a memory access is constrained to a particular range of addresses, potentially a different one per instruction. CFI can facilitate the implementation of SMAC for irregular architectures, such as the x86, on which traditional SFI has been problematic [5]. One of the goals of this paper is to show that this cooperation between CFI and SMAC is real, rather than an incorrect result of informal circular reasoning.

Section 2 defines the setting for our work: a simple machine model and a corresponding machine language. Section 3 discusses CFGs. Section 4 describes and analyzes the first technique for CFI enforcement. Section 5 concerns the second technique, with SMAC. Section 6 concludes. Some details of proofs and additional material can be found at our website [3].

2 The Setting: Programs and their Semantics

The machine model and the programs that we define in this section are typical of formal studies in programming-language theory. For the sake of simplicity, we work with a basic machine model and a small set of machine instructions which enable us to study CFI but excludes virtual memory, dynamic linking, threading, and other sophisticated features found in actual systems. Essentially, our language is a minor variant of that of Hamid et al. [6]. We have yet to attempt a similar investigation for the full x86 architecture, and for our x86 code sequences. We believe that such an investigation would be feasible, particularly because of the similarities between our x86 code sequences and those studied in this paper; on the other hand, the investigation would certainly be laborious and may yield diminishing returns.

2.1 Machine Model

For our machine model, we define words, memories, register files, and states as follows:

$$\begin{aligned} \mathit{Word} &= \{0, 1, ...\} \\ \mathit{Mem} &= \mathit{Word} \rightarrow \mathit{Word} \\ \mathit{Regnum} &= \{0, 1, ..., 31\} \\ \mathit{Regfile} &= \mathit{Regnum} \rightarrow \mathit{Word} \\ \mathit{State} &= \mathit{Mem} \times \mathit{Regfile} \times \mathit{Word} \end{aligned}$$

We often adopt the notations w and pc for elements of Word, and M, R, and S for elements of Mem, $\mathit{Regfile}$, and State, respectively.

When S is a state, we may write $S.M$, $S.R$, and $S.pc$ for the Mem component, the $\mathit{Regfile}$ component, and the pc in S, respectively. We further distinguish between code memory (M_c) and data memory (M_d), so we split memories into two functions with disjoint domains, each of them contiguous. We assume that a statically defined program that comprises $n > 0$ instructions always occupies memory locations 0 to $n - 1$, with the first instruction of the program located at address 0.

When we split a 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 hold: $\mathrm{dom}(M_c) = \{0..(n-1)\}$, and $\mathrm{dom}(M_d) = \mathrm{dom}(M) - \mathrm{dom}(M_c)$, and $M_c(a) = M(a)$ for all $a \in \mathrm{dom}(M_c)$, and $M_d(a) = M(a)$ for all $a \in \mathrm{dom}(M_d)$. 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 register files into distinguished and general registers. When we split R into R_{0-2} and R_{3-31}, we write $R = R_{0-2} | R_{3-31}$ provided the following constraints hold: $\mathrm{dom}(R_{0-2}) = \{r_0, r_1, r_2\}$, and $\mathrm{dom}(R_{3-31}) = \{r_3..r_{31}\}$, and $R_{0-2}(r) = R(r)$ for all $r \in \mathrm{dom}(R_{0-2})$, and $R_{3-31}(r) = R(r)$ for all $r \in \mathrm{dom}(R_{3-31})$. We distinguish the registers r_0, r_1, and r_2 because we assume that they are used only in CFI enforcement code. (In fact, in our x86 implementation, we need only one distinguished register and only at certain program points. This feature is important in practice, since the x86 architecture has few registers. While permanently reserving many registers for a special use is difficult, finding a free register now and then is easy.)

2.2 Instructions

Our language is that of Hamid et al. [6] plus a *label* instruction in which an immediate value can be embedded and which behaves like a nop. (It is not too hard to implement such a *label* instruction on common architectures.) The set of instructions is:

Instr ::=	instructions
label w	label (with embedded constant)
add r_d, r_s, r_t	add registers
addi r_d, r_s, w	add register and word
movi r_d, w	move word into register
bgt r_s, r_t, w	branch-greater-than
jd w	jump
jmp r_s	computed jump
ld $r_d, r_s(w)$	load
st $r_d(w), r_s$	store
illegal	illegal

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 $Dc : \mathit{Word} \rightarrow \mathit{Instr}$ that decodes words into instructions.

2.3 A Semantics of Programs under Attack

In this section we give a first semantics for instructions. Figure 1 defines two binary relations on states, $\rightarrow_n$ and $\rightarrow_a$.

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

$$\frac{}{(M_c|M_d, R_{0-2}|R_{3-31}, pc) \to_a (M_c|M_d{}', R_{0-2}|R_{3-31}{}', pc)}$$

Fig. 1. Normal steps and attacker steps.

- The relation $\to_n$ models normal small steps of execution, that is, those steps that may occur in the absence of an attacker. This relation is deliberately incomplete: many states are "stuck", including those where $Dc(M_c(pc)) =$ *illegal*.
- The relation $\to_a$ models attack steps. In such a step, an attacker may unconditionally and arbitrarily perturb data memory and non-distinguished 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 $\to$, defined below, is the union of $\to_n$ and $\to_a$. Thus, this relation represents a computation step in general, either a normal state transition or one caused by an attacker.

$$\frac{S \to_n S'}{S \to S'} \qquad \frac{S \to_a S'}{S \to S'}$$

In security, it is important to identify assumptions, and to justify them to the extent possible, because an attacker that can invalidate assumptions can often circumvent security enforcement. Our definitions embody several assumptions, which we discuss next:

1. The definition of $\rightarrow_n$ implies NXD (that is, that data cannot be executed as code). Similarly, the definitions of $\rightarrow_n$ and $\rightarrow_a$ imply that code memory cannot be modified at runtime. We call this property NWC. As indicated in the introduction, NXD is often a reasonable assumption. NWC holds on most current systems (except at special times, such as during the initial loading of dynamic libraries).
2. The definition of $\rightarrow_a$ allows for the possibility that the attacker is in control of data memory. This aspect of the model of the attacker is conservative, but unfortunately close to reality. Buffer overflows and other vulnerabilities often allow an attacker to write to arbitrary locations in data memory even before subverting control flow [11].
3. The definition of $\rightarrow_a$ implies that the attacker cannot modify the distinguished registers r_0, r_1, and r_2. In practice, one may ensure this property by avoiding the use of r_0, r_1, and r_2 outside the CFI enforcement code. Our proofs require only a weaker assumption, namely that the attacker cannot modify r_0, r_1, and r_2 during the execution of CFI enforcement code.
4. The machine model and the definition of $\rightarrow_n$ exclude the possibility that a jump would land in the middle of an instruction. In practice, many architectures (RISC architectures, in particular) exclude this possibility, and our x86 CFI implementation prevents it. For simplicity, we do not address this feature in the formal analysis.

2.4 A More Permissive Semantics of Programs under Attack

Assumptions NXD and NWC do not hold in some settings, for example on architectures without memory-protection facilities. We should therefore consider an alternative to the program semantics of Section 2.3. For brevity, and since there is no risk of ambiguity below, we reuse the symbols $\rightarrow_n$, $\rightarrow_a$, and $\rightarrow$.

The resulting, relaxed definition of normal execution steps is in Figure 2. These normal steps can arbitrarily violate NXD and NWC, possibly under the indirect influence of an attacker. On the other hand, the rules for attack steps and general steps remain those of Section 2.3. In particular, we still require that an attack step cannot directly alter code memory, the distinguished registers, or the program counter. We believe that these restrictions often hold in practice. Moreover, they are necessary: without them, an attacker could trivially create new code (outside the original CFG) and trigger its execution.

3 The CFG

Our instrumentation of a program relies on a CFG for the program, as specification of a CFI policy. Next we discuss this CFG.

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 $w' \in$

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

Fig. 2. Normal steps (assuming less memory protection).

$\text{dom}(M_c)$ such that G has an edge from w to w'. We say that w' is a destination if there exists w such that $Dc(M_c(w))$ is a computed jump instruction ($jmp\ r_s$) and $w' \in \text{succ}(w)$.

We need not constrain how the CFG is obtained, or how it matches the executions of the program before instrumentation. The CFG might be computed by analyses, static or dynamic. It might also be derived, at least in part, from a security policy, for example one expressed as a security automaton [5, 7]. (For our implementation, we derive the CFG by static analysis of binaries.) We do require:

1. If $Dc(M_c(w_0)) = label\ w$, or $add\ r_d, r_s, r_t$, or $addi\ r_d, r_s, w$, or $movi\ r_d, w$, or $ld\ r_d, r_s(w)$, or $st\ r_d(w), r_s$, then $\text{succ}(w_0) = \{w_0 + 1\} \cap \text{dom}(M_c)$.
2. If $Dc(M_c(w_0)) = bgt\ r_s, r_t, w$ then $\text{succ}(w_0) = \{w_0 + 1, w\} \cap \text{dom}(M_c)$.
3. If $Dc(M_c(w_0)) = jd\ w$ then $\text{succ}(w_0) = \{w\} \cap \text{dom}(M_c)$.
4. If $Dc(M_c(w_0)) = jmp\ r_s$ then $\text{succ}(w_0) \neq \emptyset$.
5. $Dc(M_c(w_0)) = illegal$ then $\text{succ}(w_0) = \emptyset$.
6. If $w_0, w_1 \in \text{dom}(M_c)$, then $\text{succ}(w_0) \cap \text{succ}(w_1) = \emptyset$ or $\text{succ}(w_0) = \text{succ}(w_1)$.

When these properties hold, we say that the graph in question is a CFG for M_c.

These properties hold by definition for many graphs that arise from code analysis. Only the last one (6) is non-trivial. Property 6 can be satisfied by adding edges to a CFG; the additional edges result in a looser CFI policy. We believe that this approach is satisfactory in practice: when most addresses are not destinations, even a coarse CFG that allows control to flow from any jump instruction to any destination can thwart many attacks. Alternatively, property 6 can be satisfied by duplicating nodes where the condition is violated. In the extreme, unrealistic case where the condition is violated at all nodes, we may rely on the following construction: given a graph G, we define a new graph G' such that the nodes of G' are pairs of nodes of G, and there is an edge from (a_1, a_2) to (b_1, b_2) in G' when $b_1 = a_2$ and there is an edge from a_2 to b_2 in G. (We omit the straightforward proof that G' satisfies property 6.)

Because of property 6, we can put destinations into equivalence classes. We give each equivalence class an identifier, called an ID. We represent these IDs by words. For a *jmp* instruction at address w in M_c, we let $\mathrm{dst}(w)$ be the ID of all successors of w. Thus, $\mathrm{dst}(w)$ is the ID of any element of $\mathrm{succ}(w)$.

We write $\mathrm{succ}(M_c, G, w)$ and $\mathrm{dst}(M_c, G, w)$, instead $\mathrm{succ}(w)$ and $\mathrm{dst}(w)$ respectively, when we wish to be explicit on M_c and G.

4 CFI Enforcement (without SMAC)

In this section we present and analyze our first technique for CFI enforcement.

4.1 CFI Enforcement by Instrumentation

CFI means that, during program execution, whenever a machine-code instruction transfers control, it targets a valid destination according to a given CFG. For instructions that target a constant destination, this requirement can be discharged statically. On the other hand, for computed control-flow transfers (whose destination is determined at runtime), this requirement must be discharged with a dynamic check.

Machine-code rewriting offers an attractive, realistic strategy for implementing dynamic checks. Modern tools for binary instrumentation address the substantial technical difficulties of machine-code rewriting [14, 15].

Unfortunately, machine-code rewriting remains complex and tied to many implicit compiler-specific details. Therefore, for the sake of trustworthiness, CFI enforcement should preferably depend only on simple, final, static verification steps that check that the instrumentation has produced an acceptable result. These steps, but not the machine-code rewriting, will be part of the “trusted computing base”.

For the present purposes, the verification steps consist in ensuring that a code memory M_c and a CFG G for M_c satisfy the following properties:

1. If n is the length of $\mathrm{dom}(M_c)$, then the instruction at $n - 1$ is *illegal*. (In other words, the final instruction is *illegal*.)
2. If $w_0 \in \mathrm{dom}(M_c)$ is a destination, then the instruction at w_0 is *label* w, where w is w_0's ID. Conversely, if $w_0 \in \mathrm{dom}(M_c)$ holds a *label* instruction, then w_0 is a destination. (In other words, *label* instructions can be used only for inline tagging with IDs. This requirement applies to code memory, but not to data memory. In fact, the attacker may, at any time, write *label* w into any location in data memory.)

3. If $w_0 \in \mathrm{dom}(M_c)$ holds a *jmp* instruction, then this instruction is $\mathit{jmp}\ r_0$ and it is preceded by a specific sequence of instructions, as follows:

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

   where r_s is some register, HALT is the address of the *illegal* instruction specified in property (1), and *IMM* is the word w such that $\mathit{Dc}(w) = \mathit{label}\ \mathrm{dst}(w_0)$. This code compares the dynamic target of a jump, which is initially in register r_s, to the *label* instruction that is expected to be the target statically. When the comparison succeeds, the jump proceeds. When it fails, the program halts.
4. If $\mathit{bgt}\ r_s, r_t, w$ or $\mathit{jd}\ w$ appear anywhere in M_c, then the target address w does not hold a *jmp* instruction or any of the preceding instructions

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

   described in (3). The target address may hold $\mathit{addi}\ r_0, r_s, 0$. (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 $I(M_c, G)$ means that M_c and its CFG G satisfy the conjunction of the properties above.

4.2 A Theorem about CFI

With these definitions, and under the semantics of Section 2.3, we can obtain formal results about our instrumentation method.

Here we present 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 $pc = 0$ and $I(M_c, G)$, where G is a CFG for M_c, and let $S_1, \ldots, S_n$ be states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \rightarrow_a S_{i+1}$ and $S_{i+1}.pc = S_i.pc$, or $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, G, S_i.pc)$.*

The proof of this theorem consists in a fairly classical induction on executions, with an invariant. In particular, the proof constrains the values of the distinguished registers within the instrumentation sequences, but puts no restrictions on the use of these registers elsewhere in the program.

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 a certain `libc` routine should not be reachable, then executing the code memory will never result in running that routine. Thus, "jump-to-`libc`" attacks that target dangerous routines (such as `system` in Unix and `ShellExecute` in Windows) can be effectively thwarted.

As explained in the introduction, our first technique for CFI enforcement depends on NXD. More specifically, the theorem depends on the formal version of NXD, which says that, during execution, the targets of code transfers are always in the domain of code memory. Without this property, the theorem would fail, since data memory may well contain *label* w instructions that look like the expected destinations of *jmp* instructions.

5 CFI Enforcement (with SMAC)

Our second technique for CFI enforcement builds on the first, eliminates the need for NXD, and allows program execution steps to modify code memory. While it may be viewed as a refinement of the first (perhaps via a simulation relation), in this section we present it and study it on its own, as a complete and separate mechanism.

SMAC has a number of applications beyond the one described here. For instance, it can serve to protect a call stack in memory, and thereby serve to strengthen CFI by matching calls and returns dynamically [2]. For brevity, we do not formalize those applications in this paper.

5.1 CFI Enforcement by 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.

The SMAC-based verification steps consist in ensuring that a code memory M_c and a CFG G for M_c satisfy the following properties:

1. If n is the length of $\mathrm{dom}(M_c)$, then the instruction at $n-1$ is *illegal*.
2. If $w_0 \in \mathrm{dom}(M_c)$ is a destination, then the instruction at w_0 is *label* w, where w is w_0's ID. Conversely, if $w_0 \in \mathrm{dom}(M_c)$ holds a *label* instruction, then w_0 is a destination.

3. If $w_0 \in \mathrm{dom}(M_c)$ holds at a *st* instruction, then this instruction is $\mathit{st}\ r_0(0), r_s$ and it is preceded by a specific sequence of instructions, as follows:

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

   where r_d is some register, w is some offset (a word), and HALT is the address of the *illegal* instruction specified in property (1). This code constrains a store to memory, with address initially given by $R(r_d) + w$, to be between $\min(M_d)$ and $\max(M_d)$. This constraint is imposed by two dynamic comparisons. When these two comparisons succeed, the store proceeds; otherwise, the program halts.
4. If $w_0 \in \mathrm{dom}(M_c)$ holds a *jmp* instruction, then this instruction is $\mathit{jmp}\ r_0$ and it is preceded by a specific sequence of instructions, as follows:

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

   where r_s is some register, HALT is the address of the *illegal* instruction specified in property (1), and IMM is the word w such that $Dc(w) = \mathit{label}\ \mathrm{dst}(w_0)$. This code is a combination of the code for *jmp* described in Section 4 with an analogue of the code for *st* described above. As in the code for *st*, an address is constrained to be within a range; here the range is the domain of code memory, and the address is the dynamic target of a jump, held in r_s. Then, as in the code for *jmp* in Section 4, that dynamic target is compared with the *label* instruction expected statically. The program halts unless all checks succeed.
5. If $\mathit{bgt}\ r_s, r_t, w$ or $\mathit{jd}\ w$ appear anywhere in M_c, then the target address w is in code memory (that is, $w \in \mathrm{dom}(M_c)$), and w does not hold *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 $\mathit{addi}\ r_0, r_d, w$ and $\mathit{addi}\ r_0, r_s, 0$, respectively.

The predicate $I_s(M_c, G)$ means that M_c and its CFG G satisfy the conjunction of the properties above.

5.2 A Theorem about CFI with SMAC

With the relaxed semantics of Section 2.4 and the instrumentation of Section 5, we obtain a direct analogue to Theorem 1.

Theorem 2. *Let S_0 be a state $(M_c|M_d, R, pc)$ such that $pc = 0$ and $I_s(M_c, G)$, where G is a CFG for M_c, and let $S_1, \ldots, S_n$ be states such that $S_0 \to S_1 \to \ldots \to S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \to_a S_{i+1}$ and $S_{i+1}.pc = S_i.pc$, or $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, G, S_i.pc)$.*

The proof of this theorem is analogous to that of Theorem 1.

Because SMAC is implemented by inline checks, it could be circumvented by computed control-flow transfers into or around the code sequences that perform the checks. Therefore, SMAC is intimately tied to CFI, which prevents such subversive flows of control. Accordingly, our theorem is not about SMAC in isolation, but rather about the combination of SMAC and CFI.

6 Conclusion

In this paper we study techniques for the enforcement of Control-Flow Integrity (CFI). In a simple low-level language of the kind common in programming-language theory, we give definitions for program instrumentation and theorems about the executions of the instrumented programs. The rigorous clarification of assumptions and guarantees is helpful in the development and validation of software-security techniques, and more broadly beneficial for security. While our theorems do not directly say that nothing bad will ever happen—and indeed CFI does not prevent all security problems—they do imply fundamental properties that exclude a variety of attacks.

Many attacks make use of the fact that, at the lowest levels of systems, almost any behavior is considered valid—independently of whether the executing software is written in a structured fashion, e.g., as high-level functions in C or C++. For instance, even activity that is patently invalid for programs that originate in high-level, structured languages (such as jumping into the middle of a function body) is permitted at the hardware level. Similarly, even programs that use very limited system functionality (such as those that only draw on the screen but never use the file system or network) are typically allowed to invoke any operating system service or runtime library routine.

CFI can align low-level behavior with high-level intent, as specified in a CFG. In this respect, CFI is reminiscent of the use of typed low-level languages, such as TAL [9], and of efforts to bridge the gaps between high-level languages and actual behavior (e.g., [1]). Furthermore, the basic theory of CFI enforcement that we develop in this paper relies heavily on fundamental ideas and techniques of the modern literature on programming languages. We regard the viability of this theory as an important feature of CFI. More broadly, we believe that theories based on programming-language methods can enhance assurance and provide guidance for a wide range of approaches to software security.

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

References

1. Martín Abadi. Protection in programming-language translations. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer-Verlag, 1998. Also Digital Equipment Corporation Systems Research Center report No. 154, April 1998.
2. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. Technical Report MSR-TR-05-18, Microsoft Research, 2005.
3. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Further formal material on CFI and SMAC. Manuscript, available at `http://research.microsoft.com/research/sv/gleipnir`, 2005.
4. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Symposium*, pages 63–78, 1998.
5. Ú. Erlingsson and F.B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*. ACM Press, 2000.
6. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof-Carrying Code. Technical Report YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University, New Haven, CT, 2002.
7. Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.
8. Microsoft Corporation. Changes to functionality in Microsoft Windows XP SP2: Memory protection technologies, 2004. `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`.
9. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
10. PaX Project. The PaX project, 2004. `http://pax.grsecurity.net/`.
11. J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.
12. O. Ruwase and M.S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of Network and Distributed System Security Symposium*, 2004.
13. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2004.
14. A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

15. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report WRL Research Report 94/2, Digital Equipment Corporation, 1994.
16. G.E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM Press, 2004.
17. R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1993.
18. J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*, 2003.
19. J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable and Distributed Systems*, 2003.