Control-Flow Integrity
Principles, Implementations, and Applications

MARTÍN ABADI
University of California, Santa Cruz
and
Microsoft Research, Silicon Valley

MIHAI BUDIU
Microsoft Research, Silicon Valley

ÚLFAR ERLINGSSON
Microsoft Research, Silicon Valley

and
JAY LIGATTI
University of South Florida

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

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

General Terms: Security, Languages, Verification

Additional Key Words and Phrases: Binary Rewriting, Control-Flow Graph, Inlined Reference Monitors, Vulnerabilities

Parts of this work were presented, in preliminary form, at the Twelfth ACM Conference on Computer and Communications Security [Abadi et al. 2005] and at the Seventh International Conference on Formal Engineering Methods [Abadi et al. 2005].

Authors' addresses: M. Abadi, Computer Science Department, University of California, 1156 High Street, Santa Cruz, CA 95064, and Microsoft Corporation, 1065 La Avenida, Mountain View, CA 94043; email: abadi@cs.ucsc.edu. M. Budiu and Ú. Erlingsson, Microsoft Corporation, 1065 La Avenida, Mountain View, CA 94043; email: {mbudiu,ulfar}@microsoft.com. J. Ligatti, Department of Computer Science and Engineering, University of South Florida, 4202 E Fowler Ave., ENB 118, Tampa, FL 33620; email: ligatti@cse.usf.edu.

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, trigger pre-existing software flaws. By exploiting such flaws, the attacks can subvert execution and gain control over software behavior. For instance, a buffer overflow in an application may result in a call to a sensitive system function, possibly a function that the application was never intended to use. The combined effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many ingenious vulnerability mitigations have been proposed for defending against these attacks; these include stack canaries [Cowan et al. 1998], runtime elimination of buffer overflows [Ruwase and Lam 2004], randomization and artificial heterogeneity [PaX Project 2004; Xu et al. 2003], and tainting of suspect data [Suh et al. 2004]. 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 [Pincus and Baker 2004; Shacham et al. 2004; Wilander and Kamkar 2003].

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

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

The CFI security policy dictates that software execution must follow a path of a *Control-Flow Graph* (CFG) determined ahead of time. The CFG in question can be defined by analysis—source-code analysis, binary analysis, or execution profiling. For our experiments, we focus on CFGs derived by a static binary analysis. CFGs can also be defined by explicit security policies, for example written as security automata [Schneider 2000].

A security policy is of limited value without an attack model. In our design, CFI enforcement provides protection even against powerful adversaries that have full control over the entire data memory of the executing program. This model of adversaries may seem rather pessimistic. On the other hand, it has a number of virtues. First, it is clear and amenable to formal definition and analysis. It also allows for the real possibility that buffer overflows or other vulnerabilities (e.g., [Govindavajhala and Appel 2003]) would lead to arbitrary changes in data memory, even before control flow is subverted. Finally, it applies even when an attacker is in active control of a module or thread within the same address space as the program being protected.

Whereas CFI enforcement can potentially be done in several ways, we rely on a combination of lightweight static verification and machine-code rewriting that instruments software with runtime checks. The runtime checks dynamically ensure that control flow re-

mains within a given CFG. Previous work has relied on similar instrumentation for other security policies (e.g., [Erlingsson and Schneider 1999; Evans and Twyman 1999; Bauer et al. 2005]). As we demonstrate, machine-code rewriting results in a practical implementation of CFI enforcement. This implementation applies to existing user-level programs on commodity systems, and yields efficient code even on irregular architectures with variable-length instruction encodings. Furthermore, although machine-code rewriting can be rather elaborate, it is simple to verify the proper use of instrumentation in order to ensure inlined CFI enforcement.

CFI enforcement is effective against a wide range of common attacks, since abnormal control-flow modification is an essential step in many exploits—independently of whether buffer overflows and other vulnerabilities are being exploited [Pincus and Baker 2004; Wilander and Kamkar 2003]. We have examined many concrete attacks and found that CFI enforcement prevents most of them. These include both classic, stack-based buffer-overflow attacks and newer, heap-based "jump-to-`libc`" attacks. They also include recently described "pointer subterfuge" attacks, which foil many previous mitigation techniques. Of course, CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG (e.g., [Chen et al. 2005]) are not prevented. These include, for example, certain exploits that rely on incorrect argument-string parsing to cause the improper launch of a dangerous executable.

No matter how the CFG is defined or how permissive it is, CFI can be used as a foundation for the enforcement of more sophisticated security policies, including those that prevent higher-level attacks. For example, CFI can prevent the circumvention of two well-known enforcement mechanisms, *Inlined Reference Monitors* (IRMs) and *Software Fault Isolation* (SFI) [Erlingsson and Schneider 2000; 1999; Wahbe et al. 1993]. In particular, CFI can help protect security-relevant information such as a shadow call stack [Frantzen and Shuey 2001; Nebenzahl and Wool 2004; Prasad and Chiueh 2003], which can be used for placing tighter restrictions on control flow. Further, CFI can serve as the basis of a generalized, efficient variant of SFI that we call *Software Memory Access Control* (SMAC), which is embodied in an inlined reference monitor for access to memory regions. SMAC, in turn, can serve for eliminating some CFI assumptions.

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

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

The next section, Section 2, discusses related work. Section 3 informally explains CFI and its inlined enforcement. Section 4 describes our main CFI implementation and gives performance results. It also reports on our security-related experiments. Section 5

shows how additional security enforcement can be built on CFI; it includes a discussion of IRMs and three important examples: faster SFI, SMAC, and a protected shadow call stack. Section 6 presents our formal work. (The Appendix contains the corresponding proofs.) Finally, Section 7 concludes.

2. RELATED WORK

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

2.1 CFI and Security

Constraining control flow for security purposes is not new. For example, computer hardware has long been able to prevent execution of data memory, and the latest x86 processors support this feature. At the software level, several existing mitigation techniques constrain control flow in some way, for example by checking stack integrity and validating function returns [Cowan et al. 1998; Prasad and Chiueh 2003], by encrypting function pointers [Cowan et al. 2003; Xu et al. 2003], or even by interpreting software using the techniques of dynamic machine-code translation [Kiriansky et al. 2002].

Clearly, this a crowded, important research area (e.g., [Avijit et al. 2004; Bhatkar et al. 2003; Brumley and Song 2004; Cowan et al. 2001; Crandall and Chong 2004; Frantzen and Shuey 2001; Kirovski and Drinic 2004; Larson and Austin 2003; Nebenzahl and Wool 2004; Necula et al. 2002; PaX Project 2004; Ruwase and Lam 2004; Suh et al. 2004; Tuck et al. 2004]). Next we elaborate on some of the pieces of work most closely related to ours. In short, we believe that the distinguishing features of CFI are its simplicity, its trustworthiness and amenability to formal analysis, its strong guarantees even in the presence of a powerful adversary with full control over data memory, and its deployability, efficiency, and scalability. Like many language-based security techniques, but unlike certain systems for intrusion detection, CFI enforcement applies even to the inner workings of user-level programs (not just at the system call boundary).

SFI and Inlined Reference Monitors. IRMs are a general technique for enforcing fine-grained security policies through inlined checks [Erlingsson and Schneider 2000; 1999]. SFI is an important, special IRM that performs dynamic checks for the purposes of memory protection [Erlingsson and Schneider 1999; McCamant and Morrisett 2005; Small 1997; Wahbe et al. 1993]. SFI and other IRMs operate by adding code for security checks into the programs whose behavior is the subject of security enforcement.

IRM implementations must consider that a subject program may attempt to circumvent the added checks—for example, by jumping around them. As a result, IRM implementations typically impose restrictions on control flow [McCamant and Morrisett 2005; Winwood and Chakravarty 2005]. The necessary restrictions are weaker than CFI.

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

write memory, may be executed through jumps into the middle of opcode byte sequences.

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

The performance of IRMs has been problematic, in large measure because of the need for control-flow checks, particularly on the x86 architecture [Erlingsson and Schneider 1999; McCamant and Morrisett 2005; Small 1997]. CFI offers an alternative, attractive implementation strategy; its guarantees, while stronger than strictly necessary for IRMs, imply the required control-flow properties, and thereby CFI can serve as a foundation for efficient IRM implementation. We elaborate on some of these points in Section 5.

Vulnerability Mitigation Techniques with Secrets. PointGuard [Cowan et al. 2003] stores code addresses in an encrypted form in data memory. The intent is that, even if attackers can change data memory, they cannot ensure that control flows to a code address of their choice: for this, they would have to know the corresponding decryption key. Several other techniques [Bhatkar et al. 2003; Cowan et al. 1998; PaX Project 2004; Tuck et al. 2004; Xu et al. 2003] also rely on secret values that influence the semantics of pointer addresses stored in memory. For instance, PaX ASLR shuffles the virtual-address-space layout of a process at the time of its creation, using a random permutation derived from a per-process secret [PaX Project 2004]. Some of these vulnerability mitigation schemes, such as the PaX Linux kernel patch, may be applied even to unmodified legacy binaries. Others can be more difficult to adopt, for example when they require complex source-code analysis.

Unfortunately, the reliance on secret values represents a vulnerability, because the values may not remain secret. In practice, a lucky, knowledgeable, or determined attacker can defeat these schemes (see [Shacham et al. 2004; Sovarel et al. 2005; Tuck et al. 2004]).

Secure Machine-Code Interpreters. Program Shepherding employs an efficient machine-code interpreter for implementing a security enforcement mechanism [Kiriansky et al. 2002], as does Strata [Scott and Davidson 2002]. The apparent complexity of these interpreters may affect their trustworthiness and complicate their adoption. Their performance overhead may be another obstacle to their use (see Section 4.2).

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

Other Research on Intrusion Detection. CFI is also related to a line of research on intrusion detection where a security policy for a program is derived from an inspection of the program itself or its executions [Basu and Uppuluri 2004; Feng et al. 2004; Feng et al. 2003; Forrest et al. 1996; Giffin et al. 2004; Gopalakrishna et al. 2005; Lam and Chiueh 2004; Sekar et al. 2001; Wagner and Dean 2001; Wagner and Soto 2002]. This security policy may be enforced at runtime using an isolated operating system mechanism, which cannot be circumvented or subverted, and which disallows invalid behaviors. Unlike in our work, the behaviors in question are often limited to sequences of system calls or library calls. This limitation has been regarded as a troublesome issue (e.g., in [Gopalakrishna et al. 2005, Section 8]).

In particular, Dean and Wagner describe an intrusion-detection technique that relies on a program's static CFG to achieve "a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms" at the system-call level [Wagner and Dean 2001]. Most recent work in this area aims to make the security policy more precise, reducing the number of false negatives, both by making use of runtime information about function calls and returns, and also by operating at the level of library functions as well as that of system calls.

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

CFI enforcement can be regarded as a fine-grained intrusion-detection mechanism based on a nondeterministic finite automaton. When CFI is coupled with a protected shadow call stack, the level of precision increases [Feng et al. 2004; Giffin et al. 2004]. Like previous work, CFI enforcement has difficulty with data-driven impossible paths and mimicry attacks [Wagner and Soto 2002]. CFI precision is also affected by the degree of fan-in/fan-out at choice points. (The literature contains several measurements of fan-in/fan-out in program code, which we do not repeat in this paper.) However, these difficulties are reduced because CFI restricts the behavior of every machine code instruction in subject programs. For instance, CFI ensures that, in the context of an application or library function, a system call can happen only after proper argument validation, thus eliminating mimicry attacks that provide malicious arguments to the system call (cf. [Gopalakrishna et al. 2005, Section 8]).

At the same time, CFI enforcement can be regarded as a basis for other intrusion-detection machinery. By using CFI and SMAC, one may be able to avoid modifications to the underlying operating system and the cost of operating-system interactions, while basically providing the same level of protection to the intrusion-detection machinery.

Language-Based Security. Virtually all high-level languages have execution models that imply some properties about the expected control flows. Even unsafe high-level languages are not meant to permit jumps into the middle of a machine-code instruction. Safer high-level languages, such as Java and C#, provide stronger guarantees. Their type systems, which aim to ensure memory safety, also constrain what call sequences are possible. Unfortunately, such guarantees hold only at the source level. Language implementations may not enforce them, and native method implementations may not respect them. Furthermore, it is questionable whether every piece of software will be written or rewritten in these languages. For instance, media codecs, automatic memory management, and operating-system interrupt dispatching typically rely on hand-written, optimized machine code; it seems unlikely that they will enjoy the full benefits of high-level languages, even in new systems.

Similar guarantees can be obtained at the assembly and binary levels through the use of proof-carrying code (PCC) [Necula 1997] or typed assembly language (TAL) [Morrisett et al. 1999]. Again, while PCC and TAL primarily aim to provide memory safety, they also impose static restrictions on control flow. Their properties have often been analyzed formally. The analyses focus on a model in which data memory may be modified by the subject program, but they typically do not give guarantees if another entity or a flaw may

corrupt data memory (e.g., [Govindavajhala and Appel 2003]).

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

2.2 CFI and Fault-Tolerance

Our work is also related to research on fault-tolerance of computer systems against soft faults (single-event upsets). Most relevant are methods that attempt to discern program execution deviation from a prescribed static CFG solely through software-based methods (e.g., [Oh et al. 2002; Reis et al. 2005; Venkatasubramanian et al. 2003]). Those methods exhibit many similarities with CFI enforcement, but also significant differences.

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

The method of Oh et al. [2002] is most similar to our CFI instrumentation in how it restricts control flow through inlined labels and checks. That method, like ours, encodes the CFG (or an approximation) by embedding a set of static, immediate bit patterns in the program code. However, in that method, the runtime checks are evaluated at the destinations of all branches and jumps, not at their sources. These checks are therefore ill-suited for our purposes. For instance, they fail to prevent jumps into the middle of functions, in particular jumps that may bypass security checks (such as access control checks). These details are consistent with the probabilistic failure model, but they would be unsatisfactory for security.

3. INLINED CFI ENFORCEMENT

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

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

3.1 Enforcing CFI by Instrumentation

CFI requires that, during program execution, whenever a machine-code instruction transfers control, it targets a valid destination, as determined by a CFG created ahead of time.

Since most instructions target a constant destination, this requirement can usually be discharged statically. However, for computed control-flow transfers (those whose destination is determined at runtime) this requirement must be discharged with a dynamic check.

Machine-code rewriting presents an apparently straightforward strategy for implementing dynamic checks. It is however not without technical wrinkles. In particular, a rewritten program no longer uses the same code memory, and all memory addresses in the program must be adjusted accordingly. Furthermore, changes like that of the memory layout may not be possible without potentially affecting the semantics of some unconventional programs. Modern tools for binary instrumentation address these and other wrinkles, often trading generality and simplicity for efficiency [Srivastava et al. 2001; Srivastava and Eustace 1994]. In particular, these tools exploit debug information in order to facilitate analysis and do not process stripped binaries (cf. [Harris and Miller 2005]). As a result, machine-code rewriting is practical and dependable.

It remains to design the dynamic checks. There are several possible strategies. For instance, CFI may be enforced by dynamic checks that compare the target address of each computed control-flow transfer to a set of allowed destination addresses. Such a comparison may be performed by the machine-code equivalent of a switch statement over a set of constant addresses. Since the set of allowed destination addresses may be large, this naive strategy will lead to unacceptable overhead.

Next we explain our preferred alternatives for 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 three new machine-code instructions, with an immediate operand `ID`: an effect-free `label ID` instruction; a call instruction `call ID, DST` that transfers control to the code at the address contained in register `DST` only if that code starts with `label ID`; and a corresponding return instruction `ret ID`. Such instructions could perhaps be added to common processors to form the basis for attractive hardware CFI implementations [Budiu et al. 2006]. However, it is unrealistic to expect the deployment of hardware CFI support in the near future. In the remainder of the paper, we discuss only software CFI implementations. As we demonstrate, inlined CFI enforcement can be implemented in software on current processors, in particular on the x86 processor, with only a modest overhead.

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

Figure 1 shows a C program fragment where the function `sort2` calls a qsort-like function `sort` twice, first with `lt` and then with `gt` as the pointer to the comparison function. The right side of Figure 1 shows an outline of the machine-code blocks for these four functions and all CFG edges between them. In the figure, edges for direct calls are drawn as light, dotted arrows; edges from source instructions are drawn as solid arrows, and return edges as dashed arrows. In this example, `sort` can return to two different places in

```
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 );
}
```

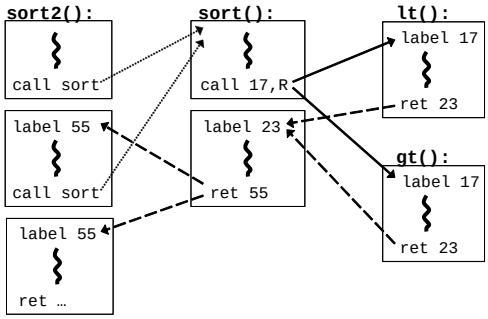

Fig. 1. Example program fragment and an outline of its CFG and CFI instrumentation.

`sort2`. Therefore, the CFI instrumentation includes two IDs in the body of `sort2`, and an ID-check when returning from `sort`, arbitrarily using `55` as the ID bit pattern. (Here, we do not specify to which of the two callsites `sort` must return; Section 5 shows how to guarantee that each return goes to the most recent callsite, by using a protected shadow call stack.) Similarly, because `sort` can call either `lt` or `gt`, both comparison functions start with the ID `17`; and the `call` instruction, which uses a function pointer in register `R`, performs an ID-check for `17`. Finally, the ID `23` identifies the block that follows the comparison callsite in `sort`, so both comparison functions return with an ID-check for `23`.

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

3.2 CFI Instrumentation Code

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

Figures 2 and 3 show example x86 CFI instrumentation of ID-checks and IDs, respectively. The figures give two alternative forms of ID-checks and IDs, showing both their actual x86 opcode bytes and x86 assembly code equivalents. The figures use as the ID the 32-bit hexadecimal value `12345678`. The source (shown in Figure 2) is a computed jump instruction `jmp ecx`, whose destination (shown in Figure 3) 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 in order to avoid a race condition (see Section 4.1). The code sequences for ID-checks overwrite the x86 processor flags and, in (b), a register is assumed available for use; Section 4 explains why this behavior is reasonable.

In alternative (a), the ID is inserted as data before the destination `mov` instruction, and

```
Bytes (opcodes)          x86 assembly code              Comment
FF E1                    jmp  ecx                       ; a computed jump instruction

                                 can be instrumented as (a):

81 39 78 56 34 12        cmp  [ecx], 12345678h          ; compare data at destination
75 13                    jne  error_label               ; if not ID value, then fail
8D 49 04                 lea  ecx, [ecx+4]              ; skip ID data at destination
FF E1                    jmp  ecx                       ; jump to destination code

                              or, alternatively, instrumented as (b):

B8 77 56 34 12           mov  eax, 12345677h            ; load ID value minus one
40                       inc  eax                       ; increment to get ID value
39 41 04                 cmp  [ecx+4], eax              ; compare to destination opcodes
75 13                    jne  error_label               ; if not ID value, then fail
FF E1                    jmp  ecx                       ; jump to destination code
```

Fig. 2. Example CFI instrumentations of an x86 computed jump instruction.

```
Bytes (opcodes)          x86 assembly code              Comment
8B 44 24 04              mov  eax, [esp+4]              ; first instruction
...                                                     ; of destination code

                                 can be instrumented as (a):

78 56 34 12              DD 12345678h                   ; label ID, as data
8B 44 24 04              mov  eax, [esp+4]              ; destination instruction
...

                              or, alternatively, instrumented as (b):

3E 0F 18 05 78 56 34 12  prefetchnta [12345678h]        ; label ID, as code
8B 44 24 04              mov  eax, [esp+4]              ; destination instruction
...
```

Fig. 3. Example CFI instrumentations of a valid destination for an x86 computed jump.

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

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

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

3.3 Assumptions

In our design, CFI enforcement provides protection even against powerful adversaries that control the data memory of the executing program. The machine-code instruction sequences that implement ID-checks and IDs do not rely on the integrity of data memory.

It is however critical that three assumptions hold. These three assumptions are:

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

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

NXD. Non-Executable Data: It must not be possible for the program to execute data as if it were code. Otherwise, an attacker could cause the execution of data that is labeled with the expected ID. NXD is supported in hardware on the latest x86 processors, and Windows XP SP2 uses this support to enforce the separation of code and data [Microsoft Corporation 2004]. NXD can also be implemented in software [PaX Project 2004]. By itself (without CFI), NXD thwarts some attacks, but not those that exploit pre-existing code, such as "jump-to-`libc`" attacks (see Section 4.3).

Somewhat weaker assumptions may sometimes do. In particular, even without NXD, inlined CFI enforcement may be successful as long as the IDs are randomly chosen from a sufficiently large set; then, if attackers do not know the particular IDs chosen, ID-checks will probably fail whenever data execution is attempted. This "probabilistic" defense is similar to that provided by StackGuard and other mitigation mechanisms based on secrets [Cowan et al. 2003; Cowan et al. 1998; Xu et al. 2003]. Since a lucky, persistent, or knowledgeable attacker will still succeed [Shacham et al. 2004; Sovarel et al. 2005; Tuck et al. 2004], we do not favor this CFI variant and do not discuss it further. We believe that CFI should be supported by either hardware or software NXD; Section 5 shows how CFI enforcement can be integrated with one particular software implementation of NXD.

The assumptions can be somewhat problematic in the presence of self-modifying code, runtime code generation, and the unanticipated dynamic loading of code. Fortunately, most software is rather static—either statically linked or with a statically declared set of dynamic libraries. For example, although the Apache web server is a complex, extensible software system, configuration files bound the set of its loadable modules prior to the start of execution. Similarly, for the Outlook email client, the Windows registry bounds the set of loadable components. Nevertheless, we have considered expanding inlined CFI enforcement with the goal of handling runtime code generation and other dynamic additions of code.

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

change code memory—in contradiction with NWC and possibly also in violation of UNQ.[1]

In general, assumptions are often vulnerabilities. When assumptions are invalidated somehow, security guarantees are diminished or void. It is therefore important to justify assumptions (as we do for NXD, for instance) or at the very least to make them explicit, to the extent possible. Of course, we recognize that, in security, any set of assumptions is likely to be incomplete. We focus on the assumptions that we consider most relevant on the basis of analysis and past experience, but for example neglect the possibility that transient hardware faults might affect instruction semantics in arbitrary ways (e.g., [Oh et al. 2002; Reis et al. 2005; Venkatasubramanian et al. 2003]).

3.4 On CFG Precision and Destination Equivalence

Preferably, control-flow enforcement should be as precise as possible. However, even the reliance of CFI on a finite CFG implies a lack of precision. In particular, a finite CFG does not capture the dynamic execution call stack; we address this limitation in Section 5.4. Furthermore, without some care, schemes based on IDs and ID-checks may be more permissive than necessary.

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

One strategy for increasing precision is code duplication. For instance, two separate copies of the function `strcpy` can target two different destination sets when they return. In general, code duplication can be used for eliminating the possibility of overlapping but different destination sets. (Specifically, we can prove that a simple construction that splits CFG nodes into multiple nodes always yields graphs in which overlapping destination sets are identical.) This approach, in the limit, amounts to complete function inlining, apart from recursive calls; it has been used in several intrusion detection implementations (e.g., [Gopalakrishna et al. 2005]).

Alternatively, refining the instrumentation is also a good option for increasing precision. For example, more than one ID can be inserted at certain destinations, or ID-checks can sometimes compare against only certain bits of the destination IDs. We have implemented such a scheme as part of an exploration of architectural support for security [Budiu et al. 2006].

Of course, the assumption of Section 3.1 can also be made true by adding edges to the CFG, thus losing precision. In practice, this alternative can often be satisfactory: even a coarse CFI instrumentation with only one ID value—or with one ID value for the start of functions and another ID value for valid destinations for function returns—will yield sig-

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

nificant guarantees. For instance, that instrumentation will prevent jumps into the middle of functions, which are necessary for some exploits.

3.5 Phases of Inlined CFI Enforcement

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

The first phase, the construction of the CFGs used for CFI enforcement, may give rise to tasks that can range from program analysis to the specification of security policies. Fortunately, a practical implementation may use standard control-flow analysis techniques (e.g., [Aho et al. 1985; Atkinson 2002; Wagner and Dean 2001]), for instance at compile time. Section 4 describes how our x86 implementation applies these techniques by analyzing binaries (rather than source code).

After CFI instrumentation (perhaps at installation time), another mechanism can establish the UNQ assumption. Whenever software is installed or modified, IDs can be updated to remain unique, as is done with pre-binding information in some operating systems [Apple Computer 2003].

Finally (for example, when a program is loaded into memory and assembled from components and libraries), a CFI verification phase can statically validate direct jumps and similar instructions, the proper insertion of IDs and ID-checks, and the UNQ property. This last verification step has the significant benefit of making the trustworthiness of inlined CFI enforcement be independent of the complexity of the previous processing phases. The verification can be seen as a special case of PCC proof-checking, where the instrumentation obviates the need for explicit logical proofs. Only the verification is required for establishing CFI; design or implementation flaws in the instrumentation do not compromise security.

4. A PRACTICAL CFI IMPLEMENTATION

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

4.1 The Implementation

We have implemented inlined CFI enforcement for Windows on the x86 architecture. Our implementation relies on Vulcan [Srivastava et al. 2001], a mature, state-of-the art instrumentation system for x86 binaries that requires neither recompilation nor source-code access. This system addresses the challenges of machine-code rewriting in a practical fashion—as evidenced by its regular application to software produced by Microsoft. Thereby, despite being only a prototype, our implementation of inlined CFI enforcement is both practical and realistic.

Our implementation uses Vulcan for building a CFG of the program being instrumented. This CFG construction correctly handles x86 instructions that perform computed control-flow transfers—including function returns, calls through function pointers, and instructions emitted for switch statements and dynamic dispatch like that of C++ vtables. Our CFG is conservative in that each computed `call` instruction may go to any function whose address is taken; we discover those functions with a flow-insensitive analysis of relocation entries in the binary. (Thus, our implementation can deal even with programs that persist

Bytes (opcodes)	x86 assembly code	Comment
FF 53 08	call [ebx+8]	; call a function pointer

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 43 08                 mov  eax, [ebx+8]       ; load pointer into register
3E 81 78 04 78 56 34 12  cmp  [eax+4], 12345678h ; compare opcodes at destination
75 13                    jne  error_label        ; if not ID value, then fail
FF D0                    call eax                ; call function pointer
3E 0F 18 05 DD CC BB AA  prefetchnta [AABBCCDDh] ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 0C 24                 mov  ecx, [esp]         ; load address into register
83 C4 14                 add  esp, 14h           ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA  cmp  [ecx+4], AABBCCDDh ; compare opcodes at destination
75 13                    jne  error_label        ; if not ID value, then fail
FF E1                    jmp  ecx                ; jump to return address
```

Fig. 5. Our CFI implementation of a function return.

function pointers in files, but cannot deal with programs such as interactive debuggers that turn arbitrary strings into addresses.) Our implementation is simplified by certain Windows particulars: threads are implemented by the kernel, there are no signals like those of Unix, and Windows binaries provide a "SafeSEH" static list of all possible runtime exception handlers. Other CFG intricacies, such as `setjmp` and `longjmp`, are addressed using techniques from the programming-languages and the intrusion-detection literatures [Atkinson 2002; Feng et al. 2003; Gopalakrishna et al. 2005; Wagner and Dean 2001].

Figures 4 and 5 show, respectively, how our CFI implementation rewrites the x86 machine code for an indirect function call and a corresponding function return. The destination of the `call` instruction is stored in memory at address `ebx+8`; the argument `10h` makes the `ret` instruction also pop 16 bytes of parameters off the stack. Next we explain some of the details of the rewritten code. On x86, CFI instrumentation can implement IDs in various ways (e.g., by successive opcodes that add and subtract the same constant). Our prototype, like alternative (b) of Section 3.2, uses prefetch instructions for IDs. Our ID-checks, however, take after the other alternative of Section 3.2: a `cmp` instruction directly compares against the destination ID bit pattern—and, hence, an infinite loop of the ID-check opcode bytes `3E...D0` is possible. (We do not regard such loops as a serious failure of CFI, since an attacker that controls all of memory invariably has many ways of causing infinite loops.) In order to avoid a time-of-check-to-time-of-use race condition [Bishop and Dilger 1996], source instructions where the destination address resides in data memory (such as `ret`) are changed to a `jmp` to an address in a register. Without this precaution, a destination address would be subject to corruption between the time of a successful ID-check and that of the corresponding computed control-flow transfer, so control may flow to a destination address other than the one that has been checked. If an ID-check fails, our implementation immediately aborts execution by using a Windows mechanism

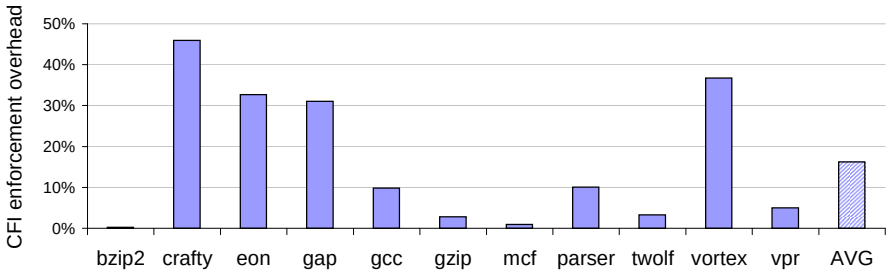

Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

for reporting security violations.

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

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

4.2 Measurements

We measured the overhead of our inlined CFI enforcement on some of the common SPEC computation benchmarks [Standard Performance Evaluation Corporation 2000]. We performed all the experiments in this paper on Windows XP SP2 in “Safe Mode” (where most daemons and kernel modules are disabled). Our hardware was a Pentium 4 x86 processor at 1.8GHz with 512MB of RAM. The target programs were compiled with Microsoft Visual C++ 7.1 using full optimizations. For SPEC, the inputs were the complete reference datasets and the output was validated as the correct result. We report the average of three runs; measurement variance was negligible, with standard deviation of less than one percent.

The CFG construction and CFI instrumentation of each binary took about 10 seconds, with the size of the binary increasing by an average 8%. Figure 6 gives the normalized overhead of CFI enforcement, shown as increase in the running time of each CFI-instrumented benchmark relative to the running time of the original benchmark binaries. On average the benchmarks took 16% longer to execute, with the measured overhead ranging from zero to 45%. This overhead results from a number of factors, including increased cache pressure; the overhead is not simply correlated with the frequency of executed computed control-flow transfers in these benchmarks (see [Hennessy and Patterson 2006, page B-41]).

As shown by Figure 6, our prototype inlined CFI enforcement hardly affects the performance of some programs, but it can cause a substantial slowdown of other programs.

Overall, the measured performance overhead seems tolerable, even though we have not yet explored most of the optimizations possible in x86 CFI instrumentation. Because of CFI verification, such further optimization should reduce overhead without making CFI enforcement less trustworthy.

Moreover, the performance overhead of CFI enforcement is competitive with—or even better than—the cost of most comparable techniques that aim to mitigate security vulnerabilities (e.g., [Cowan et al. 2003; Kiriansky et al. 2002; Ruwase and Lam 2004]). For instance, the overhead of Program Shepherding is more than 100% for the benchmark program `crafty` on Windows; the corresponding CFI enforcement overhead is 45%, and this is our highest measured overhead. Similarly, the overhead of Program Shepherding is more than 660% for `gcc` on Windows, and can be brought down to 35% only by exposing the security mechanism itself to attack; the corresponding CFI enforcement overhead is under 10%.

Note that the SPEC benchmarks focus on CPU-intensive programs with integer arithmetic. CFI will cause relatively less overhead for I/O-driven server workloads. For example, one might expect to see an even smaller performance impact on FTP than on SPEC (as in [Xu et al. 2002]).

4.3 Security-Related Experiments

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

Even so, in order to assess the effectiveness of CFI, we examined by hand some well-known security exploits (such as those of the Blaster and Slammer worms) as well as several recently reported vulnerabilities (such as the Windows ASN.1 and GDI+ JPEG flaws). CFI would not have prevented Nimda and some similar exploits that rely on the incorrect parsing of input strings, such as URLs, to cause the improper launch of the `cmd.exe` shell or some other dangerous executable (see also [Chen et al. 2005]). On the other hand, CFI would have prevented all the other exploits that we studied because, in one way or another, they all endeavored to deviate from the expected control flow. Many exploits performed a "jump-to-`libc`" control transfer from a program point where this jump was not expected. Often this invalid control transfer was attempted through heap overflows or some form of pointer subterfuge (of the kind recently described by Pincus and Baker [2004]).

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

As a concrete example, let us consider the published attack on the GDI+ JPEG flaw in Windows [Florio 2004]. This attack starts by causing a memory corruption, overwriting a global variable that holds a C++ object pointer. When this pointer is later used for calling a virtual destructor, the attacker has the possibility of executing code of their choice. A CFI ID-check at this callsite can prevent this exploit, for instance by restricting valid destinations to the C++ virtual destructor methods of the GDI+ library.

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

```
regular_qsort:
   ...
   push   ebx
   mov    eax, esi
   call   shortsort
   add    esp, 0Ch
   ...
   push   edi             ; an attack is
   push   ebx             ; possible by
   call   [esp+comp_fp]   ; going to X
   add    esp, 8
   test   eax, eax
   jle    label_lessthan
   ...

regular_library_function:
   mov    edi,edi
   push   ebx
   mov    ebx,esp
   push   ecx
   ...
   pop    ebp
X: mov    esp,ebx
   pop    ebx
   ret
```

```
qsort_with_cfi:
...
push   ebx
mov    eax, esi
call   shortsort
prefetchnta [AABBCCDDh]
add    esp, 0Ch
...
push   edi
push   ebx
mov    eax, [esp+comp_fp]
cmp    [eax+4], 12345678h  ; CFI check
jne    error label         ; prevents
call   eax                 ; going to X
prefetchnta [AABBCCDDh]
add    esp, 8
test   eax, eax
jle    label_lessthan
...
```

Fig. 7. Fragments of machine-code for the `median` example.

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

This code is vulnerable—and can be exploited by an attacker that controls the inputs—even on systems that use deployed mitigation techniques such as stack canaries and support for non-executable data. Specifically, we have constructed actual exploits for this vulnerability that work even on Windows XP SP2 with x86 hardware NXD support and with the Windows analogue of StackGuard [Cowan et al. 1998]. One exploit is based on a traditional stack-based buffer overflow; others work via C++ vtables and the heap. CFI enforcement thwarts all these exploits.

Figure 7 shows the original, vulnerable machine code relevant to the stack-based exploit. This exploit is enabled when a buffer overflow overwrites the comparison-function pointer `cmp` before it is passed to `qsort`. The exploit is triggered when `qsort` tries to call the corrupted argument `cmp`, thereby transferring control to an instruction sequence found in the middle of an existing function (labeled `X` in Figure 7). Executing this instruction sequence sets the stack pointer `esp` to the address of data chosen by the attacker and uses that data in a computed control-flow transfer (a return). The exploit subsequently proceeds through the unwinding of the stack, which holds return addresses and other data chosen by

the attacker. As each stack frame is popped, the return instruction transfers control to the start of a particular, existing library function. This sequence of library-code invocations creates a new, writable page of executable memory, writes code of the attacker's choice to that page, and transfers control to that code. As a result, the attacker gains full control over the system.

Figure 7 also shows the CFI instrumentation of `qsort`. As shown in the figure, the instrumentation includes IDs at return points and ID-checks before computed `call` instructions, but not before direct calls. (Although not shown in the figure, the instrumentation also includes IDs at the start of functions whose address is taken, such as the intended comparison function.) With CFI, the defect in the `median` function does not enable exploits. CFI forbids invalid control transfers into the middle of functions (as well as returns to the start of functions), and it therefore prevents the necessary first step of the exploits and would also prevent subsequent steps. This protection is not dependent on how closely the CFI runtime guarantees correspond to a precise CFG of the program; even a coarse CFG has the desired effect.

For a final set of experiments, we ported to Windows a suite of 18 tests for dynamic buffer-overflow prevention developed by Wilander and Kamkar [2003]. (Wilander and Kamkar were unable to provide us with the source code for two of the 20 tests from their study.) The tests in the original suite concern whether attackers could directly execute shellcode of their choosing. We extended the tests to consider also "jump-to-`libc`" and pointer subterfuge attacks. We computed CFGs for these tests, and applied our CFI instrumentation. With CFI enforcement, none of the exploits in this test suite were successful, because they attempted to deviate from the corresponding CFGs. This result is encouraging, especially since other mitigation techniques have failed these tests [Wilander and Kamkar 2003].

5. BUILDING ON CFI

CFI ensures that runtime execution proceeds along a given CFG, guaranteeing, for instance, that the execution of a typical function always starts at the beginning and proceeds from beginning to end. Thereby, CFI can increase the reliability of any CFG-based technique (for example, strengthening previous techniques against buffer overflows and for intrusion detection [Larochelle and Evans 2001; Wagner and Dean 2001]).

This section describes other applications of CFI, as a foundation for Inlined Reference Monitors (IRMs), for SFI in particular, and for Software Memory Access Control (SMAC), which we introduce here. It also shows how to tighten CFI enforcement by relying on either SMAC or standard x86 hardware support. A follow-up paper [Erlingsson et al. 2006] describes a comprehensive protection system, named XFI, based on the material in this section.

5.1 CFI as a Foundation for IRMs

IRMs enforce security policies by inserting into subject programs the code for validity checks and also any additional state that is needed for enforcement [Erlingsson and Schneider 2000]. IRMs require that the subject program can neither circumvent the inserted validity checks nor subvert the added state. By constraining the CFG enforced by CFI, the first of these requirements is easily satisfied. Further, SMAC (discussed below) supports isolated data memory regions in which the added IRM state can be safely kept. Thus, CFI and SMAC greatly facilitate the implementation of IRMs.

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

Fig. 8. Leveraging CFI for optimizations: hoisting an SFI check out of a loop.

In particular, CFI can contribute to the IRM enforcement of security policies that restrict a program's use of the underlying operating system (for instance, preventing files with some filenames from being written) [Provos 2003]. Such policies are often necessary; many of their existing implementations modify operating systems, something that CFI enables us to avoid. With CFI, it is easy to enumerate those points in a program where system calls can be made. At each such point, an IRM validity check can be inserted, and CFI can ensure that the check cannot be circumvented.

5.2 Faster SFI

Software Fault Isolation (SFI) is one particular type of IRM designed to emulate traditional memory protection. In SFI, code is inserted at each machine-code instruction that accesses memory to ensure that the target memory address lies within a certain range [Erlingsson and Schneider 1999; McCamant and Morrisett 2005; Small 1997; Wahbe et al. 1993].

Much as in Section 5.1, CFI makes SFI instrumentation non-circumventable. CFI can also reduce SFI overhead. For instance, the guarantees about control flow remove the need to check memory addresses in local variables repeatedly. Figure 8 demonstrates one such optimization. The figure shows a C function that sums the contents of an array, and the first two basic blocks of the x86 machine code that a compiler might emit for this function. (The start of the first basic block is elided.) The machine code includes an `and` instruction that masks off the top bits from the base address of the array, constraining the array to reside at an address whose top eight bits are either `00h` or `20h`. As long as the low memory (whose addresses start with `00h`) is inaccessible, this use of an `and` instruction can establish several disjoint, isolated memory regions as demonstrated in PittSFIeld, a recent, efficient x86 SFI implementation [McCamant and Morrisett 2005].

The SFI literature is full of other optimizations that simplify the inserted checks. For example, checks can often be eliminated when memory is accessed through a register plus a small constant offset, as long as inaccessible "guard pages" are placed before and after permitted memory ranges. This optimization is especially useful for accesses to local, stack variables, such as reading the value at `esp+8` in Figure 8. However, the weak control-flow guarantees of past SFI implementations make it difficult to reason about program behavior and, partly as a result, past optimizations have sometimes led to vulnerabilities [Erlingsson and Schneider 1999; McCamant and Morrisett 2005].

CFI makes optimizations more robust and enables many new ones. For the code in Figure 8, CFI allows the `and` instruction to be hoisted out of the loop; thus, at runtime, a single masking operation suffices for checking all memory accesses into the array. Past implementations of SFI require a masking operation to accompany each execution of the `add` instruction, because a computed jump might result in that instruction executing with

arbitrary values in registers `esi` and `ecx`. CFI precludes such computed jumps, and with CFI it is easy to see that loop execution does not change `esi` and increments `ecx` from a base of zero.

These optimizations can result in a striking overhead reduction. The SFI literature includes measurements of three systems for x86: Vino's MiSFIT [Small 1997], x86 SASI [Erlingsson and Schneider 1999], and PittSFIeld [McCamant and Morrisett 2005]. For comparison, we applied CFI and SFI, with the optimizations of Figure 8, to two benchmark programs, `hotlist` and the C reference implementation of MD5. The `hotlist` benchmark searches a linked list of integers for a particular value [Small 1997]. For `hotlist`, MiSFIT and SASI produce 140% and 260% overheads, respectively, when both memory reads and writes are restricted. Our corresponding measurement shows only 22% overhead. For MD5, the reported performance overheads for PittSFIeld and MiSFIT range from 23% to 50% [McCamant and Morrisett 2005; Small 1997]. Our corresponding measurement shows only 4.7% overhead.

For this preliminary investigation of SFI, we performed some of the machine-code rewriting by hand on the two benchmark programs. As is common in previous work on SFI, we also made several simplifying assumptions about memory layouts, for example that low memory is inaccessible. In many existing systems, those assumptions cannot be fully satisfied. For useful, realistic memory protection, the rewriting process should be fully automated, and those assumptions should be removed. The SFI policy should also be revisited. We have addressed these concerns in the XFI protection system [Erlingsson et al. 2006].

5.3 SMAC: Generalized SFI

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

CFI can help with SMAC optimizations, much as it does for SFI optimizations; conversely, SMAC can help in eliminating CFI assumptions. SMAC can remove the need for NWC, by disallowing writes to certain memory addresses, and for NXD, by preventing control flow outside those addresses. This synergy between CFI and SMAC is not a circular-reasoning fallacy, as we demonstrate in the formal treatment of CFI with SMAC (see Section 6.5).

Figure 9 shows SMAC instrumentation that can guarantee that only code is executed. As in Figure 8, an `and` instruction masks off the top bits of the destination addresses of computed x86 function calls and returns. Thus, code memory is restricted to addresses whose top eight bits are `40h` (provided that addresses that start with `00h` are invalid). To ensure NWC and NXD for simple regions of code, stack, and data, the SMAC checks can be as straightforward as this single `and` instruction.

Alternatively, the SMAC checks might embody elaborate policies, and allow arbitrary layouts of data and code memory regions, although the code for such checks is likely to be more complex and less efficient than that of Figure 9. In this paper, since it suffices for our

```
call  eax                ; call a function pointer (destination address)
```

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

```
and  eax, 40FFFFFFh      ; mask to ensure address is in code memory
cmp  [eax+4], 12345678h  ; compare opcodes at destination
jne  error_label         ; if not ID value, then fail
call eax                 ; call function pointer
prefetchnta [AABBCCDDh]  ; label ID, used upon the return


ret                      ; return (popping address off the stack)
```

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

```
mov  ecx, [esp]          ; load return address into a register
and  ecx, 40FFFFFFh      ; mask to ensure it is a code memory address
cmp  [ecx+4], AABBCCDDh  ; compare opcodes at destination
jne  error_label         ; if not ID value, then fail
add  esp, 4h             ; discard return address value off the stack
jmp  ecx                 ; jump to return address in the register
```

Fig. 9. Instrumentation of x86 `call` and `ret`, with CFI and SMAC.

immediate purposes, we follow the SFI literature and focus on coarser-grained memory protection.

5.4 A Protected Shadow Call Stack

Because CFI concerns a finite, static CFG, it cannot ensure that a function call returns to the callsite most recently used for invoking the function. Complementing CFI with the runtime call stack (see [Chiueh and Hsu 2001; Frantzen and Shuey 2001; Giffin et al. 2002; 2004; Nebenzahl and Wool 2004; Prasad and Chiueh 2003]) can guarantee this property and increase the precision of CFI enforcement. However, if CFI is to rely on runtime information such as a call stack, the information should not be maintained in unprotected memory, as the ordinary call stack usually is, since the attacker may corrupt or control unprotected memory. Therefore, a protected shadow call stack is required. The assumption that attackers cannot modify this stack directly is necessary, but not sufficient. It is also crucial to guard the stack against corruption that may result from program execution.

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

In this section we focus on an alternative implementation strategy. The resulting implementation is even simpler and more efficient than one that employs SMAC. It leverages the CFI guarantees and standard x86 hardware support. Specifically, we maintain the shadow call stack in an isolated x86 segment[2]. With CFI, static verification can ensure that a par-

[2]The x86 architecture allows multiple *segments* to exist simultaneously within an application. A segment is a specified region of memory, named using an ordinal *selector*. A segment is adopted by loading its ordinal into a *segment register*; there are six such registers, of which some are rarely, if ever, used in modern application

```
  call  eax               ; call a function pointer (destination address)
```

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

```
  add  gs:[0h], 4h        ; add 4 to the index to the top of the shadow call stack
  mov  ecx, gs:[0h]       ; load the index into a register
  mov  gs:[ecx], LRET     ; store the return address at the top
  cmp  [eax+4], 12345678h ; compare opcodes at destination
  jne  error_label        ; if not ID value, then fail
  call eax                ; call function pointer
LRET: ...
```

```
  ret                     ; return (popping address off the stack)
```

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

```
  mov  ecx, gs:[0h]       ; get the index to the top of the shadow call stack
  mov  ecx, gs:[ecx]      ; load the return address from the top into a register
  sub  gs:[0h], 4h        ; subtract 4 from the index
  add  esp, 4h            ; discard redundant return address from the normal stack
  jmp  ecx                ; jump to return address in the register
```

Fig. 10. Instrumentation of x86 `call` and `ret`, with CFI and a protected shadow call stack.

ticular segment register, or segment selector, is used properly by the instrumentation code for call and return instructions, and that only this instrumentation code accesses the corresponding segment. Without CFI, on the other hand, it is extremely difficult to trust the use of segments in an x86 machine-code sequence of non-trivial size. For instance, the opcodes for loading an improper segment selector might be found within basic blocks in system library code, or even within the opcodes of a long, multi-byte instruction; without CFI, an attacker might be able to direct execution to those places.

Figure 10 shows how we use segments in our instrumentation. The segment register `gs` always points to the memory segment that holds the shadow call stack and which has been created to be isolated and disjoint from other accessible memory segments. On Windows, `gs` is unused in application code; therefore, without limitation, CFI verification can statically preclude its use outside this instrumentation code. As shown in the figure, the instrumentation code maintains (in memory location `gs:[0h]`) an offset into this segment that always points to the top of the stack. The use of the protected shadow call stack implies that each return goes to the correct destination, so no ID-checks are required on returns in this instrumentation code.

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

---

code. All memory accesses are performed relative to a segment specified by a segment register; the instructions determine which segment register is to be used, either implicitly or explicitly. On most popular operating systems, user-level code can specify memory regions for its own local segments, which are then context-switched with the application.

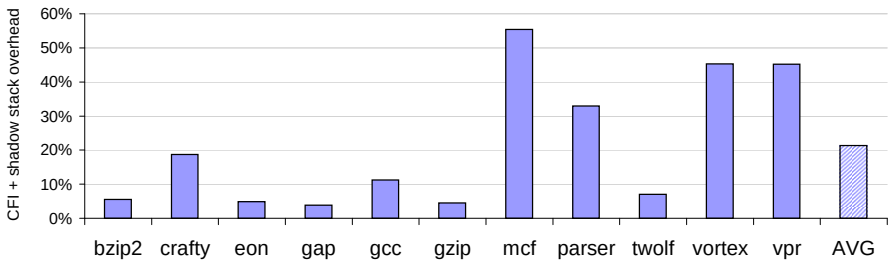

Fig. 11. Execution overhead of inlined CFI enforcement with a protected shadow call stack on SPEC2000 benchmarks.

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

We have implemented a protected shadow call stack for Windows on the x86 architecture, relying on segments and CFI, as outlined above. Figure 11 shows detailed performance measurements for the SPEC benchmarks. We observed only a modest performance overhead for our CFI-protected shadow call stack instrumentation: on average 21%, with 5% for `gzip` and 11% for `gcc`. The overhead includes that of CFI enforcement without the unnecessary ID-checks on returns. These measurements are consistent with the overhead reported in the literature for *unprotected* shadow call stacks (whose integrity could be subverted by attackers) [Prasad and Chiueh 2003]. In contrast, the overhead reported in the literature for *protected* shadow call stacks ranges from 729% (for `gzip`) to 1900% (for `gcc`) [Chiueh and Hsu 2001; Giffin et al. 2002]. While the dramatic improvement that we obtain is partly due to the use of segments, this use of segments is possible only because of CFI.

Once we have a protected shadow call stack, further restrictions on control flow become possible. For example, the control-flow policy could require that every call from a certain function `g` to another function `h` be immediately preceded by a call from a third function `f` to `g`. (Analogous restrictions often appear in the literature on intrusion detection.) Even further restrictions become possible if we keep a protected computation history that records all control transfers. For example, the control-flow policy could then require that a certain function `f` is called at most as often as another function `g`. Such restrictions may sometimes be desirable; for instance, they might prevent some "confused deputy" attacks [Hardy 1988]. On the other hand, we believe that even the simplest kind of CFI enforcement is quite effective at thwarting external attacks that aim to control software behavior.

6. FORMAL STUDY

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

provided hard-to-define protections.

This section begins with a high-level overview of our formal models and results (Section 6.1). This overview should be sufficient for casual readers. Sections 6.2 and 6.3 give details of our models of programs and attackers. Section 6.4 concerns CFI, and Section 6.5 further considers CFI with SMAC. The Appendix contains the corresponding proofs.

6.1 Overview

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

The machine model and the programs that we employ are typical of research on the principles of programming languages. They enable us to consider CFI but exclude virtual memory, dynamic linking, threading, and other sophisticated features found in actual systems. In the machine model, an execution state consists of a code memory M_c, a data memory M_d, an assignment of values to registers R, and a program counter pc. Here, M_c and M_d map addresses to words, R maps register numbers to words, and pc is a word. Essentially, our language is a minor variant of that of Hamid et al. [2002]. We add only an instruction in which an immediate value can be embedded, as a label, and which behaves like a no-op. It is directly analogous to the `label ID` instruction of Section 3.1.

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

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

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

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

As usual in programming-language theory, the operational semantics describes state

transitions by precise rules. For the instruction $add\ r_d, r_s, r_t$, for example, we have that:

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

when $M_c(pc)$ holds $add\ r_d, r_s, r_t$ and $pc + 1$ is in the domain of M_c. The relation $\rightarrow_{\mathrm{n}}$ is a binary relation on states that expresses normal execution steps. For the attacker, we have a rule that enables the following transitions, for all M_c, M_d, $M_d{}'$, R, and pc:

$$(M_c|M_d, R, pc) \rightarrow_{\mathrm{a}} (M_c|M_d{}', R, pc)$$

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

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

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

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

Thus, despite attack steps, the program counter always follows the CFG.

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.

We have yet to pursue a similar formal study for the x86 architecture. Such a study may well be viable (as suggested by recent work [McCamant and Morrisett 2005]), but it may produce diminishing returns, and it would be arduous, not least because of the current absence of a complete formal specification for the x86 architecture.

6.2 The Formal Setting: Programs and their Semantics

Next we define the setting for our formal work.

$Instr ::=$	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

Fig. 12. Instructions.

6.2.1 *Machine Model.* For our machine model, we define words, memories, register files, and states as follows:

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

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

6.2.2 *Instructions.* Our language is that of Hamid et al. [2002] plus a $label$ instruction in which an immediate value can be embedded and which behaves like a no-op. The set of instructions is given in Figure 12. In the figure, w is a word and r_s, r_t, and r_d are registers.

If $Dc(M_c(pc))$=	then $(M_c \mid M_d, R, pc) \rightarrow_{\mathrm{n}}$
$label\ w$	$(M_c \mid M_d, R, pc+1)$, 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 $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$ $(M_c \mid M_d, R, pc+1)$, 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 $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

Fig. 13. Normal steps.

$$\frac{}{(M_c \mid M_d, R_{0-2} \mid R_{3-31}, pc) \rightarrow_{\mathrm{a}} (M_c \mid M_d{}', R_{0-2} \mid R_{3-31}{}', pc)}$$

Fig. 14. Attacker steps.

Thus, instructions may contain words. Like Hamid et al., we omit the routine details of instruction storage and decoding. We assume a function $Dc : Word \rightarrow Instr$ that decodes words into instructions.

6.2.3 *A Semantics of Programs under Attack.* In this subsection we give a first semantics for instructions. Figures 13 and 14 define two binary relations on states, $\rightarrow_{\mathrm{n}}$ and $\rightarrow_{\mathrm{a}}$.

—The relation $\rightarrow_{\mathrm{n}}$ 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 $\rightarrow_{\mathrm{a}}$ 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.
An attack step is quite similar to the possible effect of a computation step in another execution thread (which our model does not represent). In particular, another thread can access all of memory, and can arbitrarily modify data memory. Moreover, registers are

specific to a thread, and the values of the registers of one thread might be affected by another thread only if those values are read from memory (possibly after being "spilled" into memory). An attack step therefore corresponds to a computation step in another thread if the values of general registers may be read from memory but those of distinguished registers are not. On the other hand, for simplicity, an attack step need not be restricted to computable functions.

The relation $\rightarrow$, defined below, is the union of $\rightarrow_{\mathrm{n}}$ and $\rightarrow_{\mathrm{a}}$. Thus, this relation represents a computation step in general, either a normal state transition or one caused by an attacker.

$$\frac{S \rightarrow_{\mathrm{n}} S'}{S \rightarrow S'} \qquad \frac{S \rightarrow_{\mathrm{a}} S'}{S \rightarrow S'}$$

In security, it is important to identify and to support assumptions, as mentioned in Section 3.3. Our definitions embody several assumptions, which we discuss next:

(1) The definition of $\rightarrow_{\mathrm{n}}$ implies NXD (that is, that data cannot be executed as code). Similarly, the definitions of $\rightarrow_{\mathrm{n}}$ and $\rightarrow_{\mathrm{a}}$ imply NWC (that is, that code memory cannot be modified at runtime). As indicated in Section 3.3, NXD and NWC are often reasonable assumptions.

(2) The definition of $\rightarrow_{\mathrm{a}}$ allows for the possibility that the attacker is in control of data memory. As indicated in the introduction, this aspect of the model of the attacker is conservative but unfortunately close to reality.

(3) The definition of $\rightarrow_{\mathrm{a}}$ implies that the attacker cannot modify the distinguished registers r_0, r_1, and r_2. 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. Section 3.3 argues the practicality of such assumptions on registers.

(4) The machine model and the definition of $\rightarrow_{\mathrm{n}}$ 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.

6.2.4 *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 6.2.3. For brevity, and since there is no risk of ambiguity below, we reuse the symbols $\rightarrow_{\mathrm{n}}$, $\rightarrow_{\mathrm{a}}$, and $\rightarrow$.

The resulting, relaxed definition of normal execution steps is in Figure 15. 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 6.2.3. 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.

6.3 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.

If $Dc(M(pc))$=	then $(M, R, pc) \rightarrow_{\mathrm{n}}$
$label\ w$	$(M, R, pc + 1)$
$add\ r_d, r_s, r_t$	$(M, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc + 1)$
$addi\ r_d, r_s, w$	$(M, R\{r_d \mapsto R(r_s) + w\}, pc + 1)$
$movi\ r_d, w$	$(M, R\{r_d \mapsto w\}, pc + 1)$
$bgt\ r_s, r_t, w$	(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. 15. Normal steps (assuming less memory protection).

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 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 $w' \in \mathrm{succ}(w)$.

We need not constrain how the CFG is obtained, or how it matches the executions of the program before instrumentation. 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 $\mathrm{succ}(w_0) = \{w_0 + 1\} \cap \mathrm{dom}(M_c)$.

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

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

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

(5) $Dc(M_c(w_0)) = illegal$ 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 $\mathrm{succ}(w_0) = \mathrm{succ}(w_1)$.

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; its significance is discussed in Section 3.4.

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

6.4 CFI Enforcement (without SMAC)

In this subsection we present and analyze a basic technique for CFI enforcement (without SMAC), using the semantics of Section 6.2.3.

6.4.1 *CFI Enforcement by Instrumentation.* For the sake of trustworthiness, as suggested in Section 3.5, CFI enforcement should preferably depend only on simple, final, static verification steps that check that 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 conditions:

(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 $jmp\ r_0$ and it is preceded by a specific sequence of instructions, as follows:

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

where r_s is some register, $HALT$ is the address of the $illegal$ instruction specified in condition (1), and IMM is the word w such that $Dc(w) = label\ \mathrm{dst}(w_0)$. 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 $bgt\ r_s, r_t, w$ or $jd\ w$ appear anywhere in M_c, then the target address w does not hold a jmp instruction or the occurrences of the instructions

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

that precede a jmp instruction according to condition (3). The target address may hold $addi\ r_0, r_s, 0$. (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.)

We let the predicate $I(M_c, G)$ mean that M_c and its CFG G satisfy the conjunction of the conditions above.

6.4.2 *A Theorem about CFI.* With these definitions, and under the semantics of Section 6.2.3, we can obtain formal results about our instrumentation method.

Here we give 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 $S_i \rightarrow_{\mathrm{a}} S_{i+1}$ 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, which we present in Appendix A.1, 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.

The basic technique for CFI enforcement described in this section depends on NXD. More specifically, Theorem 1 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.

6.5 CFI Enforcement with SMAC

CFI enforcement with SMAC eliminates the need for NXD and allows program execution steps to modify code memory, with the semantics of Section 6.2.4. While it may be viewed as a refinement of our basic technique (perhaps via a simulation relation), in this subsection we present it and study it on its own, as a complete and separate mechanism.

6.5.1 *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 conditions:

(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 $st\ r_0(0), r_s$ and it is preceded by a specific sequence of instructions, as follows:

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

where r_d is some register, w is some offset (a word), and $HALT$ is the address of the $illegal$ instruction specified in condition (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 $jmp\ r_0$ and it is preceded by a specific sequence of instructions, as follows:

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

where r_s is some register, $HALT$ is the address of the $illegal$ instruction specified in condition (1), and IMM is the word w such that $Dc(w) = label\ \mathrm{dst}(w_0)$. This code is a combination of the code for jmp described in Section 6.4 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 6.4, that dynamic target is compared with the $label$ instruction expected statically. The program halts unless all checks succeed.

(5) If $bgt\ r_s, r_t, w$ or $jd\ w$ 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 $addi\ r_0, r_d, w$ and $addi\ r_0, r_s, 0$, respectively.

We let the predicate $I_s(M_c, G)$ mean that M_c and its CFG G satisfy the conjunction of the conditions above.

6.5.2 *A Theorem about CFI with SMAC.* With the relaxed semantics of Section 6.2.4 and the instrumentation of Section 6.5, 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 $S_i \to_{\mathrm{a}} S_{i+1}$ 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, which we present in Appendix A.2, 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.

7. CONCLUSION

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

CFI instrumentation aims to change this situation by embedding within software executables both a control-flow policy to be enforced at runtime and the mechanism for that enforcement. Indeed, 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 [Morrisett et al. 1999], and of efforts to bridge the gaps between high-level languages and actual behavior (e.g., [Abadi 1998; Kennedy 2005]).

CFI is simple, verifiable, and amenable to formal analysis, yielding strong guarantees even in the presence of a powerful adversary. Moreover, inlined CFI enforcement is practical on modern processors, is compatible with most existing software, and has little performance overhead. Finally, CFI provides a useful foundation for the efficient enforcement of security policies.

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. John Wilander kindly provided us with source code that was a basis for some of our security experiments. Milenko Drinić and Andrew Edwards of the Vulcan team assisted our implementation efforts.

REFERENCES

ABADI, M. 1998. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, K. G. Larsen, S. Skyum, and G. Winskel, Eds. Lecture Notes in Computer Science, vol. 1443. Springer-Verlag, Aalborg, Denmark, 868–883. Also Digital Equipment Corporation Systems Research Center report No. 154, April 1998.

ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. 2005. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, K.-K. Lau and R. Banach, Eds. Lecture Notes in Computer Science, vol. 3785. Springer-Verlag, Manchester, U.K., 111–124. A preliminary version appears as Microsoft Research Technical Report MSR-TR-05-17, May 2005.

ABADI, M., BUDIU, M., ÚLFAR ERLINGSSON, AND LIGATTI, J. 2005. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM Press, Alexandria, VA, 340–353. A preliminary version appears as Microsoft Research Technical Report MSR-TR-05-18, February 2005.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1985. *Compilers: principles, techniques, tools*. Addison-Wesley, Reading, MA.

APPLE COMPUTER. 2003. Prebinding notes. `http://developer.apple.com/releasenotes/DeveloperTools/Prebinding.html`.

ATKINSON, D. C. 2002. Call graph extraction in the presence of function pointers. In *Proceedings of the 2nd International Conference on Software EngineeringResearch and Practice*. Las Vegas, NV.

AVIJIT, K., GUPTA, P., AND GUPTA, D. 2004. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *Proceedings of the Usenix Security Symposium*. The Usenix Association, Boston, MA, 45–56.

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

BAUER, L., LIGATTI, J., AND WALKER, D. 2005. Composing security policies with polymer. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. Chicago, 305–314.

BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the Usenix Security Symposium*. 105–120.

BISHOP, M. AND DILGER, M. 1996. Checking for race conditions in file access. *Computing Systems 9*, 2, 131–152.

BRUMLEY, D. AND SONG, D. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the Usenix Security Symposium*. 57–72.

BUDIU, M., ÚLFAR ERLINGSSON, AND ABADI, M. 2006. Architectural support for software-based protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM Press, New York, NY, USA, 42–51.

CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the Usenix Security Symposium*. 177–192.

CHIUEH, T. AND HSU, F. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*. 409–419.

COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. 2001. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the Usenix Security Symposium*. The Usenix Association, San Antonio, TX.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*. 91–104.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Symposium*. The Usenix Association, San Antonio, TX, 63–78.

CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture*.

ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. 2006. XFI: Software guards for system address spaces. In *OSDI'06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. 75–88.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*. IEEE Computer Society, Oakland, CA, 87–95.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 2000. IRM enforcement of java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 246–255.

EVANS, D. AND TWYMAN, A. 1999. Policy-directed code safety. In *Proc. of 1999 IEEE Symposium on Security and Privacy* (Oakland, CA).

FENG, H., KOLESNIKOV, O., FOGLA, P., LEE, W., AND GONG, W. 2003. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*. 62–77.

FENG, H. H., GIFFIN, J. T., HUANG, Y., JHA, S., LEE, W., AND MILLER, B. P. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 194–210.

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

FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. 1996. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*. 120–128.

FRANTZEN, M. AND SHUEY, M. 2001. StackGhost: Hardware facilitated stack protection. In *Proceedings of the Usenix Security Symposium*. 55–66.

GIFFIN, J. T., JHA, S., AND MILLER, B. P. 2002. Detecting manipulated remote call streams. In *Proceedings of the Usenix Security Symposium*. 61–79.

GIFFIN, J. T., JHA, S., AND MILLER, B. P. 2004. Efficient context-sensitive intrusion detection. In *NDSS '04: Proceedings of the Network and Distributed System Security Symposium*.

GOPALAKRISHNA, R., SPAFFORD, E. H., AND J.VITEK. 2005. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*. 18–31.

GOVINDAVAJHALA, S. AND APPEL, A. W. 2003. Using memory errors to attack a virtual machine. In *Proceedings of the IEEE Symposium on Security and Privacy*. 154–165.

HAMID, N., SHAO, Z., TRIFONOV, V., MONNIER, S., AND NI, Z. 2002. A Syntactic Approach to Foundational Proof-Carrying Code. Tech. Rep. YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University.

HARDY, N. 1988. The confused deputy. *ACM Operating Systems Review 22*, 4, 36–38.

HARRIS, L. C. AND MILLER, B. P. 2005. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News 33*, 5, 63–68.

HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer Architecture: A Quantitative Approach*, 4 ed. Morgan Kaufmann Publishers, San Mateo, CA.

KENNEDY, A. 2005. Securing the .NET programming model. APPSEM II Workshop. Available at `http://research.microsoft.com/~akenn/sec/index.html`.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the Usenix Security Symposium*. 191–206.

KIROVSKI, D. AND DRINIC, M. 2004. POPI — a novel platform for intrusion prevention. In *Proceedings of the International Symposium on Microarchitecture*.

LAM, L. AND CHIUEH, T. 2004. Automatic extraction of accurate application-specific sandboxing policy. In *RAID '04: Proceedings of the International Symposium on Recent Advances in Intrusion Detection*. 1–20.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*. 177–190.

LARSON, E. AND AUSTIN, T. 2003. High coverage detection of input-related security faults. In *Proceedings of the Usenix Security Symposium*. 121–136.

MCCAMANT, S. AND MORRISETT, G. 2005. Efficient, verifiable binary sandboxing for a CISC architecture. Tech. Rep. MIT-LCS-TR-988, MIT Laboratory for Computer Science.

MICROSOFT CORPORATION. 2004. Changes to functionality in Microsoft Windows XP SP2: Memory protection technologies. `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3, 527–568.

NEBENZAHL, D. AND WOOL, A. 2004. Install-time vaccination of Windows executables to defend against stack smashing attacks. In *Proceedings of the IFIP International Information Security Conference*.

NECULA, G. 1997. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. 106–119.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*. 128–139.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002. Control flow checking by software signatures. *IEEE Transactions on Reliability 51*, 2. Special Section on Fault Tolerant VLSI Systems.

PAX PROJECT. 2004. The PaX project. `http://pax.grsecurity.net/`.

PINCUS, J. AND BAKER, B. 2004. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2*, 4, 20–27.

PRASAD, M. AND CHIUEH, T. 2003. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the Usenix Technical Conference*. 211–224.

PROVOS, N. 2003. Improving host security with system call policies. In *Proceedings of the Usenix Security Symposium*. 257–272.

REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*.

RUWASE, O. AND LAM, M. S. 2004. A practical dynamic buffer overflow detector. In *Proceedings of Network and Distributed System Security Symposium*.

SCHNEIDER, F. B. 2000. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1, 30–50. Previous version published as Cornell University Technical Report TR98-1664, January, 1998.

SCOTT, K. AND DAVIDSON, J. 2002. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*. 209.

SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*. 144–155.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*.

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

SOVAREL, A. N., EVANS, D., AND PAUL, N. 2005. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proceedings of the Usenix Security Symposium*. 145–160.

SRIVASTAVA, A., EDWARDS, A., AND VO, H. 2001. Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. Tech. Rep. WRL Research Report 94/2, Digital Equipment Corporation.

STANDARD PERFORMANCE EVALUATION CORPORATION. 2000. SPEC CPU2000 benchmark suite. `http://www.spec.org/osg/cpu2000/`.

SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 85–96.

TUCK, N., CALDER, B., AND VARGHESE, G. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the International Symposium on Microarchitecture*.

VENKATASUBRAMANIAN, R., HAYES, J. P., AND MURRAY, B. T. 2003. Low-cost on-line fault detection using control flow assertions. In *Proceedings of 9th IEEE International On-Line Testing Symposium*.

WAGNER, D. AND DEAN, D. 2001. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*. 156–169.

WAGNER, D. AND SOTO, P. 2002. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the ACM Conference on Computer and Communications Security*. 255–264.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5, 203–216.

WILANDER, J. AND KAMKAR, M. 2003. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*.

WINWOOD, S. AND CHAKRAVARTY, M. M. T. 2005. Secure untrusted binaries—provably! Tech. Rep. UNSW-CSE-TR-0511, School of Computer Science and Engineering, University of New South Wales, Australia.

XU, J., KALBARCZYK, Z., AND IYER, R. K. 2003. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable and Distributed Systems*.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. 2002. Architecture support for defending against buffer overflow attacks.

A. APPENDIX: PROOFS

This Appendix contains proofs for the main claims of Section 6.

A.1 Proof of CFI Enforcement (without SMAC)

First we state a simple proposition about the effects of the sequence of instructions that precede each jmp instruction in instrumented programs.

Proposition 3
Let S_0, S_1, S_2, S_3, S_4, and S_5 be states with code memory M_c such that

$$Dc(M_c(S_0.pc)) = addi\ r_0, r_s, 0$$
$$Dc(M_c(S_1.pc)) = ld\ r_1, r_0(0)$$

$$\begin{aligned} Dc(M_c(S_2.pc)) &= movi\ r_2, IMM \\ Dc(M_c(S_3.pc)) &= bgt\ r_1, r_2, HALT \\ Dc(M_c(S_4.pc)) &= bgt\ r_2, r_1, HALT \\ Dc(M_c(S_5.pc)) &= jmp\ r_0 \end{aligned}$$

and

$$S_0 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_1 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_2 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_3 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_4 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_5$$

where $\rightarrow_{\mathrm{a}}^*$ *is the reflexive transitive closure of* $\rightarrow_{\mathrm{a}}$*. Assume that* $S_5.R(r_0) \in \mathrm{dom}(M_c)$ *and let* $w = S_5.R(r_0)$*. Then we have that* $M_c(w) = IMM$ *and, if* $I(M_c, G)$*, then* $Dc(M_c(w)) = label\ \mathrm{dst}(M_c, G, S_5.pc)$*.*

The proposition says what the instructions in question do, when executed from beginning to end in a straight line. It allows for the possibility of attack steps between normal steps. The first part of the conclusion, $M_c(S_5.R(r_0)) = IMM$, is an immediate consequence of the operational semantics, which ensures that attack steps do not affect distinguished registers. In case $I(M_c, G)$, the definition of the instrumentation predicate I yields the second part of the conclusion, $Dc(M_c(S_5.R(r_0))) = label\ \mathrm{dst}(M_c, G, S_5.pc)$.

Using this proposition, we establish the following lemma, of which Theorem 1 is an obvious corollary:

Lemma 4
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* $S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_n$*, with* $n \geq 0$*. Then:*

(1) $S_n.M_c = S_0.M_c$*;*
(2) $S_n.pc \in \mathrm{dom}(S_0.M_c)$*;*

and, if $n > 0$*,*

(3) either $S_{n-1} \rightarrow_{\mathrm{a}} S_n$ *and* $S_n.pc = S_{n-1}.pc$*, or* $S_n.pc \in \mathrm{succ}(S_0.M_c, G, S_{n-1}.pc)$*;*
(4) if there exists $k \in \{0..4\}$ *such that* $Dc(M_c(S_n.pc + k))$ *holds a* jmp *instruction, then* $S_{n-1}.pc = S_n.pc$ *or* $S_{n-1}.pc + 1 = S_n.pc$*.*

The hypotheses are those of Theorem 1, and so is conjunct (3) of the conclusion except that we focus on the last execution step $S_{n-1} \rightarrow S_n$ rather than on an arbitrary execution step $S_i \rightarrow S_{i+1}$. Conjunct (1) means that code memory does not change in the course of execution; it immediately implies that $I(S_n.M_c, G)$. Conjunct (2) means that execution does not leave code memory. Conjunct (4) basically forbids jumps past or into the middle of the checking sequences that precede jmp instructions.

We establish the lemma by complete induction on n (that is, we prove that it holds for a value of n assuming that it holds for all smaller values). For $n = 0$, all conjuncts are either trivially or vacuously true. For $n > 0$, we argue by cases on whether $S_{n-1} \rightarrow S_n$ is an attack step or a normal step. The former case is trivial, since attack steps cannot modify code memory or the program counter, by definition. The latter case is itself by cases on the instruction at $S_{n-1}.pc$. In each of the cases, we verify the four conjuncts of the conclusion. Conjuncts (1) and (2) always follow immediately from the definition of the operational semantics. In the following argument, we rely on conjuncts (1) and (2), writing M_c instead of $S_i.M_c$ (for $i \in \{0..n\}$), and using that $I(M_c, G)$ and that $S_n.pc \in \mathrm{dom}(M_c)$. We consider conjuncts (3) and (4):

—For $label$, add, $addi$, $movi$, ld, and st instructions, we have that $S_n.pc = S_{n-1}.pc + 1$ (by the operational semantics), so conjunct (4) holds. As $S_n.pc \in \mathrm{dom}(M_c)$, we have that $\mathrm{succ}(M_c, G, S_{n-1}.pc) = \{S_n.pc\}$ (by the requirements on CFGs), so conjunct (3) holds as well.

—For a $jd\ w$ instruction, we have that $S_n.pc = w$ (by the operational semantics) and $S_n.pc \in \mathrm{dom}(M_c)$, and therefore $\mathrm{succ}(M_c, G, S_{n-1}.pc) = \{S_n.pc\}$ (by the requirements on CFGs), so conjunct (3) holds. Moreover, for $k \in \{0..4\}$, $Dc(M_c(w + k))$ cannot be a jmp instruction, by the definition of the instrumentation predicate I, so conjunct (4) holds vacuously.

—For a $bgt\ r_s, r_t, w$ instruction, we have that $S_n.pc \in \{S_{n-1}.pc + 1, w\}$ (by the operational semantics) and $S_n.pc \in \mathrm{dom}(M_c)$, and therefore $S_n.pc \in \mathrm{succ}(M_c, G, S_{n-1}.pc)$ (by the requirements on CFGs), so conjunct (3) holds. If $S_n.pc = S_{n-1}.pc + 1$, then conjunct (4) holds immediately. On the other hand, for the case where $S_n.pc = w$, we have that $Dc(M_c(w + k))$ cannot be a jmp instruction for $k \in \{0..4\}$, by the definition of the instrumentation predicate I, so conjunct (4) holds vacuously.

—For a $jmp\ r_s$ instruction, we first observe that r_s must be r_0 and that the instruction must be immediately preceded by

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

according to the definition of the instrumentation predicate I. With the exception of the first of these instructions, none could be at location 0, so none could be the starting point for the execution; similarly, the jmp instruction could not be the starting point for the execution either. Moreover, by induction hypothesis (conjunct (4)), the execution could not have jumped past or into the middle of these instructions. From the operational semantics and conjuncts (1) and (2) we have that $S_n.pc \in \mathrm{dom}(M_c)$ and $I(M_c, G)$. Therefore, Proposition 3 applies, and yields that $Dc(M_c(S_{n-1}.R(r_0))) = label\ \mathrm{dst}(M_c, G, S_{n-1}.pc)$. From the operational semantics we also have $S_n.pc = S_{n-1}.R(r_0)$, so $Dc(M_c(S_n.pc)) = label\ \mathrm{dst}(M_c, G, S_{n-1}.pc)$, which implies that $S_n.pc$ is a destination and that $\mathrm{dst}(M_c, G, S_{n-1}.pc)$ is its ID. Since the instruction at $S_{n-1}.pc$ is a jmp instruction, $\mathrm{dst}(M_c, G, S_{n-1}.pc)$ is the ID of the elements of $\mathrm{succ}(M_c, G, S_{n-1}.pc)$. Hence, we conclude that $S_n.pc \in \mathrm{succ}(M_c, G, S_{n-1}.pc)$, so conjunct (3) holds. Since $S_n.pc$ contains a $label$ instruction, and since $label$ instructions do not appear in checking sequences before jmp instructions, conjunct (4) also follows, vacuously.

—The case of $illegal$ instructions is vacuous, since $S_{n-1} \rightarrow S_n$ can be only an attack step in this case.

A.2 Proof of CFI Enforcement with SMAC

The proof is similar to that of Appendix A.1.

First we state an analogue of Proposition 3:

Proposition 5
Let $S_0, \ldots, S_9$ be states with code memory M_c such that

$$\begin{aligned}
Dc(M_c(S_0.pc)) &= addi\ r_0, r_s, 0\\
Dc(M_c(S_1.pc)) &= movi\ r_1, \max(M_c)\\
Dc(M_c(S_2.pc)) &= movi\ r_2, \min(M_c)\\
Dc(M_c(S_3.pc)) &= bgt\ r_0, r_1, HALT\\
Dc(M_c(S_4.pc)) &= bgt\ r_2, r_0, HALT\\
Dc(M_c(S_5.pc)) &= ld\ r_1, r_0(0)\\
Dc(M_c(S_6.pc)) &= movi\ r_2, IMM\\
Dc(M_c(S_7.pc)) &= bgt\ r_1, r_2, HALT\\
Dc(M_c(S_8.pc)) &= bgt\ r_2, r_1, HALT\\
Dc(M_c(S_9.pc)) &= jmp\ r_0
\end{aligned}$$

and

$$S_0 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_1 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* \ \ldots\ S_8 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_9$$

where $\rightarrow_{\mathrm{a}}^*$ is the reflexive transitive closure of $\rightarrow_{\mathrm{a}}$. Let $w = S_9.R(r_0)$. Then we have that $w \in \mathrm{dom}(M_c)$ and $M_c(w) = IMM$. If $I_s(M_c, G)$ and $S_9.pc \in \mathrm{dom}(M_c)$, then we also have that $Dc(M_c(w)) = label\ \mathrm{dst}(M_c, G, S_9.pc)$.

Much like Proposition 3, this proposition says what the instructions in question do, when executed from beginning to end in a straight line. The first part of the conclusion, $S_9.R(r_0) \in \mathrm{dom}(M_c)$ and $M_c(S_9.R(r_0)) = IMM$, is an immediate consequence of the operational semantics. In case $I_s(M_c, G)$ and $S_9.pc \in \mathrm{dom}(M_c)$, the definition of the instrumentation predicate I_s yields the second part of the conclusion, $Dc(M_c(S_9.R(r_0))) = label\ \mathrm{dst}(M_c, G, S_9.pc)$.

A similar but even simpler proposition concerns st instructions:

Proposition 6
Let $S_0, \ldots, S_5$ be states with code memory M_c such that

$$\begin{aligned}
Dc(M_c(S_0.pc)) &= addi\ r_0, r_d, w\\
Dc(M_c(S_1.pc)) &= movi\ r_1, \max(M_d)\\
Dc(M_c(S_2.pc)) &= movi\ r_2, \min(M_d)\\
Dc(M_c(S_3.pc)) &= bgt\ r_0, r_1, HALT\\
Dc(M_c(S_4.pc)) &= bgt\ r_2, r_0, HALT\\
Dc(M_c(S_5.pc)) &= st\ \ r_0(0), r_s
\end{aligned}$$

and

$$S_0 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_1 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_2 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_3 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_4 \rightarrow_{\mathrm{n}} \rightarrow_{\mathrm{a}}^* S_5$$

where $\rightarrow_{\mathrm{a}}^*$ is the reflexive transitive closure of $\rightarrow_{\mathrm{a}}$. Then we have that $S_5.R(r_0) \in \mathrm{dom}(M_d)$, and therefore that $S_5.R(r_0) \notin \mathrm{dom}(M_c)$.

Using these two propositions, we establish the following lemma, which is analogous to Lemma 4 and of which Theorem 2 is an obvious corollary:

Lemma 7
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 $S_0 \to S_1 \to ... \to S_n$, with $n \geq 0$. Then:

(1) $S_n.M_c = S_0.M_c$;

(2) $S_n.pc \in \mathrm{dom}(S_0.M_c)$;

and, if $n > 0$,

(3) either $S_{n-1} \to_{\mathrm{a}} S_n$ and $S_n.pc = S_{n-1}.pc$, or $S_n.pc \in \mathrm{succ}(S_0.M_c, G, S_{n-1}.pc)$;

(4) if there exists $k \in \{0..8\}$ such that $Dc(M_c(S_n.pc + k))$ holds a jmp instruction, then $S_{n-1}.pc = S_n.pc$ or $S_{n-1}.pc + 1 = S_n.pc$;

(5) if there exists $k \in \{0..4\}$ such that $Dc(M_c(S_n.pc + k))$ holds a st instruction, then $S_{n-1}.pc = S_n.pc$ or $S_{n-1}.pc + 1 = S_n.pc$.

The only substantial novelty with respect to Lemma 4 is conjunct (5). This conjunct basically forbids jumps past or into the middle of the checking sequences that precede st instructions.

Again, we establish the lemma by complete induction on n and, for $n > 0$, we argue by cases on whether $S_{n-1} \to S_n$ is an attack step or a normal step. The former case is trivial, since attack steps still cannot modify code memory or the program counter, by definition. The latter case is itself by cases on the instruction at $S_{n-1}.pc$. In each of the cases, we verify the five conjuncts of the conclusion.

—For $label$, add, $addi$, $movi$, and ld instructions, we have that $S_n.M_c = S_0.M_c$ and $S_n.pc = S_{n-1}.pc + 1$ (by the operational semantics and the induction hypothesis), so conjuncts (1), (4), and (5) hold. By the definition of the instrumentation predicate I_s, the last instruction in $\mathrm{dom}(S_0.M_c)$ is $illegal$, so it cannot be one of those under consideration; moreover, by induction hypothesis, $S_{n-1}.pc \in \mathrm{dom}(S_0.M_c)$. Therefore, we have that $S_n.pc \in \mathrm{dom}(S_0.M_c)$, so conjunct (2) holds. As $S_n.pc \in \mathrm{dom}(S_0.M_c)$, we have that $\mathrm{succ}(S_0.M_c, G, S_{n-1}.pc) = \{S_n.pc\}$ (by the requirements on CFGs), so conjunct (3) holds as well.

—For a $st\ r_d(w), r_s$ instruction, we first observe that $r_d(w)$ must be $r_0(0)$, and that the instruction must be immediately preceded by

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

according to the definition of the instrumentation predicate I_s and by the induction hypothesis, which implies that $S_{n-1}.M_c = S_0.M_c$. With the exception of the first of these instructions, none could be at location 0, so none could be the starting point for the execution; similarly, the st instruction could not be the starting point for the execution either. Moreover, by induction hypothesis (conjunct (5)), the execution could not have jumped past or into the middle of these instructions. Therefore, Proposition 6 applies, and yields that $S_5.R(r_0) \notin \mathrm{dom}(M_c)$, so $S_n.M_c = S_{n-1}.M_c$ by the operational semantics of the st instruction. Hence, by induction hypothesis, conjunct (1) holds. The

argument for the other conjuncts is as in the case of $label$, add, $addi$, $movi$, and ld instructions.

—For a $jd\ w$ instruction, we have that $S_n.M_c = S_0.M_c$ (by the operational semantics and the induction hypothesis), so conjunct (1) holds. In addition, we have $S_n.pc = w$ (by the operational semantics), and $w \in \mathrm{dom}(S_0.M_c)$, by the definition of the instrumentation predicate I_s, so conjunct (2) holds as well. The argument for the other conjuncts is analogous to the corresponding argument for jd instructions in Lemma 4.

—For a $bgt\ r_s, r_t, w$ instruction, we have that $S_n.M_c = S_0.M_c$ (by the operational semantics and the induction hypothesis), so conjunct (1) holds. In addition, we have $S_n.pc \in \{S_{n-1}.pc + 1, w\}$ (by the operational semantics). By the definition of the instrumentation predicate I_s, $S_{n-1}.pc + 1 \in \mathrm{dom}(S_0.M_c)$ (because the bgt instruction cannot be the last instruction in $\mathrm{dom}(S_0.M_c)$) and $w \in \mathrm{dom}(S_0.M_c)$. Therefore, conjunct (2) holds as well. The argument for the other conjuncts is analogous to the corresponding argument for bgt instructions in Lemma 4.

—For a $jmp\ r_s$ instruction, the argument is exactly analogous to that in Lemma 4, with the following refinements. First, conjunct (1) follows from the operational semantics and the induction hypothesis, as in the other cases of this proof. More interestingly, conjunct (2) follows from Proposition 5.

—The case of $illegal$ instructions is vacuous, since $S_{n-1} \rightarrow S_n$ can be only an attack step in this case.