Efficient, Verifiable Binary Sandboxing for a CISC Architecture

Stephen McCamant
Massachusetts Institute of Technology
Computer Science and AI Lab
Cambridge, MA 02139
smcc@csail.mit.edu

Greg Morrisett
Harvard University
Division of Engineering and Applied Sciences
Cambridge, MA 02138
greg@eecs.harvard.edu

Abstract

Executing untrusted code while preserving security requires enforcement of *memory* and *control-flow safety* policies: untrusted code must be prevented from modifying memory or executing code except as explicitly allowed. Software-based fault isolation (SFI) or "sandboxing" enforces those policies by rewriting the untrusted code at the level of individual instructions. However, the original sandboxing technique of Wahbe et al. is applicable only to RISC architectures, and other previous work is either insecure, or has been not described in enough detail to give confidence in its security properties. We present a novel technique that allows sandboxing to be easily applied to a CISC architecture like the IA-32. The technique can be verified to have been applied at load time, so that neither the rewriting tool nor the compiler needs to be trusted. We describe a prototype implementation which provides a robust security guarantee, is scalable to programs of any size, and has low runtime overheads. Further, we give a machine-checked proof that any program approved by the verification algorithm is guaranteed to respect the desired safety property.

Keywords: Software fault isolation, control-flow isolation, binary translation, C, C++, mobile code, inlined reference monitors, separate verification, 386, x86, instruction alignment, formal methods, security proof, ACL2, MiSFIT, PittSFIeld

1 Introduction

A key requirement for many kinds of secure systems is to execute code from an untrusted or less trusted source, while enforcing some policy to constrain the code's actions. The code might come directly from a malicious author, or it might have bugs that allow its execution to be subverted by maliciously chosen inputs. Typically, the system designer chooses some set of legal interfaces for interaction with the code, and the challenge is to ensure that the code's interaction with the rest of the system is limited to those interfaces.

The most common technique for isolating untrusted code is the use of hardware virtual memory protection in the form of an operating system process. Code in one process is restricted to accessing memory only in its address space, and its interaction with the rest of a system is limited to a predefined system call interface. The enforcement of these restrictions is robust and has a low overhead because of the use of dedicated hardware mechanisms such as TLBs; very few restrictions are placed on what the untrusted code can try to do. A disadvantage of hardware protection, however, is that interaction across a process boundary (i.e., via system calls) is course-grained and relatively expensive. Because of this inefficiency and inconvenience, it is still most common for even large applications, servers, and operating system kernels to be constructed to run in a single address space.

A very different technique is to require that the untrusted code be written in a type-safe language such as Java. The language's type discipline limits the memory usage and control flow of the code to well-behaved patterns. This fine-grained restriction makes sharing data between trusted and untrusted components much easier, and has other software engineering benefits. However, type systems have some limitations as a security mechanism. First, they are not directly applicable to code written in unsafe languages, such as C and C++. Second, conventional type systems describe high-level program actions like method calls and field accesses. It is much more difficult to use a type system to constrain code at the same level of abstraction as individual machine instructions; but since it is the actual instructions that will be executed, only a safety property in terms of them would be really convincing.

This paper investigates a code isolation technique that lies between the approaches mentioned above, one that enforces a security policy similar to an operating system, but with ahead-of-time code verification more like a type system. This effect is achieved by rewriting the machine instructions of code after compilation to directly enforce limits on memory access and control flow. This class of techniques is known as "software-based fault isolation" (SFI for short) or "sandboxing" [WLAG93]; it is also similar to the mechanism of inlined reference monitors for machine code [ES99]. Previous SFI techniques were applicable only to RISC architectures, or gave faulty, incomplete, or undisclosed attention to key security issues. For instance, Section 5 describes how memory protection in a previous system can be easily violated because of misplaced trust in a C compiler. (Concurrently with the research described here, Abadi et al. [ABEL05a] developed a CISC-compatible binary rewriting with some SFI-like features and a rigorous security analysis; see Section 9.3 for discussion.)

In this paper, we describe a novel technique directly applicable to CISC architectures like the Intel IA-32 (x86). We explain how using separate verification, the security properties of the rewriting depend on a minimal trusted base (on the order of a thousand lines of code), rather than on tools consisting of hundreds of thousands of lines (Section 5). We give a machine-checked proof of the soundness of our rewriting technique to provide further evidence that it is simple and trustworthy (Section 6). Finally, we discuss a prototype implementation of the technique, which is as fast as and often faster than previous unsound tools, and scales easily to large and realistically-complex applications (Sections 7 and 8). We refer to our implementation as the Prototype IA-32 Transformation Tool for Software-based Fault Isolation Enabling Load-time Determinations (of safety), or PittSFIeld[1].

Our implementation is publicly available, as are the formal model and lemmas used in the machine-checked proof, and the programs used in our experiments. They can be downloaded from `http://pag.csail.mit.edu/~smcc/projects/pittsfield/`.

2 Classic SFI

The basic task for any SFI implementation is to prevent certain potentially unsafe instructions (such as memory writes) from being executed with improper arguments (such as an effective address outside an allowed data area). The key challenges are to perform these checks efficiently, and in such a way that they cannot be bypassed by carefully chosen input code. The first approach to solve these challenges was the original SFI technique (called "sandboxing") of Wahbe, Lucco, Anderson, and Graham [WLAG93].

In order to efficiently isolate pointers to dedicated code and data regions, Wahbe et al. suggest choosing memory regions whose size is a power of two, and whose starting location is aligned to that same power. For instance, we might choose a data region starting at `0xda000000` and extending 16 megabytes to `0xdaffffff`. With such a choice, an address can be efficiently checked to point inside the region by bitwise operations. In this case, we could check whether the bitwise AND of an address and the constant `0xff000000` was equal to `0xda000000`. We'll use the term *tag* to refer to the portion of the address that's the same for every address in a region, such as `0xda` above.

The second challenge, assuring that checks cannot be bypassed, is more subtle. Naively, one might insert a checking instruction sequence directly before a potentially unsafe operation; then a sequential execution couldn't reach the dangerous operation without passing through the check. However, it isn't practical to restrict code to execute sequentially: realistic code requires jump and branch instructions, and with them comes the danger that execution will jump directly to an dangerous instruction, bypassing a check. Direct branches, ones in which the target of the branch is specified directly in the instruction, are not problematic: a tool can easily check their destinations before execution. The crux of the problem is indirect jump instructions, ones where the target address comes from a register at runtime. They are required by procedure returns, `switch` statements, function pointers, and object dispatch tables, among other language features. Indirect jumps must also be checked to see that their target address is in the allowed code region, but how can we also exclude the addresses of unsafe instructions, while allowing safe instruction addresses?

The key contribution of Wahbe et al. was to show that by directing all unsafe operations through a dedicated register, a jump to any instruction in the code region could be safe. For instance, suppose we dedicate the register `%rs` for writes to the data area introduced above. Then we maintain that throughout the code's execution, the value in `%rs` always contains a value whose high bits are `0xda`. Code can only be allowed to store an arbitrary value into `%rs` if it immediately guarantees that the stored value really is appropriate. If we know that this invariant holds whenever the code jumps, we can see that even if the code jumps directly to an instruction that stores to the address in `%rs`, all that will occur is a write to the data region, which is safe (allowed by the security policy). Of course, there's no reason why a correct program *would* jump directly to an unsafe store instruction; it is incorrect or maliciously designed programs we worry about.

Wahbe et al. implemented their technique for two RISC architectures, the MIPS and the Alpha. Because separate dedicated registers are required for the code and data regions, and because constants used in the sandboxing oper-

[1] Pittsfield, Massachusetts, population 45,793, is the seat of Berkshire county and a leading center of plastics manufacturing. Our appropriation of its name, however, was motivated only by spelling.

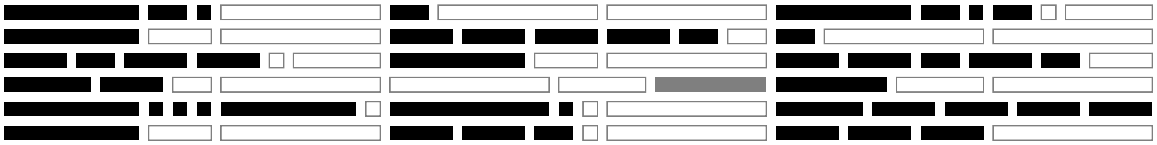

Figure 1: Illustration of the instruction alignment enforced by our technique. Black filled rectangles represent instructions of various lengths present in the original program. Gray outline rectangles represent added no-op instructions. ¡Instructions are not packed as tightly as possible into chunks because jump targets must be aligned, and because the rewriter cannot always predict the length of an instruction. Call instructions (gray filled box) go at the end of chunks, so that the addresses following them can be aligned.

ation also need to be stored in registers, a total of 5 registers are required; out of a total of 32, the performance cost of their loss was negligible. Wahbe et al. evaluated their implementation by using it to isolate faults in an extension to a database server. While fault isolation decreases the performance of the extension itself, the total effect is small, significantly less than the overhead of having the extension run in a separate process, because communication between the extension and the main server is inexpensive. As their choice of the term "fault isolation" implies, Wahbe et al. were primarily interested in isolating modules that potentially contained inadvertent errors, rather than intentionally designed attacks.

3 CISC architectures

The approach of Wahbe et al. is not immediately applicable to CISC architectures like the Intel IA-32 (i386 or "x86"), which feature variable-length instructions. (The IA-32's smaller number of registers also makes dedicating several registers undesirable, though its 32-bit immediates mean that only 2 would be needed.) Implicit in the previous discussion of Wahbe et al.'s technique was that jumps were restricted to a single stream of instructions (each 4-byte aligned, in a typical RISC architecture). By contrast, the x86 has variable-length instructions that might start at any byte. Typically code has a single stream of intended instructions, each following directly after the last, but by starting at a byte in the middle of an intended instruction, the processor can read an alternate stream of instructions, generally nonsensical. If code were allowed to jump to any byte offset, the SFI implementation would need to check the safety of all of these alternate instruction streams; but this would be infeasible. The identity of the hidden instructions is a seemingly random function of the precise encodings of the intended ones (including for instance the eventual absolute addresses of forward jump targets), and most modifications to hidden instructions would garble the real ones.

To avoid this problem, our PittSFIeld tool artificially enforces its own alignment constraints on the x86 architecture. Conceptually, we divide memory into segments we call *chunks* whose size and location is a power of two, say 16, bytes. PittSFIeld inserts no-op instructions as padding so that no instruction crosses a chunk boundary; every 16-byte aligned address holds a valid instruction. Instructions that are targets of jumps are put at the beginning of chunks; `call` instructions go at the ends of chunks, because the instructions after them are the targets of returns. This alignment is illustrated schematically in Figure 1. Furthermore, jump instructions are checked so that their target addresses always have their low 4 bits zero. This transformation means that each chunk is an atomic unit of execution with respect to incoming jumps: it is impossible to execute the second instruction of a chunk without executing the first. Thus, PittSFIeld needs no dedicated registers: it simply puts an otherwise unsafe operation and the check of its operand in the same chunk. (In general, one scratch register is still required to hold the effective address while it is being checked, but it isn't necessary for the same register to be used consistently, or for other uses of the register to be prohibited.)

4 Optimizations

The basic technique described in Section 3 ensures the memory and control-flow safety properties we desire, but as described it imposes a large performance penalty. This section describes five optimizations that reduce the overhead of the rewriting process, at the expense of making it somewhat more complex. The first three optimizations were described by Wahbe et al., and are well known; the last two have, as far as we know, not previously been applied to SFI implementations.

4.1 Special registers

One obvious way to reduce the overhead of sandboxing checks is to avoid applying them repeatedly to the same value. For instance, the register `%ebp` (the 'frame pointer' or 'base pointer') is often used to access local variables stored on the stack, part of the data region. This motivates treating `%ebp` differently from other general purpose registers: rather than allowing `%ebp` to contain any value, and checking each time the code uses it, we can instead arrange that it always be a valid pointer to the data region.

With this approach, it is changes to `%ebp`, rather than uses of it, that need to be checked; since it is usually set once at the beginning of a function and then never modified, this reduces the total amount of checking. In fact, it isn't necessary to check `%ebp` immediately after it is modified, but it must be checked before it is used, and before a jump, because the instructions at the jump target would expect it to be valid. This policy about `%ebp` could be described as treating it as 'usually-sandboxed', rather than 'usually-unsandboxed'. Note that because of the relatively unrestricted possibilities for jumps, such a decision has to be made globally for the entire code region.

4.2 Guard regions

The technique described in the previous subsection for optimizing the use of `%ebp` would be effective if `%ebp` were only dereferenced directly, but in fact `%ebp` is often used with a small constant offset to access the variables in a function's stack frame. Usually, if `%ebp` is in the data region, then so is `%ebp + 10`, but this would not be the case if `%ebp` were already near the end of the data region. To handle this case efficiently, we follow Wahbe et al. in using *guard regions*, areas in the address space directly before and after the data region that are also safe for the sandboxed code to attempt to write to. An access at a small offset from a sandboxed data address will be sure to fall either in the data region or in one of the guard regions, and thus be safe.

If we further assume that accesses to the guard region can be efficiently trapped (such as by leaving them unmapped in the page table), we can optimize the use of the stack pointer `%esp` in a similar way. The stack pointer is similar to `%ebp` in that it generally points to the stack and is accessed at small offsets, but unlike the frame pointer, it is frequently modified; in particular, it is frequently incremented and decremented as items are pushed onto and popped off the stack. Even if each individual change is small, each must be checked to make sure that it isn't the change that pushes `%esp` past the end of the allowable region. However, if attempts to access the guard regions are trapped, every use of `%esp` can also serve as a check of the new value. One important point is that we must be careful of modifications of `%esp` that do not also use it; this danger will be illustrated in Section 5.

4.3 Ensure, don't check

A final optimization that was included in the work of Wahbe et al. has to do with the basic philosophy of the safety policy that the rewriting enforces. The most important aspect of the policy is that the untrusted code should not be able to perform any action that is unsafe. We could also ask, what should happen when the untrusted code attempts an unsafe action? For instance, one possibility would be to terminate the untrusted code with an error report. Another possibility, however, would be to simply require that when an unsafe action is attempted, some action consistent with the security policy occurs instead. For example, instead of a jump to a forbidden area causing an exception, it might instead cause a jump to some arbitrary other location in the code region. To follow this policy, it isn't necessary to check whether an address is legal, and branch to an error handler if not; the code can simply set the bits of the address appropriately and use it. If the address was originally illegal, it will 'wrap around' to some legal, though likely not meaningful, location.

At first blush, this approach of substituting seemingly arbitrary values might seem reckless, and there are certainly applications (e.g., debugging) where it would be unhelpful. However, it is reasonable to optimize a security mechanism for the convenience of legitimate code, rather than of illegal code. Attempted jumps to an illegal address should not be expected to occur frequently in practice: it is the responsibility of the code producer (and her compiler), not the code user, to avoid them. The performance effects of this tradeoff are shown in Section 8.

4.4 One-instruction address operations

For an arbitrarily chosen code or data region, the sandboxing instruction must check (or, according to the optimization of Section 4.3, ensure) that certain bits of an address are set, and others are clear. This requires two instructions: an AND instruction and a comparison for a check, or an AND instruction and an OR instruction to modify the appropriate bits. By further restricting the locations of the sandbox regions, however, the number of instructions can be reduced to one. We choose the code and data regions so that their tags have only a single bit set, and then reserve from use the region of the same size starting at address 0, which we call the *zero-tag region* (because it corresponds to a tag of 0). With this change, bits in the address only need to be clear (or cleared) and not also set.

PittSFIeld uses a code region starting at 0x10000000 and a data region starting at 0x20000000. The code sequences to verify an address in %ebx for the data region are then as follows:

- If we are checking addresses:[2]:

  ```
      test   $0xdf000000, %ebx
      jz     ok
      int3
  ok:
  ```

  The test instruction checks if the tag is 0x20 by AND-ing %ebx with a mask made of the complement of the tag; if the result is zero, control continues at ok, otherwise it falls through to int3, a one-byte instruction that causes a trap.

- If we are modifying addresses:

  ```
      and    $0x20ffffff, %ebx
  ```

  This instruction turns off all of the bits in the tag except possibly the third from the top, so the address will be either in the data region or the zero-tag region.

We chose both sequences to minimize the number of instruction bytes required, 9 for the check sequence and 6 for the direct modification. Taking into account the other instructions that must fit in a single chunk, direct modification can be used with 16-byte chunks, while checking requires 32-byte chunks.

On large examples like those in Section 8.2, disabling this optimization increases PittSFIeld's overhead by about 10% (e.g., from 50% to 55%). 16-byte chunks use less space overall, and our original intuition had been that this would also improve performance by having fewer no-op instructions and better cache density. It turns out however that some programs run faster with 32-byte chunks, perhaps because of reduced fragmentation in inner loops.

4.5 Efficient returns

A final optimization helps PittSFIeld take advantage of the predictive features of modern processors. Indirect jumps are potentially expensive for processors if their targets cannot be accurately predicted. For general indirect jumps, processors typically keep a cache, called a 'branch target buffer', of the most recent target for a jump instruction. A particularly common kind of indirect jump is a procedure return, which on the x86 reads a return address from the stack. A naive implementation would treat a return as a pop followed by a standard indirect jump; for instance, an early version of PittSFIeld translated a ret instruction into:

```
    popl   %ebx
    and    $0x10fffff0, %ebx
    jmp    *%ebx
```

However, if a procedure is called from multiple locations, the single buffer slot will not be effective at predicting the return address, and performance will suffer. In order to deal more efficiently with returns, modern x86 processors keep a shadow stack of return addresses in a separate cache, and use this to predict the destinations of returns. To allow the processor to use this cache, we would like PittSFIeld to return from procedures using a real ret instruction. Thus PittSFIeld modifies the return address and writes it back to the stack before using a regular ret. In fact, this can be done without a scratch register:

```
    and    $0x10fffff0, (%esp)
    ret
```

On a worst case example, like the recursive Fibonacci function mentioned in Section 8.1, this optimization makes an enormous difference, reducing 95% overhead to 40%. In realistic examples, the difference is around 5% of the total overhead.

5 Trust

In order for a rewriting technique like ours to enhance the security of a system, careful consideration must be given to the system architecture and the trust relationships between the production, checking, and execution of code. Specifically, we advocate an arrangement in which the compilation and the rewriting of the code are performed by the untrusted code producer, and the safety policy is enforced by a separate verification tool. This architecture is familiar to users of Java: the code producer writes source code and compiles it to Java byte code using the compiler of her choice, but before the code user executes an applet he checks it using a separate byte code verifier. (One difference from Java is that once checked, our code is executed more or less directly; there is no trusted interpreter as complex as a Java just-in-time compiler.) The importance of having a small, trusted verifier is also stressed in work on proof-carrying code [NL96].

[2] Assembly language examples use the GAS, or 'AT&T', syntax standard on Unix-like x86-based systems, which puts the destination last.

Except for the original work of Wahbe et al., previously described SFI implementations have neglected this verification aspect, instead requiring that the rewriting tool, or both the rewriting tool and the compiler, be trusted. There are two serious problems with a verifier-less approach. First, if the code production process is to be trusted, the code must either be regenerated right before use, or some other trust mechanism must exist between the producer and the user. Second, compilers and rewriters may be large and complex enough that they cannot be relied upon to maintain the desired security.

At first glance, one might hope to perform an instruction transformation like SFI by simply rewriting one compiled binary into another. However, performing such a transformation is more difficult than might at first appear, because of the need to update references to instructions that move as part of the transformation. A compiled binary lacks information distinguishing machine words representing code addresses or offsets that must be updated from similarly-valued integers that should not be modified. Because of this difficulty, binary security tools usually perform their rewriting either earlier in the code life cycle, using information otherwise discarded after compilation, or later, transforming code dynamically, one basic block at a time, as it is loaded. While dynamic transformation techniques to enforce targeted security policies have been demonstrated in some recent research (see Section 9 for a discussion), they can be quite complex, especially if high performance is desired. Previous SFI implementations either integrate the rewriting tool with the compiler proper, or perform the rewriting between the compiler and the assembler.

Rewriting code prior to assembly eliminates the difficulty of relocating code addresses: they still exist as symbolic labels, so retain their correct references. However, unless code is to be distributed in source form and compiled (or assembled) before each use, some connection must be made between the rewriting and the decision to execute the code: either a trust relationship, or a verification. Previous work [SS97] proposes that the rewriting tool cryptographically sign the transformed code, and the code user verify the signature. In our opinion, a signature-based approach has two main disadvantages: first, a public-key infrastructure of some sort is required to check the signer's identity, a large technical addition; and second, a trusted third party is needed to compile the code, a significant additional existence assumption. On architectural grounds alone, a verification-based arrangement seems preferable.

An even more serious problem with a design that trusts the compiler and rewriting tool to perform correctly is that those tools may not really be worthy of our trust. For instance, several previous x86 SFI tools that operate at the level of assembly language code trust that their input code correctly uses the stack and that frame pointers to refer only to valid stack frames. While the compilers used with these tools (usually the GNU C Compiler GCC or its C++ variant G++) are generally quite reliable in correctly generating stack references, the specification of correct stack usage they conform to is not a good match for the security needs of an SFI tool.

As a concrete example, the following C source code demonstrates a problem of this sort in the MiSFIT tool [SS97]:

```
jmp_buf env;
void f(int arg,  int arg2,
       int arg3, int arg4) {
    return;
}
void poke(int *loc, int val) {
    int local;
    unsigned diff = &local - loc - 4;
    for (diff /= 4; diff; diff--)
        alloca(16);
    f(val, val, val, val);
}
```

The function poke has the effect of storing an arbitrary word at an arbitrary address (it also overwrites some adjacent words), clearly something that should be restricted by MiSFIT's security policy. However, the policy is circumvented by the above code, which performs the dangerous write as a push to the stack, in the course of calling the function f. To get the stack pointer to point at the desired location, the code repeatedly advances it using the alloca function, by an amount equal to the difference between its current location (approximated by the address of the local variable local) and the desired target. While each individual increment of the stack pointer is modest (16 bytes), because they occur in a loop the total effect is large. (Though we were unable to obtain the implementation to test, the description in [ES99] suggests that the tool there would be vulnerable to the same attack.) By contrast, PittSFIeld would recheck the stack pointer each time around the loop containing alloca, never allowing it to escape the data region.

Compilers are inevitably large and complex programs, so it makes sense to avoid relying on the correctness of a compiler, or of a rewriting performed as part of a compiler, for system security. What about trusting a small, dedicated rewriting tool? While this approach would be better, it still leaves the rewriting tool with several roles that are in tension: preserving the behavior of the original program, enforcing a security policy, and optimizing the transformation to reduce overhead. For instance, any optimization that changed how the rewriting was performed would carry the danger of accidentally opening a security hole. To us, it seems better to assign the task of checking adherence to a security policy to a separate tool, so that the security properties of the technique can be understood in isolation. Once the separate verification tool can be trusted, the compiler and rewriting tool can be modified with significantly greater flexibility.

6 Formal Analysis

Having restricted ourselves, as argued in Section 5, to a separate, minimal verification tool as the guarantor of our technique's safety, we can devote more effort to analyzing and assuring ourselves of that component's soundness. Specifically, we have constructed a completely formal and machine-checked proof of the fact that our technique ensures the security policy that it claims to. Though the security of a complete system of course depends on many factors, such a proof provides a concise and trustworthy summary of the key underlying principles. Formal theorem proving has a reputation for being arduous; we think the relative ease with which this proof was completed is primarily a testament to the simplicity of the technique to which it pertains.

To better understand how the verification works, it is helpful to borrow concepts from program analysis, and think of it as a conservative static analysis. We are interested in a particular property of the program's execution, roughly that it never jumps outside its code region or writes outside its data region. In general, this property is impossible to decide, but it is tractable if we are willing to accept one-sided error: we do not mind if we fail to recognize that a program has the safety property, as long as whenever we analyze that it does, we are correct. If your original program was correct, it already had this safety property; you can think of the rewriting as simply making the property manifest, so that the verifier can easily check it.

The verification process essentially computes, for each position in the rewritten instruction stream, a conservative property describing the contents of the processor's registers at any time when execution might reach that point. For instance, directly after an appropriate `and` instruction not at a chunk boundary, we might know that the contents of the target register are appropriately sandboxed for use in accessing the data region. The major part of the safety proof is to show that these properties are sound for any possible execution; it's then easy to see that if the properties always hold, no unsafe executions will be possible. An important aspect of the soundness is that it is inductive over the steps in the execution of the rewritten code: for instance, it is important that none of the instructions in the code region change during execution (new instructions would not necessarily match the static properties), but we can be confident of this only because in previous execution up to a given point, we can assume we were successful in preventing writes outside the data section. In program verification terminology, the soundness property is an invariant that we check as being preserved by each instruction step.

```
(defun seq-reachable-rec (mem eip k)
  (if (zp k) (if (= eip (code-start)) 0 nil)
    (let ((kth-insn
           (kth-insn-from mem (code-start) k)))
    (or (and kth-insn (= eip kth-insn) k)
        (seq-reachable-rec mem eip (- k 1))))))

(defthm if-seq-reach-in-k-then-bound-by-kth-insn
 (implies (and (mem-p mem) (natp k) (natp eip)
               (kth-insn-from mem (code-start) k)
               (seq-reachable-rec mem eip k))
          (<= eip (kth-insn-from mem
                          (code-start) k))))
```

Figure 2: Example of a typical function definition (above) and lemma (below) from our formal ACL2 proof. `seq-reachable-rec` is a recursive procedure that checks whether the instruction at location `eip` is among the first k instructions reachable from the beginning of the sandboxed code region in a memory image `mem`. The lemma states that if `eip` is among the first k instructions, then its address is at most that of the kth instruction.

We have constructed the proof using ACL2 [KM97]. ACL2 is a theorem-proving tool that combines a restricted subset of Common Lisp, used for modelling a system, with a sophisticated engine for semi-automatically proving theorems about those models. We use the programming language (which is first-order and purely functional) to construct a simplified model of our verifier, and a simulator for the x86 instruction set. Then, we give a series of lemmas about the behavior of the model, culminating in the statement of the desired safety theorem. The lemmas are chosen to be sufficiently elementary that ACL2 can automatically prove each from the model and the preceding lemmas, sometimes with the help of 'hints' about which proof strategies to use. We chose ACL2 for the proof with the hope that its concrete modelling language and familiar underlying logic (essentially quantifier-less first-order logic) would reduce the learning curve associated with a new tool. In this, we feel fairly successful: the proof has taken less than two months of effort by a user with no previous experience with proof assistants (presumably an experienced ACL2 user could have produced a more elegant proof in less time.) An example of a function from the executable model and a lemma we have proved about it are shown in Figure 2. Though ACL2's proofs cannot be automatically checked by any tool other than ACL2 itself, potentially a disadvantage for confidence in its results, it is a sufficiently well-established tool that we are not seriously troubled.

One aspect of the proof to note is that it deals with a subset of the instructions handled by the real tool: this applies both to which instructions are allowed by the simulated verifier, and to which can be executed by the x86 simulator. The instructions were chosen to exercise all of the aspects of the security policy; for instance, `jmp *%ebx` is included to demonstrate an indirect jump. However, the simulator is structured so that an attempt to execute any un-modelled instruction causes an immediate failure, so safety for a program written in the subset that is treated in the proof should extend to the complete system.

A related concern is whether the simulated x86 semantics match those of a real processor. We have proved that the technique (as formalized) is sound with respect to our model of the behavior of an x86 processor, but if our model of that behavior has an error, it might mask a problem with the technique. While the current simulator was written by hand, we have explored ways in which the processor description could be generated automatically from an independent declarative specification. Given such a trustworthy description of the complete processor, we can show a simulation result proving that our model in the proof is a correct subset of the real processor's behaviors. Our goal is similar to the machine semantics constructed for foundational proof-carrying code by Michael and Appel [MA00], but differs somewhat because of choice of a proof environment: compared to the higher-order logical framework they used, ACL2 has fewer abstraction mechanisms, but better primitive support for arithmetic. Similar machine architecture specifications in ACL2-like systems have been constructed by hand [BY96].

7 Prototype implementation

To test the practicality of our approach, we have constructed a prototype implementation, named PittSFIeld. PittSFIeld instantiates a simple version of the technique, incorporating only the most important optimizations, and is not as robust as a polished tool might be. However, PittSFIeld was designed to address some important practical considerations for a real tool, such as the separate verification model and scalability to large and complex programs. In particular, PittSFIeld makes no fundamental compromises with respect to the rigorous security guarantees that the technique offers. The performance of code rewritten by PittSFIeld (described in the next section) should also give a reasonable upper bound on the overhead of this general approach, one which could be somewhat improved by further optimization. (However, other aspects of the prototype are not representative of a practical implementation: for instance, the rewriting and verification processes themselves are unrealistically slow.)

The rewriting performed by PittSFIeld is a version of the techniques described in Sections 3 and 4, chosen to be easy to perform. The register `%ebx` is reserved (using the `--fixed-ebx` flag to GCC), and used to hold the sandboxed address for accesses to both the data and code regions. The effective address of an unsafe operation is computed in `%ebx` using a `lea` instruction. The value in `%ebx` is required to be checked or sandboxed directly before each data write or indirect code jump (reads are unrestricted). Both direct and indirect jumps are constrained to chunk-aligned targets. Guard regions are 64k bytes in size: `%ebp` and `%esp` are treated as usually-sandboxed. Accesses are allowed at an offset of up to 64k from `%ebp`, and of up to 255 bytes from `%esp`; `%esp` is also allowed to be modified up to 255 times, by as much as 255 bytes each time, between checks/modifications. Both `%ebp` and `%esp` must be restored to safe values before a jump. A safe value in `%esp` may be copied to `%ebp` or vice-versa without a check. Chunks are padded using standard no-op instructions of lengths 1, 2, 3, 4, 6, and 7 bytes, to a size of 16 or 32 bytes.

Both the rewriting and the verification in PittSFIeld are performed as single top-to-bottom passes, essentially as finite-state machines. While this prohibits some optimizations (for instance, labels that are targets only of direct jumps need not necessarily be aligned), it allows PittSFIeld to rewrite very large programs, and guarantees that its running time will be linear. (A verification technique with bad worst-case performance can allow a denial-of-service attack [GPF03].)

The rewriting phase of PittSFIeld is implemented as a text processing tool operating on input to the GNU assembler `gas`. In most cases, alignment is achieved using the `.p2align` directive to the assembler, which computes the correct number of no-ops to add; the rewriter uses a conservative estimate of instruction length to decide when to emit a `.p2align`. The rewriter adds no-ops itself for aligning call instructions, because they need to go at the end rather than the beginning of a chunk. The rewriter

```
f:  push    %ebp
    mov     %esp, %ebp
    mov     8(%ebp), %edx
    mov     48(%edx), %eax
    lea     1(%eax), %ecx
    mov     %ecx, 48(%edx)
    pop     %ebp
    ret
```

```
f:  push    %ebp
    mov     %esp, %ebp
    mov     8(%ebp), %edx
    mov     48(%edx), %eax
    lea     1(%eax), %ecx
    lea     0(%esi), %esi
--------------------------------
    lea     48(%edx), %ebx
    lea     0(%esi), %esi
    lea     0(%edi), %edi
--------------------------------
    and     $0x20ffffff, %ebx
    mov     %ecx, (%ebx)
    pop     %ebp
    lea     0(%esi), %esi
--------------------------------
    and     $0x20ffffff, %ebp
    andl    $0x10fffff0, (%esp)
    ret
```

Figure 3: Before and after example of code transformation. `f` is a function that takes an array of integers, increments the 12th, and returns (in `%eax`) the value before the increment. The assembly code on the left is produced by GCC; that on the right shows the results of the PittSFIeld rewriter after assembly. Rules separate the chunks, and no-op instructions are printed in gray. (Though they look the same here, the first three no-ops use different instruction encodings so as to take 4, 6, and 7 bytes respectively).

notices instructions that are likely to be used for their effect on the processor status flags (e.g., comparisons), and saves and restores the flags register around sandboxing operations when the flags are deemed live. An example of the rewriter's operation on a small function is shown in Figure 3.

The verification phase is also implemented via text-processing, as a filter that parses the output of the disassembler from the GNU "binutils" package (the program named `objdump`). A more careful implementation would probably examine the binary values of opcodes directly, to avoid trusting the disassembler (which is much larger than the prototype verifier). Because it uses a single disassembly pass, the verifier enforces alignment by checking that an instruction in the single stream must appear at each chunk starting address. The verifier currently verifies only the style of rewriting in which pointers are modified, and not the style in which they are checked and execution halted if they are incorrect; but supporting the checking style would not be particularly difficult. As mentioned above, the verifier is essentially finite-state: at each code location, it keeps track of variations from the standard safety invariant, checking them and then updating its knowledge for each instruction. Operations that 'strengthen' the invariant (for instance, sandboxing a pointer value in `%ebx`) expire after one instruction or at a chunk boundary, whichever comes first. Operations that 'weaken' the invariant (for instance, loading a new value into `%ebp`) persist until corrected, and must not reach a jump.

PittSFIeld supports a large subset of the x86 32-bit protected mode instruction set, including most commonly used integer instructions, and a large number of floating point instructions. Supervisor mode instructions, string instructions, and multimedia SIMD (e.g. MMX) instructions are not supported, as we had no use for them; the verifier will reject any program containing an instruction it does not recognize. The rewriter and the verifier are both implemented as Perl scripts, using a common library of regular expressions for parsing instructions. The common library, the rewriter, and the verifier represent approximately 150, 525, and 450 lines of code respectively, including blank lines and comments.

8 Performance results

To asses the time and space overheads imposed by our technique, we used our PittSFIeld tool to run a variety of stand-alone applications in fault-isolated environments. The programs were not chosen as code one might particularly want to run from an untrusted source, merely as computation-intensive benchmarks. The 'untrusted' code in each case consisted of the application itself, and some simple standard library routines. More complex library routines and system calls were treated as 'trusted,' and accessed via special stubs allowing controlled access out of the sandbox. In a realistic application, these stubs would include checks of their arguments to enforce desired security policies. In our prototype, the trusted loading application and stub trusted calls consisted of approximately 550 lines of C code, including blank lines and comments.

8.1 Microbenchmarks

To understand the performance overhead introduced by the use of PittSFIeld, we modified several simple C and C++ programs with the tool, and compared their performance with and without modification, giving the results shown in Figure 4. When possible, we also translated them into a type-safe language, to compare the performance of that approach. `factor`, based on the program with the same name from the GNU coreutils distribution, factors an 18-digit number that is the product of two large primes by a brute force method. For both SFI tools, we treated the internal `__udivdi3` routine, which GCC uses to divide 64-bit integers, as trusted. `fib` computes the 42nd Fibonacci number using the standard exponential-time recursive algorithm, and mainly tests the overhead incurred by functions calls. `md5` computes the MD5 checksum of a one and a half gigabyte string by reading a 15 megabyte buffer 100 times over, using essentially the reference implementation from RFC 1321 [Riv92]. `switch`, `fp`, and `virtual` mimic three styles of inner operation loops for an interpreted programming language, using a switch statement, function pointers, and C++ virtual method calls respectively; they stress the performance of indirect jumps. We wrote Java versions of the benchmarks trying to match the C and C++ versions as closely as possible; for `md5` we used the fastest freely implementation of the hash function we could find (by Timothy Macinta); versions that looked more like the C reference implementation were strangely much slower. Java does not have function pointers, so there is no Java implementation of `fp`. Because of the inherent difficulty of identifying 'the same program' in different languages, all the cross-language comparisons should be taken with a grain of salt.

Each program was compiled in an unmodified version, and with the SFI tools PittSFIeld and MiSFIT. To explore in more detail the sources of overhead for PittSFIeld, we also applied two of the transformations required by PittSFIeld on the programs without other changes: reserving the `%ebx` register so that it is not used in the compiled code, and inserting padding to prevent instructions from crossing 16-byte boundaries. The C and C++ programs were all compiled with GCC and G++ version 3.3.5, at optimization level `-O3`. We also tested Java versions of the programs in two ways: GCJ is an ahead-of-time compiler for Java that uses the same back-end as GCC; we again used version 3.3.5 at optimization level `-O3`. In addition we tested the just-in-time-compiler-based virtual machine supplied by Sun, version 1.5.0 of the HotSpot Client VM. The hardware used in the tests was an AMD Athlon running at 1066MHz, with 256KB of cache and 1GB of main memory. The otherwise unloaded machine uses Linux 2.4.27 and actually has two processors, but all of the test programs were single threaded. (Because our technique depends only on values in registers and a (thread-private) stack, it would be equally applicable to multithreaded programs.)

We used version 0.2 of MiSFIT, with the 'write protection', 'call protection', 'stack protection', and 'optimization' options enabled, though in our examples the stack protection appeared to have no effect. Because as distributed, MiSFIT is designed to be used in a larger system, we had to recreate some parts of its supporting infrastructure by hand. We wrote our own assembly-language implementation of the trusted return address stack, interfacing to hash table code supplied in the MiSFIT distribution. We did not actually bother to rearrange data areas used by the MiSFIT-sandboxed programs to use any particular area of memory, so we used a region tag of 0 and a region mask of all ones so that sandboxing never caused pointers to be modified. Two differences should be noted between the protection supplied by MiSFIT and PittSFIeld in these examples: On one hand, MiSFIT protects return addresses from being subverted, while PittSFIeld does not. On the other hand, MiSFIT's trust in GCC makes its protection of the stack incomplete, as mentioned earlier, and it also trusts the jump table that GCC generates for the switch statement in `switch`, including that the index into it will always be in bounds. (By contrast, PittSFIeld places the jump table in the data segment, and sandboxes the addresses read from the table as for any indirect jump.)

The unusual specialization of some of these benchmarks leads to some out-of-scale effects on performance. For instance, `fib`, `fp`, and `virtual` consist of little but procedure calls, and so demonstrate an inefficiency in how PittSFIeld handles them; for 32-byte chunks, the fact that the rewriter does not have complete knowledge of alignment means that much more padding than necessary is added. The high performance of the Java JIT on `fib` and `switch` suggests that its dynamic optimizer can find unusually productive opportunities for improvement there (perhaps inlining recursive calls to `fib`, say).

Overall, these results show that PittSFIeld and MisFIT's performance are in the same general range, with MiSFIT showing more variation. PittSFIeld's alignment-based sandboxing technique is more efficient that MiSFIT's table-based one for programs with many procedure

	factor	fib	md5	switch	fp	virtual
unmodified time = 1.0	7.03 s	11.17 s	16.81 s	10.68 s	11.35 s	12.91s
%ebx reserved	1.04	1.01	1.02	1.00	0.99	0.99
padding	1.05	1.16	1.10	0.99	0.98	0.99
%ebx reserved and padding	1.13	1.17	1.13	0.99	0.97	0.99
PittSFIeld (ensure)	1.29	1.40	1.25	1.07	1.16	1.14
PittSFIeld (check)	1.38	2.94	1.39	1.09	2.16	1.47
MiSFIT	1.15	2.20	1.50	1.02	1.59	1.64
Java (gcj)	1.10	1.01	1.45	0.90	N/A	1.06
Java (JIT)	2.25	0.81	1.76	0.52	N/A	1.15

Figure 4: Runtime measurements comparing sandboxed programs to unmodified programs and programs protected via other techniques. The first row gives times for unmodified programs in seconds; subsequent rows are slowdown factors (1.0 means 'same as unmodified') compared to that time.

calls, especially indirect ones. On the other hand, MiSFIT's technique imposes less overhead on programs that make relatively few writes or jumps, such as factor. In general, SFI is competitive in performance with type-safety based languages such as Java, while allowing existing C code to be used with little or no modification.

8.2 Larger benchmarks

To test the scalability of the PittSFIeld approach, we also used it to isolate faults in some full-size applications that are standard in the open source community. These results are shown in Figure 5. bzip2 is a general-purpose lossless compression tool; in the benchmark, it compresses a 21 megabyte file containing C source code. oggenc is an audio compression tool for a format similar to MP3; in the benchmark, it compresses a 44 megabyte file containing CD-quality spoken-word audio. We chose oggenc as a program that makes heavy use of floating-point arithmetic. bc is the GNU implementation of an arbitrary-precision desk calculator; in the benchmark, it computes 150,000 products of 80-digit integers. gcc[3] is a development version of the GNU C compiler; in the benchmark, it compiles itself as a single compilation unit. gcc and its subsidiary libraries (including a C preprocessor and a custom garbage collector) consist of approximately 750,000 lines of code. Because of gcc's size, we used a different memory layout in which the sandbox data region is 1GB. A previous version of our tool required that its input be a single compilation unit, so we modified bzip2, oggenc, and gcc to compile as single .c files; bc was not modified in this way. The other changes needed to the programs, to remove uses of library calls not included in our stubs when they were incidental to the program's functioning, were minor.

Two rows of Figure 5 show how PittSFIeld's transformation affects the size of the code. The row "PittSFIeld size ratio" shows the ratio of the size of an object file processed by PittSFIeld to that of an unmodified program, ranging up to 100% overhead; the increase includes the addition of both sandboxing instructions and padding no-ops. Besides using additional memory, expanding code tends to decrease instruction cache locality; this accounts for some of the slow-down measured in the "padding" row. (The size ratio for oggenc is smaller than the others because it contains a large amount of static data, which is unchanged by the code rewriting.) The row "compressed size ratio" is analogous, except that both the transformed and original object files were first compressed with bzip2: these overheads are smaller, at most 25%. Compressed size is relevant, for instance, to the cost of distributing software; the compressed ratios are smaller because the added instructions tend to be repetitive.

The runtime overheads of 30-45% (somewhat higher for checking) shown for these computationally intensive examples are not insignificant, but we believe they would often be acceptable for one component of a larger system, given the security and usability benefits isolation provides. Reserving a scratch register has relatively little effect on performance; adding no-ops for padding is somewhat more detrimental. Together they account for a little less than a third of the technique's overhead, on average. These examples also show that the technique is compatible with all of the complexities of realistic programs in unsafe languages, including complex explicit memory management, without requiring programmers to modify or annotate working code. The security benefits described

[3] To be precise, the executable tested is actually what would be installed as cc1, representing the compiler pass that transforms C code into assembly language.

	bzip2	bc	oggenc	gcc
lines of code	6753	10212	58372	752986
PittSFIeld size ratio	1.96	1.81	1.14	1.84
compressed size ratio	1.24	1.19	1.08	1.10
unmodified time = 1.0	21.68 s	12.41 s	35.22 s	167.75 s
%ebx reserved	1.05	0.99	1.01	1.01
padding	1.03	1.18	1.05	1.09
%ebx reserved and padding	1.09	1.18	1.08	1.11
PittSFIeld (ensure)	1.28	1.45	1.41	1.30
PittSFIeld (check)	1.34	1.60	1.52	1.63

Figure 5: Code size and runtime measurements comparing sandboxed programs to unmodified programs. Lines of code include blank lines and comments.

in previous sections can be obtained for existing C and C++ components at an acceptable cost in performance and developer effort.

9 Related work

This section compares our work with previous implementations of SFI, and with other techniques that ensure memory safety or isolation including code rewriting, dynamic translation, and low-level type systems. It also distinguishes the *isolation* provided by SFI from the *subversion protection* that some superficially similar techniques provide.

9.1 Other SFI implementations

Binary sandboxing was introduced as a fault-isolation technique by Wahbe, Lucco, Anderson, and Graham [WLAG93]. The basic features of their approach were described in Sections 2 and 4. Wahbe et al. mention in a footnote that their technique would not be applicable to architectures like the x86 without some other technique to restrict control flow, but then drop the topic.

Subsequent researchers generally implemented a restriction on control flow by collecting an explicit list of legal jump targets. The best example of such a system is Small and Seltzer's MiSFIT [SS97], an assembly-language rewriter designed to isolate faults in C++ code for an extensible operating system. MiSFIT generates a hash table from the set of legal jump targets in a program, and redirects calls and indirect jumps through code that checks that the target appears in the table. Function return addresses are also stored on a separate, protected stack. Because control flow is prevented from jumping into the middle of them, the instruction sequences to sandbox memory addresses don't require a dedicated register, though MiSFIT does need to spill to the stack to obtain a scratch register in some cases. A less satisfying aspect of MiSFIT is its trust model. The rewriting engine and the code consumer must share a secret, which the rewriter uses to sign the generated code, and MiSFIT relies on the compiler to correctly manage the stack and to produce only safe references to call frames. We described the security difficulties with this approach in Section 5.

Erlingsson and Schneider's SASI tool for the x86 [ES99] inserts code sequences very similar to MiSFIT's, except that its additions are pure checks that abort execution if an illegal operation is attempted, and otherwise fall through to the original code, like PittSFIeld's 'check' mode. In particular, the SASI tool is similar to MiSFIT in its use of a table of legal jump targets, and its decision to trust the compiler's manipulation of the stack. Lu's C+J system [Lu00b, Lu00a] also generates a table of legal jump destinations (separately for calls and returns, in his case), but the indices into the table are assigned sequentially at translation time, rather than being the addresses themselves as in other systems, so there's no danger of collision. For procedure calls, the index is stored right before the first instruction; for returns, the index is stored on the stack in place of the return address.

Silver's SFI implementation [Sil96] follows the ideas of Wahbe et al. quite closely, except that no verification was implemented; only RISC architectures are targeted. The Omniware virtual machine [ATLLW96], on which Wahbe and Lucco worked after the classic paper, uses SFI in translating from a generic RISC-like virtual machine to a variety of architectures, including the x86. The Omniware VM implemented extensive compiler-like optimizations to reduce the overhead of sandboxing checks, achieving average overheads of about 10% on selected SPEC92 benchmarks. However, the focus of the work appears to have been more on performance and portability than on security; available information on the details of the safety checks, especially for the x86, is sparse. In a patent [WL98] (assigned to Microsoft, which purchased Omniware maker Colusa Software in March of 1996) Wahbe and Lucco disclose that later versions of the system enforced more complex, page-table like memory permissions, but give no more details of the x86 implementation.

As far as we know, our work described in Section 6 is the first machine-checked or completely formalized soundness proof for an SFI technique or implementation. Necula and Lee [NL96] proved the soundness of SFI as applied to particular programs, but not in general, and only in the context of simple packet filters. Abadi et al. ([ABEL05b], see Section 9.3 for discussion) give a human-readable prose proof for the safety of a model of an SFI-like system. Key to making a reasonably-sized soundness proof, and to confidence in a technique's security more generally, is an architecture based on separate verification; this is missing from all the implementations described in the preceding paragraphs except for Wahbe et al.'s original one.

9.2 Isolation and preventing subversion

In comparing the security provided by an SFI tool like PittSFIeld to other code-rewriting techniques, there is a fundamental distinction in what kind of security property a tool provides. In general, a security failure of a system occurs when an attacker chooses input that causes code to perform differently than its author intended, and the subverted code then uses privileges it has to perform an undesirable action. Such an attack can be prevented either by preventing the code's execution for being subverted, or by isolating the vulnerable code so that even if subverted, it can still be prevented from taking an undesirable action. Many security techniques are based on the prevention of subversion: for instance, ensuring that procedure calls always return to their call sites, even if the stack has been modified by a buffer overrun. SFI, by contrast, is fundamentally a technique for isolating one part of a program from another. To function as a security technique, this isolation must be used to support a design that divides a system into more and less trusted components, and restricts the interactions between the two. (At the very least, because it isn't possible to make system calls from inside a sandbox, SFI requires the definition of some interface constraining code's interaction with the outside world.) Whether SFI or another isolation mechanism is used, designing a system to separate and restrict the interface of privileged operations is generally a good choice to improve system security. However, it requires some additional developer effort in defining appropriate interfaces, which may not exist in a monolithic system. By contrast, a system aimed just at preventing subversion can be used on an unmodified monolithic program, on the assumption that, say, executing user-provided code is always undesirable.

For instance, we can imagine using SFI as an alternative to system-level mechanisms for separating programs into differently-privileged subprograms. Provos et al. [PFH03] divide the OpenSSH network login server into more and less-trusted components that operate as separate Unix processes, communicating using sockets and shared memory. They show that this technique is effective in containing attacks, but splitting a once-monolithic program into separately communicating tasks is nontrivial. That the two processes have separate operating-system level state makes some aspects of isolation easier (the trusted code can run concurrently with the untrusted) and others more difficult (the untrusted code can still make many system calls). While an SFI-based approach would require the same design decisions about which parts of the authentication process require privilege, the implementation of communication would be simpler: for instance, the privileged code could access data structures in the unprivileged sandbox completely transparently. Also related are system-level sandboxing techniques for complete programs, such as those described by Peterson et al. [PBP02]. These take advantage of the operating system design to provide isolation and to define the interface to the untrusted code (system calls); this makes them very convenient to apply to unmodified applications, but prevents them from giving different privileges to different components of a program, as would often be desirable.

Incidentally, we might point out that SFI subsumes some mechanisms that have previously been suggested as partial measures to make program subversion more difficult. For instance, PittSFIeld prohibits the execution of code on the stack and reduces the number of possible targets of an overwritten function pointer; other tools like MiSFIT restrict the set of targets further, and protect procedure return addresses. However, these side-effects should not be confused with the isolation policy that they are intended to support, as described above. SFI implementations do not provide general protection against attacks on the untrusted code; they simply contain those attacks within the component.

9.3 Gleipnir/CFI

In concurrent work [ABEL05a], the Gleipnir project at Microsoft Research has investigated a binary-rewriting security technique called Control-Flow Integrity, or CFI. As suggested by the name, CFI differs from SFI in focusing solely on constraining a program's jumps: in the Gleipnir implementation, each potential jump target is labelled by a 32-bit value encoded in a no-op instruction, and an indirect jump checks for the presence of an appropriate tag before transferring control. This approach gives finer control of jump destinations than the SFI techniques of Wahbe et al., or PittSFIeld, though the ideal precision could only be obtained with a careful static analysis of, for instance, which function pointers might be used at which indirect call sites. In terms of the discussion of Section 9.2, this makes CFI much more effective at preventing program subversion. In the basic presentation, CFI relies on an external mechanism (such as hardware) to prevent changes to code or jumps to a data region, but it can also be combined with inserted memory-operation checks, as in SFI, to enforce these constraints simultaneously.

In the control-flow-only use, CFI has overheads ranging from 0 to 45% on a Pentium 4; the wide variation presumably results from a large overhead on indirect jumps combined with little overhead on any other operation. By comparison, PittSFIeld imposes a smaller overhead on jumps, but significant additional overheads on other operations. Performance measurements for CFI with software memory protection were not included in [ABEL05a], so a more detailed performance comparison is not yet possible. Like PittSFIeld, Gleipnir/CFI performs a separate verification to enforce proper rewriting at load time, so the compiler and binary instrumentation infrastructure need not be trusted. The Gleipnir authors have written a human-checked proof [ABEL05b] that a CFI-protected program will never make unsafe jumps, even in the presence of arbitrary writes to data memory. However, the proof is formulated in terms of a miniature RISC architecture whose encoding is not specified. This is somewhat unsatisfying, as the safety of the real Gleipnir/CFI technique is affected in subtle ways by the x86 instruction encoding (for instance, the possibility that the immediate tag value used in the comparison at a jump site might be itself interpreted as a safe jump target tag.)

9.4 Static C safety mechanisms

Beyond SFI, much other work has used pre-execution binary or source rewriting to work around the unsafe aspects of C programs. Besides security, finding bugs during development is another major application. Such tools generally provide a more precise guarantee of memory behavior than SFI, for instance that writes only occur to allocated bytes, or even only to the allocated block from which a pointer was derived, before the block was freed. However, such precise guarantees require additional runtime overheads. Perhaps the best-known binary rewriting tool is Purify [HJ92], which checks for memory-usage bugs. Because it is targeted only for use during debugging, its relatively high overheads are not problematic. Tools that rewrite C source code to perform memory usage tracing can achieve even more precise error tracking, again at the expense of performance; examples include the systems of Austin et al. [ABS94] and more recently Xu et al. [XDS04]. Techniques such as ours achieve better performance by enforcing a simpler memory policy; our technique is also simpler and more scalable than source-code rewriting approaches.

Another class of program rewriting tools (often implemented as compiler modifications) are focused on ensuring fairly narrow security policies, for instance that the procedure return address on the stack is not modified [CPM$^+$98]. Such tools can be very effective in their intended role, and tend to have low overheads, but they do not provide protection against more esoteric subversion attacks. They also do not provide isolation between components, and are not intended for untrusted code. They could, however, be used in conjunction with SFI if both isolation and protection from subversion are desired.

9.5 Dynamic translation mechanisms

Several recent projects has borrowed techniques from dynamic optimization to rewrite programs on the fly; such techniques allow for fine control of program execution, as well as avoiding the difficulties of static binary rewriting. Valgrind [NS03] is a powerful framework for dynamic rewriting of Linux/x86 programs, which is best known for Purify-like memory checking, but can also be adapted to a number of other purposes. Valgrind's rewriting uses a RISC-like intermediate language, sacrificing performance for ease of development of novel applications. A research tool with a more security-oriented focus is Scott and Davidson's Strata [SD02]; it has achieved lower overheads (averaging about 30%) while enforcing targeted security policies such as system call interception. A similar but even higher performance system is Kiriansky et al.'s program shepherding [KBA02], based on the DynamoRIO dynamic translation system. Their work concentrates on preventing attacks on a program's control flow, as an efficient and transparent means to prevent stack- and function-pointer-smashing vulnerabilities from being exploited. Further innovations include efficient techniques for protecting the return stack, such as saving 16-bit tags in XMM registers [KBA03], an idea that could also be applied to SFI. In general, such dynamic techniques can, if carefully implemented, enforce control flow policies more efficiently than a technique like ours. This control flow restriction can then be leveraged for a variety of other purposes, including various types of memory protection, which can be implemented very similarly in dynamic or static contexts. A disadvantage of dynamic techniques is that they are inherently somewhat complex and difficult to reason about, relative to a comparable static translation.

9.6 Low-level type safety

Verifying that low-level program representations preserve safety features has been a theme of much recent research, though more emphasis has been placed on guarantees using static invariants such as type systems, rather than inlined dynamic checks as in SFI. For instance, typed assembly language [MWCG99, MCG$^+$99] can provide quickly checkable, fine-grained safety properties for a sublanguage of x86 assembly, but requires that the original program be written in a type-safe language. Type inference can also be used to transform C code into a type-safe program with a minimal set of dynamic checks, as in the CCured system [CHM$^+$03]. These inferred types can encode more detailed structural information than SFI, making them useful for bug-finding or sharing of data structures across trust boundaries, but the inference process is complicated and hard to fully automate. SFI's weaker guarantees can be provided with less programmer intervention, and because they are coarse-grained, the overhead of checking them is comparable to the checks in a CCured-like system, even if more of them are superfluous. Because they can constrain writes to a occur on specific objects, type-based safety properties are generally quite effective at preventing subversion attacks that overwrite function pointers.

Proof-carrying code [NL98] represents a more general framework for software to certify its own trustworthiness. Most work on PCC has focussed on type-like safety properties, but under the banner of foundational PCC [App01], efforts have been made to place proofs on a more general footing, using fully general proof languages that prove safety with respect to concrete machine semantics [MA00]. This approach seems to carry the promise, not yet realized, of allowing any safe rewriting to certify its safety properties to a code consumer (the most flexible framework described in the literature is probably that of Chang, Chlipala, Necula, and Schneck [CCNS05]). For instance, one could imagine using the lemmas from the proof of Section 6 as part of a foundational safety proof for a PittSFIeld-rewritten binary. It is unclear, however, if any existing foundational PCC systems are flexible enough to allow such a proof to be used.

10 Conclusion

We have argued that software-based fault isolation can be a practical tool in constructing secure systems. Using a novel technique of artificially enforcing alignment for jump targets, we show how a simple sandboxing implementation can be constructed for an architecture with variable-length instructions like the x86. We give two new optimizations, which along with previously known ones minimize the runtime overhead of the technique, and argue for the importance of an architecture that includes separate verification. We have constructed a machine-checked soundness proof of our technique, to further enhance our confidence in its security. Finally, we have constructed an implementation of our technique which demonstrates separate verification, gives performance comparable to previous unsafe tools, and is easily scalable to realistically large and complex applications.

Acknowledgements

Michael Ernst and the members of the MIT Program Analysis Group provided helpful suggestions on the presentation of this work. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

References

[ABEL05a] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. Technical Report MSR-TR-05-18, Microsoft Research, Redmond, WA, February 2005.

[ABEL05b] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. Technical Report MSR-TR-05-17, Microsoft Research, Redmond, WA, February 2005.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, USA, June 1994.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, June 2001.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[BY96] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1), 1996.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The open verifier framework for foundational verifiers. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, Long Beach, California, January 2005.

[CHM$^+$03] Jeremy Condit, Mathew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[CPM$^+$98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Austin, Texas, January 1998. USENIX Association.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, September 1999.

[GPF03] Andreas Gal, Christian W. Probst, and Michael Franz. A denial of service attack on the Java bytecode verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, November 2003.

[HJ92] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, January 20–24, 1992.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002. USENIX Association.

[KBA03] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Execution model enforcement via program shepherding. Technical Report MIT-LCS-TM-638, Massachusetts Institute of Technology Laboratory for Computer Science, May 2003.

[KM97] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[Lu00a] Fei Lu. C Plus J software architecture. Undergraduate thesis, Shanghai Jiaotong University, June 2000.

[Lu00b] Fei Lu. Introducing C+J research project, 2000. `http://flyland.cs.jhu.edu/cpj/CPJ_guide.htm`.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *17th International Conference on Automated Deduction (CADE-17); Lecture Notes in Artificial Intelligence 1831*. Springer-Verlag, June 2000.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, 17–19 June 1998.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[PBP02] David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002. USENIX Association.

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., August 2003. USENIX Association.

[Riv92] Ronald Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. Status: INFORMATIONAL.

[SD02] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 2002 Annual Computer Security Application Conference*, Las Vegas, Nevada, December 2002.

[Sil96] Scott M. Silver. Implementation and analysis of software based fault isolation. Technical Report PCS-TR96-287, Dartmouth College, June 1996.

[SS97] Christopher Small and Margo Seltzer. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, June 1997.

[WL98] Robert S. Wahbe and Steven E. Lucco. Methods for safe and efficient implementations of virtual machines. U.S. Patent 5,761,477, June 1998. Assigned to Microsoft Corporation.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993.

[XDS04] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, Newport Beach, CA, USA, November 2004.