Evaluating SFI 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 that the code be prevented from modifying memory or executing instructions except as explicitly allowed. Software-based fault isolation (SFI) or "sandboxing" enforces such a policy by rewriting the untrusted code at the instruction level. However, the original sandboxing technique of Wahbe et al. is applicable only to RISC architectures, and most other previous work is either insecure, or has been not described in enough detail to give confidence in its security properties. We present a new sandboxing technique that can be applied to a CISC architecture like the IA-32, and whose application can be checked at load-time to minimize the TCB. We describe an implementation which provides a robust security guarantee and has low runtime overheads (an average of 21% on the SPECint2000 benchmarks). We evaluate the utility of the technique by applying it to untrusted decompression modules in an archive tool, and its safety by constructing a machine-checked proof that any program approved by the verification algorithm will respect the desired safety property.

1 Introduction

Secure systems often need to execute a code module while constraining its actions with a security policy. The code might come directly from a malicious author, or it might have bugs that allow it to be subverted by maliciously chosen inputs. The system designer chooses a 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. Software-based fault isolation (SFI) implements such isolation via instruction rewriting, but previous research left the practicality of the technique uncertain. The original SFI technique works only for RISC architectures, and much followup research has neglected key security issues. By contrast, we find that SFI can be implemented for the x86 with runtime overheads that are acceptable for many applications, and that the technique's security can be demonstrated with a rigorous machine-checked proof.

The most common technique for isolating untrusted code is the use of hardware 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; few restrictions are placed on what the untrusted code can do. A disadvantage of hardware protection, however, is that interaction across a process boundary (i.e., via system calls) is coarse-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, making fine-grained sharing of data between trusted and untrusted components relatively easy. However, type systems have some limitations as a security mechanism. First, unlike operating systems, which are generally language independent, type system approaches are often designed for a single language, and can be hard to apply to at all to 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

```
void f(int arg,  int arg2,
       int arg3, int arg4) {
    return;
}
void poke(int *loc, int val) {
    int local;
    unsigned diff = &local - loc - 2;
    for (diff /= 4; diff; diff--)
        alloca(16);
    f(val, val, val, val);
}
```

Figure 1: Example attack against SFI systems which depend on the compiler's management of the stack for safety. The function `poke` modifies the stack pointer by repeatedly allocating unused buffers with `alloca` until it points near an arbitrary target location `loc`, which is then overwritten by one of the arguments to `f`. MiSFIT [25] and Erlingsson's x86 SASI tool [10] allow this unsafe write because they incorrectly assume that the stack pointer always points to the legal data region.

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 writes and control flow. This class of techniques is known as "software-based fault isolation" (SFI for short) or "sandboxing" [27]. Previous SFI techniques were applicable only to RISC architectures, or their treatment of key security issues was incomplete, faulty, or never described publicly. For instance, several previous systems [25, 10] depended for their safety on an assumption that a C compiler would manage the stack pointer to keep the untrusted code's stack separate from the rest of memory. As shown in the example of Figure 1, this trust is misplaced, not just because compilers are large and may contain bugs, but because the safety guarantees they make are loosely specified and contain exceptions. (Concurrently with the research described here, some other researchers have also developed SFI-like systems that include more rigorous security analyses; see Section 10 for discussion.)

Many systems programming applications can benefit from a code isolation mechanism that is efficient, robust, and easily applicable to existing code. A useful technique to improve the reliability of operating systems is to isolate device drivers so that their failures (which may include arbitrary memory writes) do not corrupt the rest of a kernel. The Nooks system [26] achieves such isolation with hardware mechanisms that are robust, but impose a high overhead when many short cross-boundary calls are made; SFI could provide similar protection without high per-call overheads. To reduce the damage caused by attacks on network servers, they should be designed to minimize the amount of code that requires high (e.g., root) privileges; but retrofitting such a design on an existing server is difficult. Dividing servers along these lines by using separate OS-level processes [23, 14] is effective at preventing vulnerabilities, but is far from trivial because of the need to serialize data for transport and prevent an untrusted process from making damaging system calls. SFI could make such separation easier by automatically preventing system calls and making memory sharing more transparent. Section 8 discusses VXA [11], an architecture that ensures compressed archives will be readable in the future by embedding an appropriate decompressor directly in an archive. Applying our SFI tool to VXA we see that it very easily obtains a strong security guarantee, without imposing prohibitive runtime overheads. Note that all of these examples include large existing code bases written in C or C++, which would be impractical to rewrite in a new language; the language neutrality of SFI techniques is key to their applicability.

This paper:

- Describes a novel SFI technique directly applicable to CISC architectures like the Intel IA-32 (x86), as well as two optimizations not present in previous systems (Sections 3 and 4).
- Explains how using separate verification, the security of the technique depends 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).
- Analyzes in detail the performance of an implementation on the standard SPECint2000 benchmarks (Section 7).
- Evaluates the implementation as part of a system to safely execute embedded decompression modules (Section 8).
- Gives a machine-checked proof of the soundness of the technique (specifically, of the independent safety verifier) to provide further evidence that it is simple and trustworthy (Section 9).

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 (the version described here is 0.4), as are the formal model and lemmas used in the machine-checked proof. They can be downloaded from the project web site at `http://pag.csail.mit.edu/~smcc/projects/pittsfield/`.

[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.

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 maliciously designed jumps in the input code. The first approach to solve these challenges was the original SFI technique (called "sandboxing") of Wahbe, Lucco, Anderson, and Graham [27]. (The basic idea of rewriting instructions for safety had been suggested earlier, notably by Deutsch and Grant [7], but their system applied to code fragments more limited than general programs.)

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 the address `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 (Deutsch and Grant's system did not allow them). 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 is no reason why a correct program *would* jump directly to an unsafe store instruction; the technique is needed for incorrect or maliciously designed programs.

Wahbe et al. implemented their technique for two RISC architectures, the MIPS and the Alpha. Because memory reads are more common than writes and less dangerous, their implementation only checked stores and not loads, a tradeoff that has also been made in most subsequent work, including ours. (In the experiments in [25], adding protection for out-of-bounds reads often more than doubled overhead compared to checking only writes and jumps.) Because separate dedicated registers are required for the code and data regions, and because constants used in the sandboxing operation 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 running the extension run in a separate process, because communication between the extension and the main server becomes inexpensive.

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

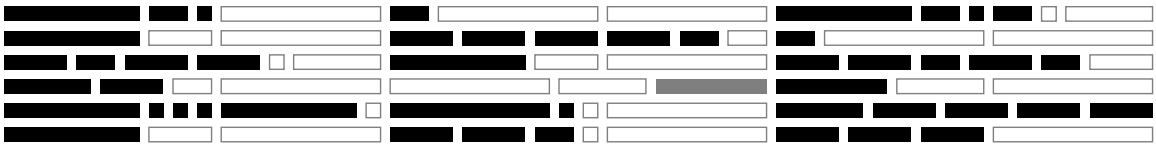

Figure 2: Illustration of the instruction alignment enforced by our technique. Black filled rectangles represent instructions of various lengths present in the original program. Gray outline rectangles represent added no-op instructions. Instructions are not packed as tightly as possible into chunks 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 are aligned.

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 2. 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. To ensure that an otherwise unsafe operation and the check of its operand cannot be separated, PittSFIeld additionally enforces that such pairs of instructions do not cross chunk boundaries, making them atomic. Thus, our technique does not need dedicated registers as in classic SFI. A scratch register is still required to hold the effective address of an operation while it is being checked, but it isn't required that the same register be used consistently, or that other uses of the register be prohibited. (For reasons of implementation simplicity, though, our current system consistently uses `%ebx`.)

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.

Special registers. 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. Since `%ebp` is generally set only at the start of a function but then used repeatedly thereafter, checking its value at each use is inefficient. A better strategy is to make sure that `%ebp`'s value is a safe data pointer everywhere by checking its value after each modification. This policy treats `%ebp` like the reserved registers of Wahbe et al., but since `%ebp` is already reserved by the ABI for this purpose, the number of available general-purpose registers is not decreased.

Guard regions. The technique described in the previous paragraph 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.

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 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.

The danger of a sequence of small modifications is illustrated in the example of Figure 1: each call to `alloca` decrements `%esp` by a small amount but does not use it to read or write. Our system prevents this attack by requiring a modified `%esp` to be checked before a jump.

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. Clearly, the untrusted code should not be able to perform any action that is unsafe; but what should happen when the untrusted code attempts an unsafe action? The most natural choice 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. The latter policy can be more efficient because no branch is required: the code simply sets the bits of the address appropriately and uses it. If the address was originally illegal, it will 'wrap around' to some legal, though likely not meaningful, location.

There are certainly applications (such as debugging) where such arbitrary behavior 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. Our rewriting tool supports both modes of operation, but we here follow Wahbe et al.in concentrating on the more efficient ensure-only mode, which we consider more realistic. Experiments described in a previous report [18] show that the checking mode introduces an average of 12% further overhead over the ensure-only mode on some realistic examples.

One-instruction address operations. For an arbitrarily chosen code or data region, the sandboxing instruction must check (or, according to the optimization above, ensure) that certain bits of an address are set, and others are clear. This requires two instructions: an AND instruction to turn some bits off and an OR instruction set others. 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 cleared, and not also set.

PittSFIeld by default uses code and data regions of 16MB each, starting at the addresses `0x10000000` and `0x20000000` respectively. The code sequence to ensure that an address in `%ebx` is legal for the data region is:[2]

```
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. On examples such as the set of larger programs appearing in a previous report [18], disabling this optimization increases PittSFIeld's overhead over normal execution by about 10%.

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 (in this example and the next, the final two instructions must be in a single chunk):

```
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 a recursive implementation of the Fibonacci function, this optimization makes an enormous difference, reducing 95% overhead to 40%. In more realistic examples, the difference is smaller but still significant; for the SPECint2000 benchmarks discussed in Section 7, disabling this optimization increases

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

the average overhead from 21% to 27%, and almost doubles the overhead for one program, 255.vortex.

With the exception of this optimization, the rest of the PittSFIeld system can maintain its security policy even if arbitrary changes to the data region occur between instructions, because instructions always move addresses to registers before checking them. However, the `ret` instruction unavoidably uses the stack, so this optimization is applicable under the more limited attack model in which untrusted data changes come from a single untrusted thread. The optimization should not be used if multiple threads run in the same data sandbox, or if other untrusted memory changes (such as memory-mapped I/O) might occur in parallel.

5 Verification

The intended use of PittSFIeld is that 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 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 [20]. Though the advantages of this architecture are well known, they have been neglected by some previous SFI implementations, leading to predictable problems with usability and security (see Section 10.1).

Responsibility for ensuring the safety of execution in the PittSFIeld system lies with a verifier which examines the rewritten code just prior to execution, conservatively checking properties which, if true, ensure that execution of the code will not violate the system's security policy. In a more complex system, one could imagine the rewriting process supplying hints describing why the rewritten code satisfies the security policy (like the proof in a proof-carrying code system), but PittSFIeld's policies are simple enough that this is not necessary. In particular, the verifier does not require debugging or symbol-table information; the verifier must disassemble the rewritten code, but the rewriter ensures that the disassembly can be performed in a single pass without respect to function boundaries. The role of the verifier is to prove that the rewritten code is safe, so its design is best thought of as automating such a proof. Section 9 will describe in more detail how that intuitive proof can be formalized.

To understand how the verification works, it is helpful to borrow concepts from program analysis, and think of it as a conservative static analysis. The verifier checks a 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 the verifier fails to recognize that some programs have the safety property, as long as whenever it concludes that one does, it is correct. If the original program was correct, it already had this safety property; the rewriting simply makes 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 is 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, as new instructions would not necessarily match the static properties. 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 the verifier checks as being preserved by each instruction step.

6 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. 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 rewriter is 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. 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.

Because it operates on assembly code, our prototype rewriting tool is intended to be used by a code producer. A system that instead operates on off-the-shelf binaries without the code producer's cooperation is often described as a goal of SFI research, but has rarely been achieved in practice. The key difficulty is that binaries do not contain enough information to adjust jumps when instructions are added: for instance, it may not be possible to distinguish between an address referring to an instruction and an integer with the same numeric value. A more feasible approach is to operate on binaries supplemented with additional relocation information, such as the debugging information used by the Vulcan library in [1], or the SELF extension to ELF proposed in [8].

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 the verification's running time will be linear. (A verification technique with bad worst-case performance can allow a denial-of-service attack [12]).

The rewriting phase of PittSFIeld is implemented as a text processing tool, of about 720 lines of code, 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 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. However, such save and restore operations can be costly on modern architectures, so to prevent GCC from moving comparisons away from their corresponding branches, we disable instruction scheduling with the `-fno-schedule-insns2` option when compiling for PittSFIeld. An example of the rewriter's operation on a small function is shown in Figure 3.

We have implemented two prototypes for the verification phase of PittSFIeld, which implement the same algorithm. Because they use a single disassembly pass, the verifiers enforce alignment by checking that an instruction in the single stream must appear at each chunk starting address. The verifiers currently verify 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. As mentioned in Section 5, the verifiers are essentially finite-state: at each code location, they keep track of variations from the standard safety invariant, checking them and then updating their 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.

Our first verifier is implemented using the same text-processing framework as the rewriter: it is a filter that parses the output of the disassembler from the GNU "binutils" package (the program named `objdump`), and represents about 500 lines of code. Our second verifier is implemented directly in the program that loads and executes sandboxed code objects, using a pre-existing disassembly library; this allows for a better assessment of the performance overheads of verification. Though it does not yet check the complete safety policy, the second verifier is complete enough to give rough performance estimates: for instance, it can verify the 2.7MB of rewritten code for GCC, the largest of the programs from Section 7, in about half a second. Both of our verifiers are much smaller than the disassemblers they use, so the total amount of trusted code could be reduced by disassembling only to the extent needed for verification, but using existing disassemblers takes advantage of other users' testing. Performing more targeted disassembly in this way would also be a way to further improve performance. PittSFIeld supports a large subset of the x86 32-bit protected mode instruction set, but supervisor mode instructions, string instructions, and multimedia SIMD (e.g. MMX, SSE) instructions are not supported; the verifier will reject any program containing an instruction it does not recognize.

```
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 shown 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).

7 Performance results

To asses the time and space overheads imposed by our technique, we used our PittSFIeld tool to run standalone 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 800 lines of C code, including blank lines and comments. A previous technical report [18] gives results for an older version of PittSFIeld run on a set of microbenchmarks, and some larger applications. For better comparison with other work, we here concentrate on a standard set of compute-intensive programs, the integer benchmarks from the SPEC CPU2000 suite.

The SPECint2000 suite consists of 12 programs and reference inputs intended to test the performance of CPUs, compilers, and memory subsystems. One of the programs is written in C++, and the rest in C. In our tests, we compiled the programs with GCC or G++ version 3.3.5 at the `-O3` optimization level. The test system was a 3.06GHz Pentium 4 "Northwood", with 512KB of cache and 2GB of main memory, running Debian Linux 3.1 with kernel version 2.4.28 and C library version 2.3.2. We changed the layout of the code and data sandbox areas to allow a larger data area. Each test was run five times with the reference inputs using the standard SPEC scripts; we discarded the slowest and fastest runtimes and took the average of the remaining three.

In order to measure the effect on performance of different aspects of PittSFIeld's rewriting, we ran the programs using a number of treatments, representing increasing subsets of the transformation the real tool performs. Figure 4 shows the increase in runtime overhead as each transformation is enabled, from bottom to top. The base treatment uses PittSFIeld's program loader, but compiles the programs with normal optimization and uses none of the features of the rewriter. The measurements of Figure 4 are all measured as percentage overhead relative to the base treatment. The first (bottom) set of bars in Figure 4 represents disabling instruction scheduling with an option to GCC. Disabling this optimization has a small performance penalty, but avoids higher overheads later by reducing the need to save and restore the EFLAGS register as discussed in Section 6. The next set of bars represents the effect of directing GCC to avoid using the `%ebx` register in its generated code, reducing the number of general purpose registers from 6 to 5; PittSFIeld requires `%ebx` to be available to hold the effective address of indirect writes and jumps. The next treatment, labelled "padding", reflects the basic cost of requiring chunk alignment: the rewriter adds enough no-op instructions so that no instruction crosses a 16-byte boundary, and every jump target is 16-byte aligned. The next set of bars, labelled "NOP sandboxing", attempts to measure all of the additional overheads related to PittSFIeld's code size increase, beyond those measured in "padding". To achieve this, this treatment adds just as many bytes of new instructions as PittSFIeld normally would, but makes all of them no-ops: this includes both sandboxing instructions, and additional padding required for the new in-

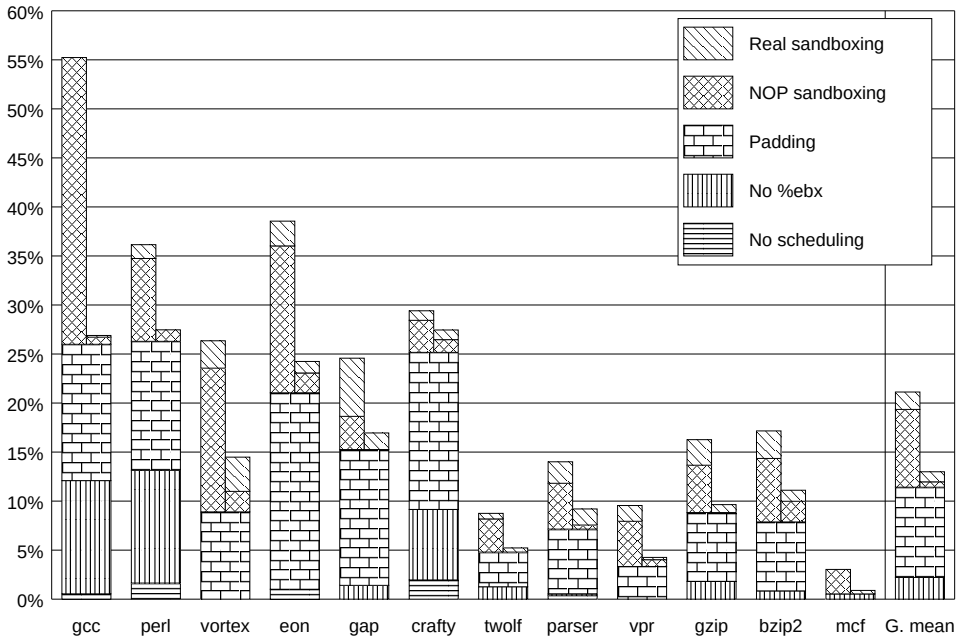

Figure 4: Runtime overheads of PittSFIeld for the SPECint2000 benchmarks, by source. The left half of each bar represents overhead when both jumps and memory writes are protected; the right half shows the overhead of protecting jumps only. The programs are listed in decreasing order of binary size. See the body of the text, Section 7, for details on the meaning of each type of overhead.

structions and to keep some instruction pairs in the same chunk. Finally, the last set of bars represents the complete PittSFIeld transformation; exactly the same number of instructions as "NOP sandboxing", but with AND instructions instead of no-ops as appropriate. For the last two treatments, we also considered another subset of PittSFIeld's rewriting: the left half of each bar shows the overhead when PittSFIeld is used to protect both writes to memory and indirect jumps; the right half shows the overhead for protecting jumps only. For some combinations of programs and treatments, we actually measured a small performance improvement relative to the previous treatment, either because of the inaccuracy of our runtime measurement or because of unpredictable performance effects such as instruction cache conflicts. In these cases the corresponding bars are omitted from the figure.

The SPECint2000 results shown in Figure 4 make clear which of the sources of PittSFIeld's overhead are most significant. Disabling instruction scheduling has little to no effect at this scale, and the sandboxing instructions themselves, bitwise operations on registers, are almost as cheap as no-ops. The effect of reducing the number of available registers varies greatly between programs, but is never the most important overhead. The largest source of overhead is unfortunately the one most fundamental to the technique, the increase in the number of instructions. Added no-op instructions cause two kinds of overhead: first, they take time to execute themselves, and second, they use cache space that would otherwise be used by useful instructions. The relative importance of these two effects can be estimated by comparing the size of the "padding" overhead across programs. Though the proportion of padding instructions can be expected to vary slightly among programs (for instance, being smaller in programs with larger basic blocks), the variation in padding overheads is larger that could be explained by this effect, so the remaining variation must be explained differences in instruction cache pressure. For instance, the padding overhead is larger for large programs than for small ones. The very low overheads for `mcf` likely have two causes: first, it is the smallest of the benchmarks, so instruction cache pressures affect it the least; second, it makes many random accesses to a large data structure, so its runtime depends more on main memory latency than anything happening on the CPU. The final column of Figure 4 shows the average overhead of the technique over all the programs (a geometric mean). This is approximately 21% for memory and jump protection, and 13% for jump protection only.

Figure 5 show how PittSFIeld's transformation affects the size of the code. The row labelled "Size" shows the size of a binary rewritten by PittSFIeld, in bytes (K = 2^{10}, M = 2^{20}). This size includes the program and the stub li-

Program	gcc	perl	vortex	eon	gap	crafty	twolf	parser	vpr	gzip	bzip2	mcf
Size	2.7M	1.2M	1010K	923K	853K	408K	390K	276K	267K	109K	108K	50K
Ratio	1.84	1.96	1.63	1.72	1.84	1.62	1.80	1.92	1.67	1.65	1.63	1.74
Compressed	1.05	1.07	0.98	1.05	1.05	1.06	1.08	1.06	1.07	1.10	1.09	1.13

Figure 5: PittSFIeld space overheads for the SPECint2000 benchmarks. "Size" is the size of the PittSFIeld-rewritten binary. "Ratio" is the ratio of the size of the rewritten binary to the size of a binary generated without rewriting. "Compressed" is like "Ratio", except with both binaries compressed with `bzip2` before comparing their sizes.

brary, both rewritten by PittSFIeld in its default mode (16-byte chunks, both memory and jump protection). The row "Ratio" shows the ratio of the size of an object file processed by PittSFIeld to that of an unmodified program. The row "Compressed" is analogous, except that both the transformed and original object files were first compressed with `bzip2`. Which of these measurements is relevant depends on the application. The most important effect of PittSFIeld's size increase in most applications is likely its effect on performance, discussed in the previous paragraph. Uncompressed size is relevant for memory usage, for instance on smaller devices. Compressed size is more relevant, for instance, to the cost of storing and distributing software; the compressed ratios are smaller because the added instructions tend to be repetitive.

8 Application case study

To assess the usability of PittSFIeld for a realistic application, this section investigates using PittSFIeld as the isolation mechanism for VXA, a system in which compressed archives contain their own decompressors [11]. A major challenge to our ability to preserve digital information for the future is the proliferation of incompatible file formats. Compression formats are a particular problem: for instance while uncompressed audio formats have been stable since the early 1990s, four major new formats for compressed audio have become popular since 2000. By comparison, the basic IA-32 architecture retains backwards compatibility with software written for the first 386es in 1985. To exploit these relative rates of change, the VXA system introduces an archive file format and tools called vxZIP and vxUnZIP that extend the well-known ZIP format by including decompressors in a standardized IA-32 binary format inside compressed archives. IA-32 was chosen as the standard decompressor format so that vxZIP and vxUnZIP can be used with low overhead today.

Clearly a key challenge for vxUnZIP is to run the supplied decompressor without allowing a malicious decompressor to compromise security. At the same time, it is desirable for the decompressors to run in the same process as the rest of vxUnZIP. Compared to using a separate OS-level process for isolation, running in a single process avoids performance overheads associated with process creation and copying data between processes, but the most important advantage is the ease of supplying a limited interaction interface to the compressor instead of the operating system's full set of system calls. VxUnZIP achieves these goals with a virtualized execution environment, VX32, that combines two different isolation mechanisms. To prevent untrusted code from reading or writing memory locations outside the sandbox, VX32 uses the IA-32 legacy segmented memory addressing mode to restrict the locations available to it. This requires operating system support to modify the local descriptor table (LDT), and segmentation is not supported in the 64-bit mode of newer processors, though VX32 can still work in 32-bit compatibility mode. To control which instructions the untrusted code executes (to protect for instance against unsafe indirect jumps or instructions that modify the segment registers), VX32 uses dynamic translation, rewriting code one fragment of a time into a cache and interpreting indirect jumps.

The author of VXA was not aware of PittSFIeld at the time it was designed, but to examine whether PittSFIeld would be a suitable replacement for VX32, we used it to protect the execution of the six VXA decompression modules demonstrated in [11]. We used VX32's virtual C library rather than the one used in Section 7; this required implementing VXA's four virtual system calls (`read`, `write`, `_exit`, and `sbrk`). We also used VX32's library of math functions, but compiled to use the x87-style floating point instructions supported by PittSFIeld rather than the SSE2 ones VX32 uses. The runtime overheads of VX32 and PittSFIeld are compared in Figure 6. Zlib and BZip2 are decompressors for the same general-purpose compression formats as the SPECint2000 gzip and bzip2 programs (which also include compression); JPEG and JPEG2000 are lossy image compression formats, and FLAC and Vorbis are lossless and lossy audio formats respectively. In each case the programs decompressed large single files of each format. To minimize I/O overhead, we made sure the input files were cached in memory, and sent the decompressed output to `/dev/null`; measurements are based on elapsed time. The system was the same Pentium 4 machine described in Section 7, except that VX32 uses a specially compiled version of GCC 4.0, and the native

	Zlib	BZip2	JPEG	JPEG2000	FLAC	Vorbis	Geom. Mean
VX32	1.006	0.975	1.034	1.283	0.954	0.948	1.028
PittSFIeld jump-only	1.238	1.018	1.134	1.114	1.142	1.239	1.145
PittSFIeld full	1.398	1.072	1.328	1.211	1.241	1.458	1.278

Figure 6: Run time ratios for VX32 and PittSFIeld on the VXA decompressors, compared to natively compiled decompressors.

build uses Debian's GCC 4.0 to match.

The occasional speedup of execution under VX32, also seen in [11], appears to result from increased instruction locality introduced by translating dynamic execution traces sequentially. (For instance, VX32 is faster than native execution in the FLAC example even though it executes more instructions, 97.5 billion compared to 96.0 billion.) These examples have also been tuned to minimize the number of indirect jumps: two frequently called functions were inlined. The measured overhead of PittSFIeld for the vxUnZIP examples is noticeably higher than that of VX32, but still not excessive (28% on average). PittSFIeld's overhead is also smaller when protecting only jumps (averaging 15%); this simulates the performance of combining PittSFIeld with VX32-like segment-based memory protection.

Some qualitative features also affect the choice between PittSFIeld and VX32. An advantage of VX32 is that it prevents the decompressor from reading memory outside its sandbox; though not as critical for security as preventing writes, this is useful to ensure decompressors are deterministic. Controlling reads is possible with SFI, but would significantly increase the technique's overhead. On the other hand, VX32's use of segment registers decreases its portability, including to future processors; conversely, VX32's use of SSE2 floating point currently keeps it from working on older processors, though the latter limitation is not fundamental. Arguably, PittSFIeld's simple, static approach and separate verification make it more trustworthy, but VX32 is not yet as mature as PittSFIeld, and it is significantly simpler than previous dynamic translation systems.

9 Formal Analysis

Having restricted ourselves 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 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.

We have constructed the proof using ACL2 [13]. ACL2 is a theorem-proving system that combines a restricted subset of Common Lisp, used to model 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. The proof required less than two months of effort by a user with no previous experience with proof assistants (the first author). An experienced ACL2 user could likely have produced a more elegant proof in less time; our inexperience in choosing abstractions also made the going more difficult as the proof size increased. An example of a function from the executable model and a lemma we have proved about it are shown as the first two parts of Figure 7. A disadvantage of ACL2 compared to some other theorem provers is that its proofs cannot be automatically checked by a simpler proof checker. However, ACL2 has been well tested by other academic and industrial users, and its underlying logic is simple, so we still consider it trustworthy.

The precise statement of our final safety result appears as the bottom part of Figure 7. It is a correctness result about the verifier, modeled as a predicate `mem-sandbox-ok` on the state of the code region before execution: if the verifier approves the rewritten code, then for any inputs (modelled as the initial contents of registers and the data region), execution of the code will continue forever without performing an unsafe operation. (Unlike the real system, the model has no `exit()` function.) Note that the rewriter does not appear in the formal proof, so the proof makes no claims about it: for instance, we have not proved that the code produced by the rewriter has the same behavior as the original code. Though a statement like that could be formalized, it would require a number of additional hypotheses; in particular, because the rewriter changes the

```
(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-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))))
(defthm safety
  (implies
    (and (mem-p mem) (mem-sandbox-ok mem)
         (addr-p eax) (addr-p ebx) (addr-p ecx)
         (addr-p edx) (addr-p esi) (addr-p edi)
         (addr-p ebp) (addr-p esp)
         (addr-p eflags) (data-region-p ebp))
    (consp
     (step-for-k
       (x86-state (code-start) eflags eax ebx
                  ecx edx esi edi ebp esp mem)
        k))))
```

Figure 7: From top to bottom, a typical function definition, a typical lemma, and the final safety result 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. The safety theorem states that if a memory image `mem` passes the verifier `mem-sandbox-ok`, then whatever the initial state of the registers, execution can proceed for any number of steps (the free variable k) without causing a safety violation (represented by a `nil` return value from `step-for-k`, which would not satisfy the predicate `consp`).

address of instructions, code that say examined the numeric values of function pointers would not behave identically.

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 subset used in the current version of the proof appears in Figure 8. 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. Though small compared to the number of instructions allowed by the real tool, this set is similar to the instruction sets used in recent similar proofs [2, 29]. We constructed the proof by beginning with a minimal set of instructions and then adding additional ones: adding a new instruction similar to an existing one required few changes, but additions that required

```
nop         mov addr, %eax     xchg %eax, %ebx
inc %eax    mov %eax, addr     xchg %eax, %ebp
jmp addr    and $immed, %ebx   mov %eax, (%ebx)
jmp *%ebx   and $immed, %ebp   mov %eax, (%ebp)
```

Figure 8: List of instructions in the subset considered in the proof of Section 9.

a more complex safety invariant often involved extensive modifications. 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 extends to the complete system. A related concern is whether the simulated x86 semantics match those of a real processor: though the description of the subset used in the current proof can be easily checked by hand, this would be impractical for a more complete model. To facilitate proofs like ours in the future, as well as for applications such a foundational proof-carrying code (see Section 10.6), it should be possible to generate a description of the encoding and semantics of instructions from a concise, declarative, and proof-environment-neutral specification.

In total, the proof contains approximately 60 function definitions and 170 lemmas, over about 2400 lines of ACL2 code. The description of the model and the statement of the safety result require about 500 lines; assuming ACL2's verification is correct, only this subset must be trusted to be convinced of the truth of the result. The technical details of the proof are straightforward and rather boring; for space reasons, we do not discuss them further here. Interested readers are referred to a companion technical report [17]; the proof in its machine-checkable form is also available from the PittSFIeld project website.

10 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.

10.1 Other SFI implementations

Binary sandboxing was introduced as a technique for fault-isolation by Wahbe, Lucco, Anderson, and Graham [27]. 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 for CISC architectures by collecting an explicit list of legal jump targets. The best example of such a system is Small and Seltzer's MiSFIT [25], 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 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. Besides the trustworthiness problems of C compilers related to their complexity and weak specification (as exemplified by the attack against MiSFIT shown in Figure 1), this approach also requires something like a public-key certificate infrastructure for code producers, introducing problems of reputation to an otherwise objective question of code behavior.

Erlingsson and Schneider's SASI tool for the x86 [10] 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 [16] also generates a table of legal jump destinations, but the indices into the table are assigned sequentially at translation time, so there is no danger of collision.

The Omniware virtual machine [3], 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 [28] 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 9 was the first machine-checked or completely formalized soundness proof for an SFI technique or implementation. Necula and Lee [20] proved the soundness of SFI as applied to particular programs, but not in general, and only in the context of simple packet filters. In work concurrent with ours, Abadi et al. ([2], see Section 10.3 for discussion) give a human-readable prose proof for the safety of a model of their CFI system, which is similar to SFI. In work subsequent to our proof (first described in [18]), Winwood and Chakravarty developed a machine-checked safety proof in Isabelle/HOL for a model of an SFI-like rewriting technique applicable to RISC architectures [29]. To avoid having to move instructions, their approach overwrites indirect jump instructions with direct jumps of the same size to a trusted dispatcher. Unfortunately, this puts a 2MB limit on the size of binaries to which their technique is applicable: for instance, they were able to rewrite only a subset of the SPECint2000 suite.

10.2 Isolation and preventing subversion

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 cannot take 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. Examples of such designs include the device driver and network server isolation techniques discussed in Section 1, and the untrusted VXA decompressors of Section 8.

Incidentally, SFI subsumes some mechanisms that have been suggested as measures to make program subversion more difficult. For instance, PittSFIeld prohibits execution of code on the stack and reduces the number of possible targets of an overwritten function pointer. However, these side-effects should not be confused with the intended isolation policy. SFI does not provide general protection against attacks on the untrusted code; it simply contains those attacks within the component.

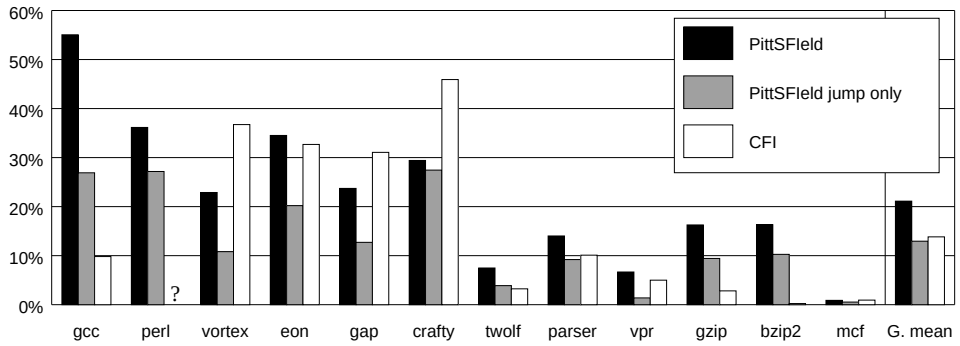

Figure 9: Runtime overheads for PittSFIeld in the default mode (black bars), PittSFIeld in jump-only mode (gray bars), and CFI (white bars) for the SPECint2000 benchmarks. PittSFIeld results are the same as those in Figure 4, but not broken down by cause. CFI results are taken from Figure 4 of [1], which does not include results for Perl. Because these were separate experiments with other variables not held constant, care should be used in comparing them directly.

10.3 CFI

In concurrent work [1], 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 CFI 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 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. Figure 9 compares the overheads reported in [1] with those for PittSFIeld from Figure 4. Because a different C compiler, library, and hardware were used, caution should be used in directly comparing the PittSFIeld and CFI results, but overall the average overheads of the tools can be seen to be comparable. The benchmark labelled "?", 253.perlbmk, was omitted from [1] because of last-minute implementation difficulties [9], and is excluded from the CFI average.

Like PittSFIeld, CFI performs a separate verification to enforce proper rewriting at load time, so the compiler and binary rewriting infrastructure need not be trusted. The CFI authors have written a human-checked proof [2] 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 CFI technique is affected in subtle ways by the x86 instruction encoding (for instance, the possibility that the immediate value used in the comparison at a jump site might be itself interpreted as a safe jump target tag.)

10.4 Static C safety mechanisms

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 [6]. 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.

10.5 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 [22] 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 simplified 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 [24]; 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 [15], 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. The VX32 system described in Section 8 also falls into this category. A disadvantage of dynamic techniques is that they are inherently somewhat complex and difficult to reason about, relative to a comparable static translation.

10.6 Low-level type safety

Recent research on verifiable low-level program representations has concentrated most strongly on static invariants, such as type systems. For instance, typed assembly language [19] 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 [5]. 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 [21] represents a more general framework for software to certify its own trustworthiness. Most work on PCC has focused on type-like safety properties, but under the banner of foundational PCC [4], 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. This approach seems to carry the promise, not yet realized, of allowing any safe rewriting to certify its safety properties to a code consumer. For instance, one could imagine using the lemmas from the proof of Section 9 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.

11 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 and is scalable to large and complex applications. The performance overhead of the technique, as measured on both standard compute-intensive benchmarks and a realistic data compression application, is relatively low. Though some related techniques have lower runtime overheads, and others can offer additional security guarantees, SFI's combination of simplicity and performance is a good match for many uses.

Acknowledgements

Bryan Ford provided us with the VXA infrastructure used in the case study of Section 8, and Mihai Budiu and Úlfar Erlingsson provided results for Figure 9 and answered other questions about CFI. Members of the MIT PDOS and PAG groups, and the Harvard programming languages and compilers groups, provided a number of helpful suggestions. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, VA, USA, November 7–11, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *ICFEM 2005, Proceedings of the 7th International Conference on Formal Engineering Methods*, pages 111–124, Manchester, UK, November 1–4, 2005.

[3] 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*, pages 127–136, Philadelphia, PA, USA, May 21–24, 1996.

[4] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–257, Boston, MA, USA, June 16–19, 2001.

[5] 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*, pages 232–244, San Diego, CA, USA, June 9–11, 2003.

[6] 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 *7th USENIX Security Symposium*, pages 63–78, Austin, TX, USA, January 28–29, 1998.

[7] Peter Deutsch and Charles A. Grant. A flexible measurement tool for software systems. In *Information Processing 71: Proceedings of IFIP Congress 71*, pages 320–326, Ljubljana, Yugoslavia, August 23–28, 1971.

[8] Daniel C. DuVarney, Sandeep Bhatkar, and V.N. Venkatakrishnan. SELF: a transparent security extension for ELF binaries. In *Proceedings of the 2003 New Security Paradigms Workshop*, pages 29–38, Ascona, Switzerland, August 18–21, 2003.

[9] Úlfar Erlingsson. Personal communication, May 2006.

[10] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, pages 87–95, Caledon Hills, ON, Canada, September 22–24, 1999.

[11] Bryan Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX Conference on File and Storage Technologies*, pages 295–308, San Francisco, CA, USA, December 14–16, 2005.

[12] 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.

[13] 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.

[14] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 273–284, San Antonio, TX, USA, June 12–14, 2003.

[15] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, USA, August 7–9, 2002.

[16] Fei Lu. C Plus J software architecture. Undergraduate thesis, Shanghai Jiaotong University, June 2000. English summary at `http://www.cs.jhu.edu/~flu/cpj/CPJ_guide.htm`.

[17] Stephen McCamant. A machine-checked safety proof for a CISC-compatible SFI technique. Technical Report 2006-035, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, May 2006.

[18] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report 2005-030, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, May 2005. (also MIT LCS TR #988).

[19] 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 *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1, 1999.

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX 2nd Symposium on OS Design and Implementation*, pages 229–243, Seattle, WA, USA, October 29–31, 1996.

[21] 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, June 17–19 1998.

[22] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification*, Boulder, CO, USA, July 13, 2003.

[23] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, pages 231–242, Washington, DC, USA, August 6–8, 2003.

[24] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 2002 Annual Computer Security Applications Conference*, pages 209–218, Las Vegas, NV, USA, December 9–13, 2002.

[25] Christopher Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 174–184, Portland, OR, USA, June 16–20 1997.

[26] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, Bolton Landing, NY, USA, October 19–22, 2003.

[27] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 5–8, 1993.

[28] 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.

[29] Simon Winwood and Manuel M. T. Chakravarty. Secure untrusted binaries - provably! In *Third International Workshop on Formal Aspects in Security and Trust*, pages 171–186, Newcastle upon Tyne, UK, July 18-19, 2005.