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.

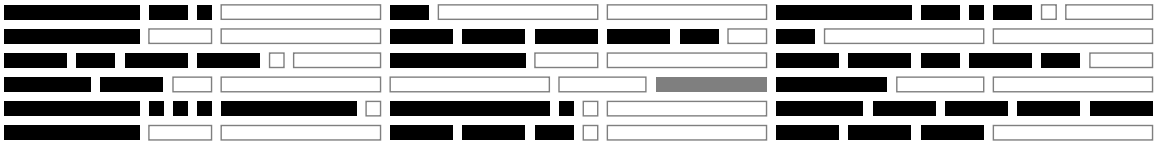

Figure 1: Illustration of the instruction alignment enforced by our technique. Black filled rectangles represent instructions of various lengths present in the original program. Gray outline rectangles represent added no-op instructions. ¡Instructions are not packed as tightly as possible into chunks 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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.