Portable Software Fault Isolation

Joshua A. Kroll
Computer Science Department
Princeton University
Princeton, NJ
kroll@cs.princeton.edu

Gordon Stewart
Computer Science Department
Princeton University
Princeton, NJ
jsseven@cs.princeton.edu

Andrew W. Appel
Computer Science Department
Princeton University
Princeton, NJ
appel@princeton.edu

***Abstract*—We present a new technique for *architecture portable* software fault isolation (SFI), together with a prototype implementation in the Coq proof assistant. Unlike traditional SFI, which relies on analysis of assembly-level programs, we analyze and rewrite programs in a compiler intermediate language, the Cminor language of the CompCert C compiler. But like traditional SFI, the compiler remains outside of the trusted computing base. By composing our program transformer with the verified back-end of CompCert and leveraging CompCert's formally proved preservation of the behavior of safe programs, we can obtain binary modules that satisfy the SFI memory safety policy for any of CompCert's supported architectures (currently: PowerPC, ARM, and x86-32). This allows the same SFI analysis to be used across multiple architectures, greatly simplifying the most difficult part of deploying trustworthy SFI systems.**

I. Introduction

Recently, there has been significant interest in systems that use *Software Fault Isolation* (SFI) to couple near native performance with strong isolation. SFI is a memory safety technique applicable to arbitrary programs and first proposed by Wahbe et al. [1]. SFI limits isolated code modules to only a portion of their address space, making it possible to load untrusted modules in the same address space as trusted code. SFI isolates programs by rewriting them to enforce policy (and verifying that binaries have a specific safe form), rather than proving safety either of the program statically or of particular executions.

For example, SFI has been used to provide high performance, securely sandboxed client side web applications [2]; to provide integrity for kernel modules [3], [4]; and to extend the Java Security Manager across the Java Native Interface [5]. SFI is a natural choice, given its demonstrated low overhead [6], minimal programmer annotation load and language independence [7], and minimal trusted computing base (TCB) [8]. The original SFI system was designed to support safe (i.e. non-crashing) extensional functionality for a database engine [1].

However, SFI techniques are very tightly bound to the architecture to which they are applied, and so these applications suffer from a lack of portability: SFI modules must be specially compiled and verified for each architecture on which they are to be deployed [1]. And this means that the trusted SFI verifier must be completely rewritten for every targeted architecture [9], adding to the size of the TCB. Further, while the trusted verifier is generally quite small, its interaction with the architecture is subtle and has led to security-critical bugs even in carefully evaluated systems [10].

We propose a different approach, which simultaneously improves security and portability: instead of doing SFI translation and verification at the level of assembly code, we perform SFI on a compiler intermediate language, Cminor, rewriting programs to guarantee safety and policy compliance. The invariant we establish by rewriting the code allows us to pass the rewritten program through the back end of CompCert [11], [12], a certified-correct compiler, to obtain a safe, policy-compliant executable binary, leveraging the safety- and semantics-preserving properties of CompCert. Because this technique allows us to perform our security analysis at a level which is architecture-independent, the resulting analysis is portable to any of the architectures targeted by CompCert, currently PowerPC, ARM, and x86. We prove that our method is correct, producing only security-compliant programs, by building a prototype implementation in Gallina, the specification language of the automated proof assistant Coq [13], [14], and proving that it implements our security policy soundly. In turn, we use Coq's extraction facility to turn our Gallina functions into OCaml code to compile and isolate real programs.

Our system provides for a small trusted computing base: we have no need for a separate trusted SFI verifier, as in traditional SFI [1], [8]. Instead, we can guarantee that the output of our transformation is safe with respect to the Cminor operational semantics and thus that CompCert will produce assembly language that has the same (safe) observable behavior. While there has been previous work on formally verified SFI systems (e.g. [15]–[19]), in each case the SFI and its formal verification are tightly coupled to the choice of instruction-set architecture.

Specifically, our analysis takes in any program in our chosen intermediate language Cminor, and produces a transformed program that is *safe* in the sense that it is not

stuck[1] with respect to the Cminor operational semantics provided by CompCert. The transformed program satisfies two properties:

Program safety/SFI The transformed program is guaranteed to execute safely with respect to the standard SFI memory safety policy: no reads or writes are done outside predefined, pre-allocated regions.

Correctness Every safe execution of the input program corresponds exactly (i.e. one-to-one) to a single execution of the transformed program with the same observable behavior. Programs which are not safe may be transformed to have arbitrary (but safe) behavior. This correspondence property follows easily by inspection for each of our portable SFI transformations but has not yet been proved in Coq.

This approach differs from classical static analysis: we establish the dynamic safety of *any* execution of *any* transformed program. This is the defining feature of SFI: any program can be transformed into a safe program even without determining whether the input program was in fact safe to begin with. Because CompCert is guaranteed to compile safe programs correctly, this scheme provides the same model of assurance as SFI but retains architecture portability and offers a small trusted computing base.

What is particularly notable about this analysis is that it is possible to isolate the security properties we need at the level of an intermediate representation. Previous SFI systems have been described in terms of their action on concrete addresses and have relied for correctness on the ability of the SFI compiler and verifier to perform arithmetic on concrete representations of addresses as numbers (for example, by bitwise `and` and `or`). However, an optimizing compiler must treat addresses abstractly in order to perform program transformations—such as spilling/reloading and stack frame allocation—that rearrange a program's memory layout. Indeed, the abstract nature of the CompCert memory model (described in Section III) is central to CompCert's proof of correctness. Thus, we cannot rely on numerical properties of addresses in our reasoning. One of our major contributions, then, is a demonstration of how to address this gap: we present an abstract SFI enforcement mechanism and demonstrate how to make that mechanism concrete while preserving its essential properties. In order for our analysis to be valid, we must carefully examine how assumptions in the abstract CompCert memory model can be realized faithfully in the memory of a real machine. This is described in more detail in Section VI.

An alternative proposal to address the portable distribution of code meant for an SFI system is to distribute bytecode for an intermediate language such as LLVM (as with Portable Native Client [20]). The bytecode can be compiled by an untrusted compiler back-end on the client and then verified by an SFI verifier for that client's architecture. This approach solves operational concerns surrounding application portability (e.g. that applications which have to be distributed on a per-architecture basis will become fragmented between platforms over time). But it does not address the problem of needing to re-engineer and re-verify the trusted computing base for every architecture.

Additionally, such an approach requires exposing a compiler back-end, software typically developed to run in a friendly environment, to arbitrary input. Modern optimizing compiler back-ends are large and complicated artifacts not typically built under the assumption that input may be chosen maliciously to cause unintended side-effects. One recent study of widely used compilers found several severe bugs including incorrect code generation and crashes [21]. CompCert was the only compiler that did not exhibit bugs when subjected to arbitrary input; no issues were found in CompCert's proved-correct components. Because our analysis accepts arbitrary abstract syntax in one of CompCert's intermediate languages, it is no more vulnerable to bugs caused by malformed input than CompCert as a whole.

Cminor and LLVM are in fact quite similar intermediate representations. Both are sufficiently high-level to be architecture independent. Both are sufficiently low-level that a variety of source languages can be compiled to either one. Armed with the system we present here, it would be possible to build a drop-in replacement for Portable NaCl [20] just by building a verified translation from LLVM to Cminor. Indeed, there have been recent efforts to formalize the semantics of LLVM in Coq [22], making this goal all the more realistic.

The main contributions of this work are:

- A method for software fault isolation (SFI) that is portable across multiple architectures.
- A prototype implementation of our method together with a mechanized specification and a machine-checked proof that guarantees the prototype's soundness for memory operations at the Cminor level. We also describe how to prove additionally that our prototype does not change the behavior of already-safe programs. We have not yet completed a machine-checked proof of end-to-end security of assembly language programs produced by composing our prototype with CompCert.

We begin by discussing traditional SFI methods in Section II. We discuss our approach as a counterpoint in Section III. In Section IV, we describe our transformation

[1]The *operational semantics* of a programming language define the rules for transitions between states of a model abstract machine. If, at some state, the execution of the program would cause the machine to transition according to these rules to another, well defined state, we say that the program takes a *step*. A program which can always take a step (or which is safely halted) is said to be *safe*. That is, executing the program from its initial state never yields a machine state that is incompatible with all rules in the operational semantics. If a program reaches a state where it cannot take a step, we say it is *stuck*. Note that this does not mean that the program stops executing - it just means our model no longer explains the execution behavior.

in detail. In Section V, we show how to prove the transformation correct and how our specification ties in to the correctness theorem in CompCert to provide both safety and isolation. In Section VI, we describe our model of memory and how we relate our abstract address analysis to a flattened address space. In Section VII, we describe early benchmarks of programs compiled with our techniques and expand on our early experiences with the system. Finally, Section IX presents related ideas from the literature and Section X contains our conclusions.

II. Traditional SFI

Here, we explain the goals and operation of classical SFI methods, which typically have three essential components: a *rewriter*, a *verifier*, and a *runtime system*.

In traditional SFI, sandboxed assembly language is rewritten from the originally generated assembly language [8] or generated by a specialized compiler [1], [2]. Code is then passed through a trusted assembler[2] to create a *sandboxed binary* in machine language. A trusted *verifier* and *runtime system* ingest this binary, disassemble it, and verify that it satisfies certain structural invariants, which imply a two-pronged policy over dynamic executions:

Reliable Execution Only machine instructions that have been disassembled and analyzed will ever be executed. All of these instructions are found in memory in a region of addresses bounded by $[\texttt{code\_lo}, \texttt{code\_hi})$.

Memory Safety All writes to memory (and in some implementations also reads from memory) will occur only to the region bounded by the addresses $[\texttt{data\_lo}, \texttt{data\_hi})$.

The code and data bounds must be chosen at compile-time and must be known to the compiler and verifier. Typically, the SFI region has a base address that, taken as a number, is evenly divisible by some power of two and also has a size equal to the same power of two, allowing extremely efficient computation of in-bounds addresses using bit-level arithmetic [8].[3] We will assume these conditions on the SFI regions for the rest of this paper.

The goal of the SFI code generator takes arbitrary programs and make them verifiably policy-compliant. The verifier uses the special structure of sandboxed binaries to trade soundness for completeness; any program accepted by the verifier is guaranteed to be policy-compliant, but the verifier will not accept all policy-compliant programs. Conversely, programs which "go wrong" by violating the SFI policy will be rewritten to have different (safe) behavior or else will be rejected by the verifier.

The SFI rewriter must check static references to memory to ensure that they are in-bounds and reject or modify unsafe programs. But many memory accesses occur through dynamic constructs such as fixed offsets from unknown pointers. It is not possible to know statically whether all such accesses will be policy-compliant. For these, including all memory accesses through a CPU register, the rewriter inserts *sandboxing instructions*: extra CPU instructions which modify the address held in the register or pointer, fixing the high order bits of the address to a specific value, the *SFI mask*. Note that if the address was in-bounds, the sandboxing instructions have no effect. If the address was out-of-bounds, a new, possibly unknown (but definitely in-bounds) address is substituted. In this way, the rewriter need not determine whether a particular memory access is safe; all sandboxed accesses are safe by construction.

A program with arbitrary control flow could try to jump into the middle of an instruction as decoded by the verifier, yielding a different stream of instructions than that checked for safety. Thus, all control transfers must target a special set of aligned addresses. The difficult case here are dynamic control transfers, such as indirect jumps, calls via function pointers and returns to addresses held on the stack, which are handled like dynamic memory accesses: by the insertion of sandboxing instructions before an indirect call, jump, or return.

Finally, it is important that the sandboxing checks strictly dominate (in the control flow graph) the operations they are meant to protect. Otherwise, a program could prepare an invalid value in a particular CPU register and then jump to an operation in which that unsafe value would be used. This is an under-appreciated, subtle aspect of the structural invariants imposed by an SFI rewriter: it is mentioned briefly in the explanation of verifier invariants in Pittsfield [8] but is missing from the provided implementation; Native Client [2] does not describe these invariants, but does implement them correctly. The fact that this observation was deficient from the discussion and implementation of SFI systems until recently underscores the need for formal verification. Later work (e.g. XFI [23]) exploits dominator analysis of the control flow graph to avoid redundant checks and reduce overhead.

SFI is typically implemented as a translation validation: the rewriter need not be part of the TCB. Instead, trust is generally placed in a separate verifier that operates on rewritten code. However, in some systems (including ours) no verifier is necessary; the SFI code generator generates policy-compliant code. In any case, as discussed below, SFI requires a trusted runtime (at minimum, to load the isolated code correctly).

[2]The assembler need not be trusted, but at least one of the assembler or the disassembler used by the verifier must be. For simplicity, for the rest of the paper, we will refer to the problem of converting (in either direction) between assembly and machine code in a trustworthy way as the problem of requiring a trusted assembler.

[3]These details are inessential but are common features of prior implementations [1], [2], [7]–[9]. For example, the dynamic enforcement of pointer values could be with respect to any bounds, not just power-of-two aligned bounds. And rather than explicitly modifying bad pointer values, the inlined reference monitor code could simply redirect control to a trusted error handler.

The structural invariants described above can be checked by a verifier that is just a finite-state machine. Typically, the verifier operates at load time and ensures that the loaded binary meets the above requirements. The verifier can be implemented very simply in a few hundred lines of code. For such an SFI system to be secure, only the verifier and the runtime system (including the loader) need to be trusted. A small TCB is one of the things that makes SFI attractive as a mechanism for module isolation. Compared with techniques such as full-scale reference monitors [24, Chapter 8], interpreted languages [25], and process-level hardware-enforced isolation [26], SFI requires a much smaller TCB. Still, even in carefully hardened systems, mistakes in the verifier do occur; modern architectures are complicated and supporting a large subset of the instructions in any ISA leads to complexity that is difficult to manage successfully.[4]

Finally, most SFI implementations include runtime systems that provide services such as inter-module communications and RPC, limited access to system calls, or calls into trusted, unsandboxed libraries. These runtime systems are also necessarily part of the trusted computing base. We will not discuss such elements, as they are highly variable.

III. Portable SFI

Traditional SFI requires multiple trusted verifiers, one for each target architecture. Portable SFI instead defines a *single* SFI analysis, at the level of an architecture-independent compiler intermediate language, CompCert's Cminor, and relies on the compiler's correctness theorem to prove security of the generated assembly programs.

At the Cminor level, CompCert treats memory abstractly, not as a collection of addressable bytes. This is highly desirable: CompCert aims to be an optimizing compiler, and many of the phases after ours will apply optimizations that change the memory layout of the program significantly. An abstract representation of memory is necessary to facilitate the reasoning in these later phases.

Thus, memory locations are described by a block number and an offset. While infinitely many blocks may be created dynamically, each block has a finite size assigned to it when it is allocated. Our strategy is to treat the entire SFI data region as a single block and to map all potentially unsafe accesses to memory (all accesses through pointers visible at the Cminor level) into accesses only into this block. As in traditional SFI, we aim not to *check* that this property holds of particular pointers but rather to *ensure* this property through the use of special *mask* operations, as we describe below.

A. Overview

Figure 1 gives a high-level overview of the two major phases of the portable SFI (PSFI) process. In phase one

[4]See [27] for a discussion of one such bug in Native Client.

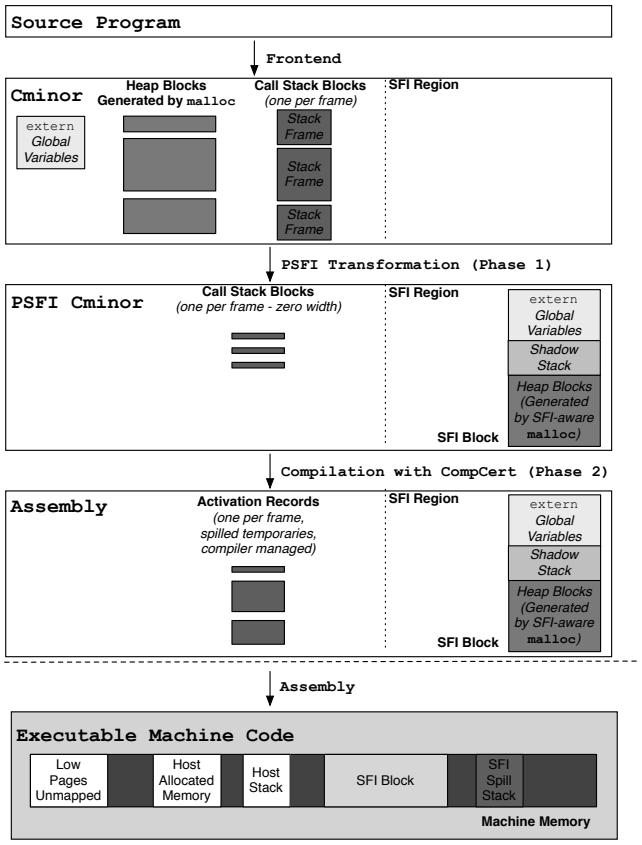

Figure 1. Our Portable SFI system, including a diagram of the abstract memory layout before SFI, after the SFI transformation, after both the SFI transformation and compilation with the CompCert back end, and finally after assembly in a one-dimensional, finite machine memory.

(labeled "PSFI transformation" in the figure), the PSFI compiler rewrites the input Cminor program to a new Cminor program in which all loads and stores are to the safe SFI region (as fixed at load time). The SFI transformations, which we detail in Section IV-E, include operations like relocating address-taken function local variables and `malloc`s to the SFI region. (Locals which are not address-taken are treated as temporaries in Cminor, and therefore are independent from memory.) In the figure, we depict the stack reallocation operation as the movement of the stack frames on the left-hand side of the Cminor box to a portion of the SFI region we call the "shadow stack" on the right-hand side of the box labeled PSFI Cminor. In a similar way, we relocate heap blocks generated by calls to `malloc` in the original Cminor program to heap blocks in the SFI region. This is accomplished rewriting the original program to call an SFI-aware version of `malloc`.

In phase two (labeled "Compilation with CompCert" in the figure), PSFI compiles the rewritten Cminor program to PowerPC, x86, or ARM using stock CompCert. CompCert's safety and semantics preservation theorems make it possible to show that the Cminor programs produced by stage one result in compiled assembly programs satisfying the following

SFI security invariant:

Definition (SFI Security)**.** *The output assembly program loads and stores only in the allowed SFI region, as fixed at load time, or in spill locations in stack frames introduced by CompCert (containing,* e.g.*, spilled temporaries and function return addresses).*

Informally, SFI security follows from the following four observations:

1) Stage one produces safe Cminor programs.

2) Cminor programs rewritten by stage one do not allocate memory outside the SFI region: PSFI transforms calls to `malloc` in the source program to `malloc`s within the SFI region. Address-taken local variables are relocated to the SFI region as well. Cminor local variables that are not address-taken are modeled in a separate "temporaries" environment independent from memory.

3) In the CompCert languages, there are two ways to allocate memory: Calling observable external functions such as `malloc` or pushing activation records onto the stack. (CompCert does not consider the latter an observable event.)

4) CompCert's semantics preservation theorem means the compiler does not introduce observable calls to external functions such as `malloc` that were not present in the source program. Thus by 2) and 3), the target program at most allocates memory for activation records or at external calls to the SFI-aware `malloc`. By 1) and CompCert's safety preservation theorem, we get that the target program *could not* have read or written to locations outside the SFI region or stack frames introduced by the compiler, since otherwise it would have become stuck.

B. Advantages of PSFI over traditional approaches

By piggy-backing on CompCert's compiler correctness theorem, we avoid the need to define and trust an SFI verifier for each target architecture. This is a significant gain; the RockSalt project [17] demonstrated that removing the verifier for even a single target architecture like x86 from the TCB of an SFI tool requires significant effort, requiring about 15,000 lines of machine checked specification and proof. One must first define semantics for a realistic subset of x86 (enough to serve as a target for multiple compilers and for hand-written assembly), then verify the verifier with respect to this semantics. PSFI sidesteps the "proliferating verifiers" problem by defining a proved-correct SFI analysis, then compiling with a proved-correct compiler.

Because our SFI compiler produces verified assembly code, no separate verification step is required. In effect, we have used the *certified* compiler CompCert to build a *certifying* (in the sense of proof carrying code [28][5]) SFI compiler. The resulting program can be assembled and executed without a heavyweight runtime because its safety is *certified* by the compilation process.

Performing the security analysis at a higher level in the compiler allows for the easy analysis of more interesting and complicated properties than the standard SFI invariants. Further, the optimizing phases of the compiler have an opportunity to act on the sandboxed code rather than the sandboxing transformation acting on the optimized code. This has the potential to make the sandboxed code more efficient (for example, redundant or unnecessary sandboxing operations can potentially be eliminated) and also makes the sandboxing transformation simpler (since it does not need to know about idioms generated by particular optimizations). Indeed, the fact that we do not need to worry about the security of most types of indirect control flow (with the exception of calls through function pointers) significantly reduces the implementation complexity of our system relative to competing approaches.

In the next section, we describe the SFI transformation and the parts of CompCert on which it relies: the intermediate language Cminor, the CompCert memory model, and the certified back end of CompCert that ties the system together.

IV. Technical Approach

A. CompCert and Compiler Correctness

CompCert [12] is an industrial-strength compiler supporting nearly all of the ISO C 90 standard (a.k.a. ANSI C). The compiler is built using the Coq proof assistant and has a machine-checked proof that the code it generates is observationally equivalent to the compiled source program. This is the key to our method: because the compiler preserves semantics, invariants established by analysis and rewriting at the Cminor level are true at lower levels.

Accordingly, we can enforce the SFI security specification of Section III-A by rewriting programs as they are compiled. Of course, for the specification to be preserved, we must also guarantee that the program we analyze does not get stuck. So we also rewrite programs so that they are safe with respect to the Cminor operational semantics. Our SFI transformation T is implemented as a series of seven new compilation phases in CompCert that convert an arbitrary Cminor source program S to a Cminor program $T(S)$, such that $T(S)$ is safe and policy-compliant (in the sense of Section III-A). Safety of $T(S)$ ensures that CompCert will produce an assembly-language program which has the same observable behavior as $T(S)$, meaning that the compiled program C is also policy-compliant.

B. The CompCert Memory Model

All of the CompCert intermediate languages, including Cminor and assembly, share a common memory model [29], [30]. Memory states in this model are collections of memory blocks of finite size, each with an integer block number $b \in \mathbb{Z}$. Blocks are assigned a size at allocation time, and are

[5]The *certificate* in this case would be the correctness proof of our SFI transformation, coupled with the correctness proof of CompCert itself.

by construction separate. Byte-level addresses are referenced via an offset δ, which must be within the block's bounds. Pointers are constructed by specifying a block number b and an offset δ, so an address is written (b, δ). In the version of CompCert we target in this work (1.11), functions are modeled as code blocks occurring at negative block addresses, meaning it is possible to have function pointers in all CompCert intermediate languages.

It is perhaps surprising that, in such an abstract memory model, we can express a technique such as software fault isolation, which is usually thought of in terms of the very concrete expression of pointers as patterns of bits. Certainly, it is not possible in CompCert's memory model to understand where a pointer will end up in the concrete byte-addressed memory array of the real machine on which the program will eventually run. However, because memory blocks in CompCert are separate by construction, we can reason about interference properties of pointers. As we will show, this is sufficient for our purposes.

C. Cminor, CompCert, and Infrastructure

Cminor is a simple imperative language designed as a compiler intermediate language for CompCert. It is comparable to a stripped-down, typeless variant of C. Like high-level languages, it is processor-independent (indeed, it is the lowest level processor-independent intermediate representation in the CompCert stack). CompCert provides a small-step operational semantics for Cminor, which we adopt for this work. We chose Cminor because it is processor-independent yet also easy to target as an intermediate representation as it has few high-level constructs. Any lower-level intermediate stage in CompCert would require architecture-dependent analysis. Any higher-level intermediate stage would introduce source-language dependence. Cminor is used in other work as a natural target in the CompCert stack for compiling various source languages [31].

Temporaries in Cminor do not have rich type information. While the Cminor operational semantics specifies what values must be held in temporaries in order for expressions to be valid or for statements to step in the operational semantics, the temporaries themselves are not typed and Cminor is encountered in the compiler after type checking. Therefore, the first pass of our SFI transformer is a simple type inferencer for int/float temporaries, as we describe below. We use this type information in the PSFI transformation phases to rewrite potentially stuck expressions.

As in C, Cminor programs may make calls both to internal and to external functions. Internal functions are those that are defined in the current translation unit. External functions are only prototyped. In CompCert's trace model, calls to external functions generate observable events.

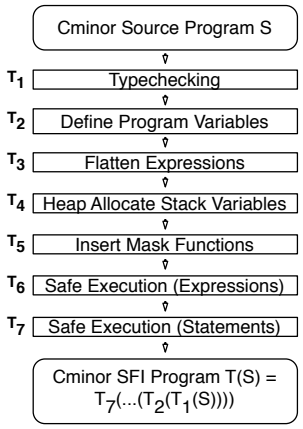

Figure 2. A diagram of the structure of our SFI transformation. Each box T_i shows what is rewritten at that stage.

D. The SFI Mask

Recall that our strategy for expressing SFI constraints at the Cminor level is to rewrite all memory accesses in the sandboxed program so that they reference a single distinguished CompCert memory block (the *SFI block*). The SFI block is allocated during program initialization, before control has passed to `main`. Its size is fixed to the size of the data region in the SFI policy.

To relocate loads and stores to the SFI block, we add to CompCert a new external function called *SFI mask*, or often just *mask*. At the CompCert level and for the purposes of proofs, we axiomatize this function to have the following properties:

- After a call to the SFI mask, the variable passed as an argument to the function contains a pointer to a valid address inside the SFI block.
- If the variable passed to SFI mask contained a pointer to a valid address inside the SFI block, then it contains the same pointer after the function returns.
- The SFI mask is idempotent.
- The SFI mask is a pure, total function.

We insert masks to restrict expressions that are used as addresses in potentially unsafe loads and stores. Once masked, pointers are guaranteed to point inside the SFI region.

E. The SFI Transformation

The SFI transformation comprises several passes. Each pass is implemented as a function on Cminor ASTs. Here we describe each pass briefly as well as the invariants the pass establishes. Figure 2 shows the order in which the passes are applied. Figure 1 shows the memory layout of the program in the Cminor source, in the transformed Cminor after all SFI passes, in CompCert's abstract assembly language, and finally in a flat machine memory model.

Because CompCert is an ANSI C compiler, the semantics, even at the Cminor intermediate representation, reflect the semantics of ANSI C. The details and restrictions described

below are not merely idiosyncratic: they make C programs whose executions are not "unpredictable."

1) Type Inference: In the first pass, we use a typechecker to infer the int-floatness of the local variables occurring in the program. Type inference will either fail with a type error, or will succeed with a symbol table mapping each identifier in the program to either `Tint` or `Tfloat`. The symbol table is then used in later passes to convey typing information.

By this point in the compiler, the input program has already been typechecked in C, but that information is not accessible to our transformations, so we have implemented our own typechecker for Cminor. This is necessary anyway, as our system supports any source program which can be targeted to Cminor, not just those written in C.

2) Define Program Variables: In the second phase, we use the type information to explicitly define (initialize) every local variable that occurs in the program. Although safe Cminor programs would never reference an undefined variable, this phase ensures that even *unsafe* programs, which are acceptable inputs to our SFI transformation, do not exhibit this particular unsafe behavior.

Safe programs, moreover, must define each program variable before use. Thus it is conservative to predefine each program variable with an arbitrary value of the appropriate type, as long as the new definitions occur before any existing definitions. The variable definition pass ensures that this is the case by inserting the new definitions as a block in the preamble of each function. For example, the following unsafe Cminor function (expressed in C syntax).

```
void f(void) {
  int c; double d;
  c = (int)(-1.*d);
}
```

would be transformed to

```
void f(void) {
  int c; double d;
  d = -0.;
  c = (int)(-1.*d);
}
```

in order to properly initialize `d` before first use.

3) Flatten: Because loads in Cminor may occur at nested positions within expressions, it is not possible to apply the *mask* operation directly to every loaded or stored address. Instead, we perform an intermediate "flattening" transformation that pulls loads to top-level, while ensuring that loads only occur to the right of assignment statements. To do so, we must introduce fresh local variables of the appropriate types to store intermediate values. However, because our SFI passes operate above register allocation and optimization in the compiler stack, these temporaries will often be optimized away by later compiler phases.

As an example of flattening, consider the following code block containing nested loads in the call to `printf`.

```
int i = 10;
int * p;

p = &i;
printf('%d\n', *p + *p);
```

After flattening, we get

```
int i = 10;
int * p;

p = &i;
a = *p;
b = *p;
printf('%d\n', a + b);
```

where `a` is a fresh temporary, and the expression `*p + *p` in the call to `printf` has been replaced by the new assignments `a = *p; b = *p` and the expression `a + b`.

4) Malloc Shadow Stack: In Cminor, there are three kinds of local variables. First, there are stack-allocated local variables which have their addresses taken. At the Cminor stage in CompCert, these have been turned into explicit loads and stores into the programmer-visible stack frame. Second, there are *temporaries* that do not have their addresses taken. Temporaries at the Cminor level may be allocated to registers or spilled later in compilation. Finally, there are user-invisible local variables such as the return address of each call frame.

Later in the compilation process, CompCert will spill temporaries and store return addresses into a user-invisible portion of the stack frame. A complete stack frame consists of a user-visible portion, made up of variables accessed by dereferencing user-visible pointers, surrounded by a user-invisible portion made up of spilled temporaries and user-invisible local variables. To enforce the integrity of the user-invisible stack, in this pass we move the user-visible portion into a *shadow stack* in the main SFI block.

To accomplish this, we `malloc` sufficient memory for all of the stack-allocated locals in each Cminor function and rewrite stack accesses to reference this new memory in the SFI block.[6] We must link at runtime with a memory allocator which is aware of the location and size of the SFI block and which only allocates safe memory.

Additionally, if a function has no stack-allocated local variables, we explicitly check for any stack accesses in the function and remove them. This is necessary because Cminor contains a special constant type for stack references, which is

[6]To preserve locality, this `malloc` may be a special shadow stack `malloc` which allocates safe memory on a shadow stack within the SFI region.

interpreted as an offset to the current stack block. This allows us to guarantee that, after this pass is complete, the program contains only heap accesses to memory, local temporaries, and global variables.

Concretely, consider the following Cminor program.

```
void main(void) {
  int i = 10;
  int * p = &i;

  *p = 11;
  *&i = 12;
  return;
}
```

Here `i` is allocated on the stack because it is addressed in the assignment `int * p = &i`. After the shadow stack transformation, we get the program

```
void main(void) {
  void * sb = malloc(4);
  *(sb + 0) = 10;
  int * p = sb + 0;

  *p = 11;
  *(sb + 0) = 12;
  free(sb);
  return;
}
```

in which a new stack frame has been allocated within the SFI region by a call to an SFI-aware `malloc` function. Each access of `i` in the original program has been replaced by a dereference of the address `sb + 0`, the top of the allocated stack frame. Here `sb` is a fresh temporary pointer variable. Before the transformed function returns, a call to an SFI-aware `free` is inserted to deallocate the stack frame.

The vast majority of functions in typical C programs do not contain *any* addressed local variables. In these cases, we can avoid the calls to `malloc` and `free` entirely along with the concomitant performance penalty.

5) Mask: We add the SFI mask to protect every operation that dereferences a pointer. Since the previous Flatten pass simplified expressions, it is easy to insert the mask function where required. We model the mask function as a compiler builtin function with a signature `Tint -> Tint`. As in the Flatten pass, we must add fresh temporaries in order to place the return value in the correct location. These temporaries are removed by optimization.

We must also ensure correct control flow at this stage. In Cminor, function pointers are the only real source of indirect control flow. Other control flow structures (such as conditional branches, switch statements, block-and-exit constructions, and jumps to labels) have well-defined semantics in Cminor. CompCert will generate correct, faithful assembly-level implementations which exhibit the same observable behavior as long as we guarantee that the constructions are semantically safe at the Cminor level. Thus, control flow (with the exception of indirect control flow through function pointers) is confined to the control flow graph visible at the Cminor level. For calls through function pointers to be semantically safe, they must point to a known function (functions pointers are represented as pointers to blocks with block numbers $b < 0$) and the known function must have a signature that matches the stated signature in the call node.

We thus introduce a new mask function to isolate function pointers. We require a portion of the text region to be set aside for special SFI-aligned memory blocks, one per distinct function signature. These new blocks may not overlap each other or the data region but may otherwise be located anywhere in memory. Inside these blocks, we place *trampolines*: small, fixed instruction sequences that implement calls to the appropriate function. Function pointers are allowed to point at this sequence, even if they are not allowed to point at the actual function they reference (which might be loaded at a different address). The new allowable regions are quite small: each allowable region needs only to be large enough to hold the trampolines for those functions with the given arity and signature that are called by function variables.

We must specify an implementation in assembly for the data mask function and for function pointer trampolines for each architecture we wish to target. The particular choice of sandboxing instructions is not important except for performance, so long as the implementation has the desired properties in the CompCert assembly semantics for the given architecture. For example, on x86, the mask instruction might be a single `and` instruction that fixes the high-order bits of pointers by clearing any that are set erroneously. Together with a loader that unmaps a memory block equal in size to the SFI data block at the bottom of the address space, this mask is sufficient to satisfy the SFI properties [8].

6) Reliable Execution: Finally, we rewrite the program so that it is guaranteed never to get "stuck" in the Cminor operational semantics. That is, executing the program from its initial state never yields a machine state that is incompatible with all rules in the operational semantics.

This is analogous to the code alignment constraints in traditional SFI where aligned chunks allow the verifier to guarantee that it observes the same sequence of instructions as will be executed. Here, we need not worry about rogue control flow—CompCert guarantees that it will create a program that satisfies (at least observably) the control flow constraints we can see at the Cminor level. However, in order to take advantage of CompCert's correctness theorem, we must give it a program that we know does not observably "go wrong", that is, a program which never makes a state transition not covered by the provided operational semantics.

First, we consider expressions, which must always evaluate to well-defined values. In order to guarantee expres-

sion evaluation, it is sometimes necessary to insert runtime checks (when subexpression values must satisfy certain constraints) and to substitute made-up values for poorly formed expressions. For example, if `a` and `b` have type `Tint`, the statement `c = a/b;` will only evaluate in the Cminor operational semantics if `b` $\neq 0$. So we replace it with the statement `c = (b != 0)? a / b : 0;` where we replace the evaluation of the expression with an invented value if it would not otherwise evaluate.

The constant value `0` chosen here is arbitrary. Recall that because we only guarantee to preserve the behavior of semantically safe (i.e. well defined) programs but require safe behavior of all output programs, we can sensibly invent safe (but not semantics-preserving) behavior for programs that are not themselves semantically safe. Such runtime checks are also necessary to resolve value-dependent execution semantics for pointer subtraction, modular arithmetic, bit-level shift operations, and pointer-integer comparison (in Cminor, a pointer may be compared sensibly to the integer 0, but not to any other integer).

Next, we turn our attention to statements. We ensure that all named function calls are safe with respect to the Cminor operational semantics by ensuring that the calls appropriately match the template of the named function. Function pointers must match the template of some declared function (we only require that a function pointer *can* be resolved, since we cannot know which function it actually points to).

F. Runtime

Our system requires a minimal runtime for each architecture that will be targeted. The primary role of the runtime is to lay out memory correctly. The loader must place the data section of the SFI executable in an appropriately-aligned location for the choice of implementation of data mask function (and may also have to protect pages around this location appropriately). Also, the loader must ensure that global (**`extern`**) variables are mapped into the SFI region, especially if those variables have their addresses taken. Additionally, the loader must install function pointer trampolines as described above (or load the functions themselves at aligned addresses). The loader may also need to protect certain regions of memory depending on the choice of implementation of the mask function: for example, the loader may need to unmap guard pages around the SFI block to account for pointer-relative addressing modes or to unmap pages at the bottom of the address space to allow for efficient masks that only clear (but do not set) erroneous bits in sandboxed pointers.

Finally, the runtime must include an architecture-appropriate `malloc` implementation which will return only buffers in the SFI region. We can provide such a `malloc` library simply by applying the PSFI transformation to a standard memory allocation library. Thus, the memory allocator is also verified and need not be part of the trusted base.

V. Verification and Trust

In this section, we describe the specification of the SFI system and outline the proof that our implementation meets this specification. There are two properties that define a security system as SFI: (1) The system must be able to transform any any input program so that it executes safely; and (2) the program transformer does not alter the behavior of safe programs. SFI systems typically achieve (1) by *modifying* input programs so that they are safe, without analyzing whether that behavior would have been safe without modification.

To define SFI security for Cminor programs, we first introduce the notion of OK addresses.

Definition (OK Address)**.** *An address (b, δ) is OK, where b is a CompCert memory block and δ is an offset into block b, when either*

- *b is the distinguished SFI block sb, as fixed at program load time, and $lo_{sb} \leq \delta < hi_{sb}$, where lo_{sb} and hi_{sb} are the low and high offsets, respectively, of the SFI region; or*
- *(b, δ) addresses a compiler-managed memory region such as the part of a stack frame used for spilling.*

We say a program is *SFI-secure* when it is (1) safe and (2) loads or stores only to OK addresses.

Now, we can define the desired properties of our system more formally

Property 1 (Program Safety and Security)**.** *Let S be a source Cminor program and let $T(S)$ be the program produced by the SFI transformer. Let C be the assembly program compiled by CompCert from $T(S)$. Then C is SFI-secure.*

Property 2 (Semantics Preservation)**.** *Let S be a safe Cminor source program. Let $T(S)$ be the Cminor program produced by running the SFI transformer on S. Then S and $T(S)$ are observationally equivalent.*[7]

Furthermore, by CompCert's correctness theorem, we have the following corollary.

Corollary (Semantics Equivalence of Compiled Code)**.** *Assembly programs C compiled from $T(S)$ have the same observable behavior as $T(S)$, and thus as S.*

To establish that our PSFI transformations, when composed with compilation by CompCert, produce assembly programs conforming to the SFI security policy, we first prove that the *Cminor* programs output by the program transformer described in Section IV satisfy SFI security. To do so, we establish formally the transformation invariants described in Section IV. For example, after the Masking pass, programs satisfy the following invariant:

[7]As in CompCert, programs are observationally equivalent when they produce equivalent event traces.

Lemma (Masking Invariant)**.** *Take source Cminor program S. Assume S satisfies the invariant established by the flattening pass (loads only at top-level, and only within an assignment statement). Let $T(S)$ be the program produced by the masking transformation. Then*

$$loaded\_ids(T(S)) \cup stored\_ids(T(S)) \subseteq masked\_ids(T(S))$$

In other words, assuming the flattening invariant holds initially of S, the set of masked identifiers in $T(S)$ is a superset of the set of identifiers loaded or stored in $T(S)$ (every identifier in $T(S)$ containing an address that is either loaded or stored has also been masked). This invariant, together with the fact that masks are inserted directly before the corresponding loads and stores, and a second invariant which ensures that loads and stores are only to (potentially fresh) temporaries, never of arbitrary expressions, establishes that every memory access in $T(S)$ is guarded correctly.

The formal statements of the invariants for other passes are not very interesting. For example, to formally express the flattening invariant used above, we first characterize what it means for an expression e to be load-free:

$$\begin{array}{l} load\_free\ (e : \mathsf{expr})\ = \\ \quad \mathsf{match}\ e\ \mathsf{with} \\ \qquad \mid \mathsf{Evar}\ i \Rightarrow \mathsf{True} \\ \qquad \mid \mathsf{Econst}\ c \Rightarrow \mathsf{True} \\ \qquad \mid \mathsf{Eunop}\ op\ e_1 \Rightarrow load\_free\ e_1 \\ \qquad \mid \mathsf{Eload}\ ch\ e_1 \Rightarrow \mathsf{False} \\ \qquad \mid \mathsf{Ebinop}\ op\ e_1\ e_2 \Rightarrow \\ \qquad\quad load\_free\ e_1 \wedge load\_free\ e_2 \\ \qquad \mid \mathsf{Econdition}\ e_1\ e_2\ e_3 \Rightarrow \\ \qquad\quad load\_free\ e_1 \wedge load\_free\ e_2 \wedge load\_free\ e_3 \end{array}$$

A statement s satisfies the flattening invariant (all loads in s are top-level assignments) when every expression in s (besides top-level loads themselves) is load-free:

$$\begin{array}{l} stmt\_fmap\ (P : \mathsf{expr} \to \mathsf{Prop})\ (s : \mathsf{stmt})\ = \\ \quad \mathsf{match}\ s\ \mathsf{with} \\ \qquad \mid \mathsf{Sassign}\ i\ (\mathsf{Eload}\ ch\ e) \Rightarrow P(e) \\ \qquad \mid \mathsf{Sassign}\ i\ e \Rightarrow P(e) \\ \qquad \mid \mathsf{Sloop}\ s \Rightarrow stmt\_fmap\ P\ s \\ \qquad \mid \cdots \\ \\ load\_free\_stmt\ =\ stmt\_fmap\ load\_free \end{array}$$

We lift this property to whole programs in a similar way.

A. Trusted Computing Base

Our system enjoys a very small TCB: it is not necessary to trust that the isolated program is bug free, because we rewrite it to be safe. It is not necessary to trust the SFI compiler because we have proved it sound. It is not even necessary to trust the proofs of our transformation, as they are checked by machine. One must only trust the statements of our soundness theorems above.

Some of the theorems above are in fact about completeness, not security: in reality, while the code *consumer* cares about security and the trusted base, as it is her trust in question, the code *producer* also needs assurance that the code has not been broken by the SFI system.

At runtime, only the loader need be trusted. Any additional runtime services which might be implemented (such as intermodule communication or trusted system call handlers) would, of course, also be trusted.

The mechanized proof infrastructure must be trusted. In Coq, a small kernel verifier checks proof objects against types generated from the theorem specifications. It has been carefully validated and is widely considered trustworthy. Additionally, in order to actually run our code on real programs, it is necessary to extract the Gallina code to executable OCaml. The extraction facility itself must be trusted, along with the OCaml runtime. The compiler, CompCert, need not be completely trusted: because it has a machine-checked proof of correctness, only the (much smaller) specification of that proof is in the TCB.

Finally, as in traditional SFI, it is necessary to trust the assembler to preserve the semantics of the assembly language program faithfully in the final, executable machine language representation (or to analyze those semantics properly during disassembly). Here, again, it would be possible to make our technique foundational by extending the semantic preservation proof in CompCert to machine language.

VI. Modeling Memory

In our model of the SFI transformation, we define mask as an external function that takes as argument a (possibly unsafe) address and returns an address into the SFI region as result. We model the SFI region as a distinguished CompCert block. Since the dynamic semantics of both Cminor and the CompCert assembly languages employ exactly the same memory model, we can specify uniformly the semantics of mask across these languages. Here we consider what happens when we expand our built-in mask function to inlined assembly instructions in a finite, flat, byte-addressable memory model. We justify this transformation step by showing that the block-level abstraction of mask is an adequate model of our implementation of mask as inlined assembly.

Claim (Adequacy of the Mask Specification)**.** *The inlined assembly implementation of* mask *refines, in the one-dimensional virtual address space allocated to a running process by the operating system, the specification of* mask *defined above on CompCert memories.*

This follows from the fact that CompCert blocks represent contiguous regions of memory. In particular, CompCert preserves the behavior of programs that perform pointer arithmetic within a block, but behavior is not preserved for pointer arithmetic across blocks.

In a real memory, the SFI region will be a power-of-two sized block with a base address that is evenly divisible by the same power of two. In this way, all addresses in the SFI region have the property that their high-order bits are fixed to a value known as the *tag* while choice of low-order bits is in-bounds. The inlined assembly implementation of mask must ensure that all pointers match the tag for the target region before they are dereferenced. This is usually accomplished by a two-instruction sequence: one `or` to set the relevant bits of the tag and one `and` to clear the others.

Memory in CompCert is infinite, in the sense that the internal `alloc` operation over memories never fails to generate a fresh memory block. This operator is used to create new stack frames, so this is equivalent to saying that it is always possible to create a new stack frame. And this is also true in the C language specification. Whatever techniques are used to prevent stack overflow in a given system must be applied here to prevent arbitrary growth of the program's stack, which cannot overlap with the SFI region.

VII. Evaluation

We evaluated our prototype on 18 single-threaded benchmarks included with the CompCert distribution, totaling over 15,700 lines of ANSI C according to the `sloccount` tool.

A. Evaluation on x86

In most cases, we observe that our compiler has a performance somewhere between `gcc -O0` and `gcc -O1`. We hypothesize that our prototype can benefit from significant optimization; for example, we should likely modify our system to use a `malloc` implementation that assigns memory in a block specified by and set up by a trusted runtime loader. Even still, across all benchmarks, we measure only a 24.7% average runtime overhead over unmodified CompCert. Our overhead ranges from a 0.5% speedup over unmodified CompCert for one benchmark, `fib` (we believe after investigation that this reflects luck in cache alignment—the only difference between the two programs in assembly is a single inlined call to our data mask) to a 100% slowdown on `floats` (which makes heavy use of memory comparisons and so requires many mask operations). Full results are reported in Figure 3, in which the geometric means of each benchmark are presented by compiler, normalized so that the results from unmodified CompCert always have a mean of 1.0. In this figure, the ratio of any two bars represents the overhead incurred by running a program compiled in one configuration over running the same program compiled in another. Unmodified CompCert and `gcc -O1` produce comparable-quality code for many benchmarks. Our benchmarks were performed on a dual-core Intel Core 2 Duo 2.33GHz machine with 2GB of memory running Linux 3.2.0. For comparison, we also compiled the benchmark suite with an unmodified version of CompCert 1.11 and with GCC 4.7.2 with flags `-O1`, `-O2`, `-O0`, and `-O3`.

B. Evaluation on ARM

Our benchmarks were performed on a Raspberry Pi, Model B, which uses a Broadcom BCM2835 System on a Chip (SoC), containing an ARM ARM1176JZFS processor running at 700 MHz with 512 MB of RAM (we used a configuration with 1024 MB of virtual memory, backed by a swap file on an SD card, although none of our benchmarks uses a substantial amount of memory. The Pi was running Linux 3.10.25 and was equipped with GCC 4.6.3.

We evaluated our prototype on 14 single-threaded benchmarks totaling about 10,500 lines of ANSI C. These were a subset of the microbenchmarks included with CompCert used to benchmark our prototype on x86.[8]

In most cases, we observe again that our compiler performs similarly to `gcc -O0`, often significantly better (close to `gcc -O1`). Our results likely would be greatly improved by using a standard "dedicated register" sandboxing scheme, as is standard for SFI systems on RISC architectures [1], [7]. Also, as in the x86 case, we could benefit significantly from careful further work on optimization to eliminate redundant mask operations or to improve cache performance. Across all benchmarks, we see a very reasonable mean overhead of 16.4% with a maximum overhead of 68.9% on the fannkuch microbenchmark. Two microbenchmarks (fib and integr) required no mask operations at all on ARM because the calling convention eliminates memory accesses related to arguments to `main` and so the masks are removed as dead code by CompCert. Full results for ARM are reported in Figure 4.

We regret that we were unable to procure test environments to measure the overhead of our techniques the PowerPC architecture.

VIII. Limitations

There are a few general limitations to our approach. First, as in [20], it is necessary either for the user of the system to compile an intermediate representation of the desired program or for some kind of trusted compile-and-sign service to generate verified binaries. Compiling a program, especially a large program, can add unacceptably to the start up latency. However, in our experience, the time required to compile a program with CompCert is comparable to other compilers and, while noticeable, is not unacceptable

[8]CompCert 1.11 unfortunately does not pass its own regression test suite on the Raspberry Pi for reasons we have not determined. We also left out of both sets three microbenchmarks that made heavy use of I/O (there are provided benchmarks for compression/decompression algorithms), since we determined that the execution environments were not usefully comparable.

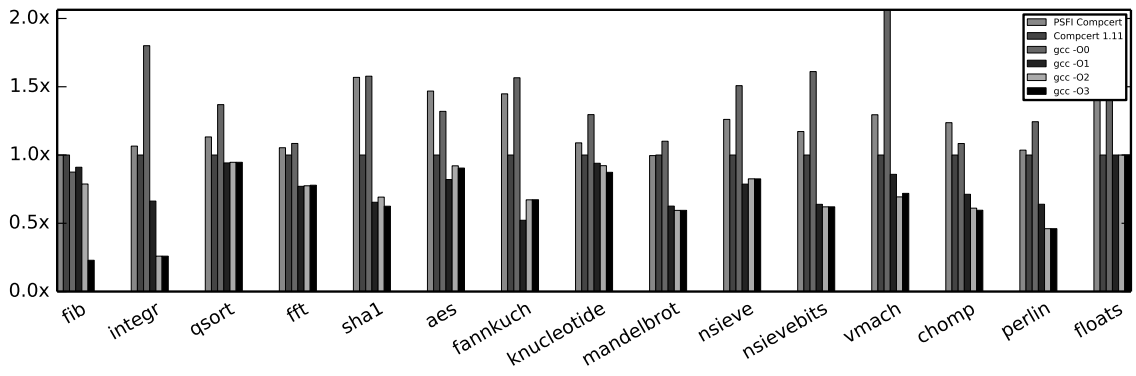

Figure 3. The geometric mean of many runs of each benchmark (grouped by compiler) in our suite $n = 100$ times on a desktop x86 machine, normalized so that the mean benchmark results for programs compiled with unmodified CompCert, Version 1.11 are always equal to 1.0. In this figure, the ratio of any two bars represents the runtime overhead of running programs compiled with one configuration over another. The average overhead across all benchmarks for our Portable SFI compiler was 24.7% over unmodified CompCert.

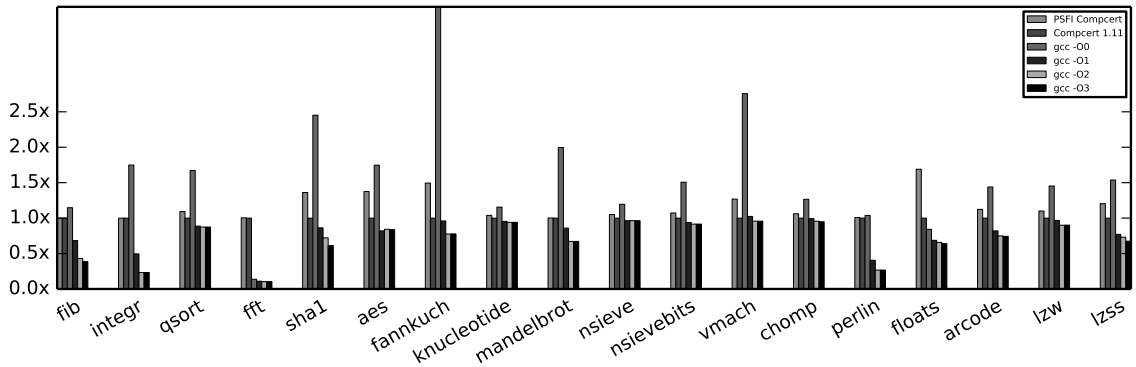

Figure 4. The geometric mean of many runs of each benchmark (grouped by compiler) in our suite $n = 100$ times on a Raspberry Pi, which contains an ARM SoC. Results are normalized so that the mean benchmark result for programs compiled with unmodified CompCert, Version 1.11 is always exactly 1.0. In this figure, the ratio of any two bars represents the runtime overhead of running programs compiled with one configuration over another. The average overhead across all benchmarks for our Portable SFI compiler was 16.4% over unmodified CompCert.

for interactive use.[9] Additionally, other deployment solutions could eliminate this problem: there could be a trusted code producer who signs correct binaries; the client could cache or remember a hash of previously compiled binaries to facilitate efficient re-use; or the system could be modified to include a client-side verifier as in traditional SFI.

Second, the CompCert correctness properties really only apply to single-threaded programs; CompCert provides no semantics for concurrent programs. Relatedly, CompCert does not have well-defined semantics for linking together multiple translation units. While we know that an SFI-isolated module will not interfere with other modules outside the SFI block, we are interested in a more complete model for composing multiple programs.

Our current prototype has several limitations, which we are actively working to address. None of these is fundamental. First, we do not yet rewrite all cases in which a program might get stuck in the Cminor operational semantics and so do not support certain critical C language features (function pointers and **`extern`** variables). We also have not built a custom runtime for our isolated binaries and so we use a modified SFI mask for measurement purposes (the mask is functionally a no-op with the same computational cost as the real mask function). These limitations require only straightforward engineering effort to overcome.

Status of the Mechanized Coq Development: While we have proved in Coq that our PSFI transformation pro-

[9] We found that compiling programs with CompCert was approximately 10% slower than `gcc -O3` and twice as slow as `gcc -O0` on a workload of 19 diverse C programs in the CompCert benchmark suite comprising about 10.8 kloc. Our benchmarks of CompCert-compiled programs show that CompCert produces code which is about as efficient as `gcc -O1`; compiling these benchmarks takes approximately 70% longer in CompCert than with `gcc -O1`.

duces Cminor programs satisfying SFI security as defined in Section III, we have not yet mechanically proved that 1) our transformation produces safe Cminor programs; 2) that our PSFI passes preserve the behavior of programs that were originally safe; or 3) the end-to-end theorem that SFI security theorem holds for assembly language programs produced by CompCert composed with our program transformer. We have paper-and-pencil proofs of these outstanding theorems, however.

IX. Related Work

SFI ([1]–[3], [6]–[9], [17], [32]) uses code-rewriting to inline a reference monitor and guarantee safe execution. The original SFI applied only to RISC-style architectures because of the requirement that only instructions which had been disassembled and analyzed by the verifier may be executed [1]. Much later work on PittSFIeld generalized the original SFI scheme to CISC architectures [8] and proved formally that the static invariants checked by their SFI verifier do in fact imply the desired dynamic policy [15].

The more recent Native Client (NaCl) [2] implements the PittSFIeld SFI techniques [8] but with the well-known optimization that sandboxed memory addresses are held in segment registers, eliminating the need for many dynamic checks [26], [33]. Native Client is targeted at web browsers and has been deployed as a prototype both as a browser extension and as a built-in security technology in major modern browsers. Later versions of Native Client have supported additional architectures [6] as well as dynamic languages via JIT compilation techniques [7].

More recent work joins SFI with other enforcement methods to provide fine-grained memory protections [34], enforce policies on native extensions for type-safe code run in a virtual machine [5], or enforce dynamically the contract specified at an API interface [4].

There has been much previous work on formally verified SFI systems (e.g. [15]–[19]), in each case the SFI and its formal verification are tightly coupled to the choice of instruction-set architecture. In particular, the RockSalt project [17] is notable as the first complete, executable, formally proved-correct SFI system. Like our system, RockSalt uses the Coq proof assistant for its formalism. However, unlike our system, RockSalt's formalization only covers a single instruction set architecture. Similarly, the ARMor project [18] provides a proved-correct SFI system for the ARM architecture.

Control-flow integrity (CFI) [16] and the related technique XFI [23] use a different kind of inline reference monitor to enforce a higher-level invariant that only designated code paths accepted by a verifier can be executed. This in turn allows more complicated policies to be enforced by static analysis. More recent work uses these techniques to eliminate redundant SFI sandboxing instructions, creating a faster hybrid enforcement system [35]. Further work (CCFIR) restructures CFI checks to vastly improve overhead [36].

Also in the vein of systems for isolating untrusted extensional modules is Xax, a browser plug-in which isolates untrusted modules in their own system-level address spaces and limits the availability of certain system calls, creating a so-called "picoprocess" abstraction [37]. Picoprocesses are, like SFI and CFI/XFI modules, required to call into a trusted security manager for access to system resources.

Proof-carrying code [28], [38] and typed assembly language [39] are other methods for providing executable binaries with certified safety properties. In proof-carrying code, each statement in a program is related to pre- and post-conditions in a program logic and a binary comes together with an encoded proof of these conditions. The proof is checked at load time, allowing a client to be sure that a particular policy will hold when the binary is executed. Foundational Proof-Carrying Code [40] eliminates the need to trust the verifier. SFI is very much like proof-carrying code, although the invariants are fixed ahead of time rather than generated and proved based on a particular program and policy. This allows the proof checking mechanism in SFI to be very simple. SFI methods, including our approach, differ from proof-carrying code in that programs are modified in a separate transformation to ensure compliance. In a sense, our SFI method is both certified correct and certifying of the programs given to it as input. Our approach also adds a completeness property: we can accept *any* Cminor program as input and execute it safely, not just those which have been proved correct.

Typed assembly language provides type-style guarantees for values acted on by assembly languages. Type annotations attached to values can be used by a type checker to certify particular invariants about the action of a program. SFI does not keep track of type information when examining a program for safety. Indeed, SFI abuses the type of addresses, modifying addresses as bit patterns for efficiency.

Finally, there are many variants of a research paradigm that is perhaps best called "safe C". See for example CCured [41], Cyclone [42], [43], and more recently CETS for temporal safety of pointers [44] and SoftBound for spatial safety of pointers [45]. These methods are concerned with providing high-level safety guarantees such as guarantees that a pointer does not move past the end of its allocation. SFI takes a coarser view of safety, providing safety not at the level of individual pointers but at the module level. Pointers in SFI-sandboxed code may in fact point to "improper" locations, so long as those locations are still within the data range for the module that owns the pointer. SFI does not provide a mechanism to prove the safety or correctness of a piece of code on its own. Rather, SFI provides inter-module confinement, guaranteeing that such faults in one module cannot affect anything outside that module.

X. Conclusions and Further Work

We present an architecture for software fault isolation that is portable across multiple instruction-set architectures and does not depend on details of the particular instruction set desired for the security of binary execution. We do this with the help of CompCert, building a certifying SFI compiler out of CompCert's certified back end, by performing the critical security analysis and program rewriting steps at the level of one of CompCert's intermediate languages, Cminor. Further, we prove the soundness of a prototype implementation.

This method has several advantages. First and foremost, it significantly reduces the implementation complexity of SFI systems stemming from architecture-specific concerns. Also, we have a smaller trusted computing than deployment models that require separate verifiers for each targeted architecture. Finally, because our analysis is done at a high level, it can easily be extended to provide more elaborate guarantees than a simple memory safety policy.

We stress that our analysis, while very different from traditional methods, is much more akin to SFI than it is to static analysis. What results from our rewriting is a *dynamic* policy over program executions—we are able to guarantee dynamic pointer safety through only local reasoning about pointer use. Pointer dereferencing is, in fact, only dynamically safe, since the safety is imparted by a call to the mask function, not as a property of the value of the pointer itself. Also, our analysis will guarantee the safe execution of *any* input Cminor program, not just those which can be effectively analyzed.

There are many potential future directions for this work. First and foremost, it would be a useful and straightforward engineering exercise to extend our prototype to a usable tool suitable for real-world deployment. But there are many future research directions as well. One would be to make use of higher-level analysis of the control flow graph and the relationship between modifications to and uses of pointers in order to eliminate redundant dynamic checks, similar to [16], [23], [35], [36]. Because our analysis happens before optimization, however, it should be possible for the compiler to perform this analysis in the course of its normal duties as long as the pure and idempotent nature of the SFI mask can be expressed to the later optimizers. It should also be possible to add additional certified analyses to our tool chain, as in [46], in order to effect more robust policies. Also, the method could be made foundational by extending the semantic preservation theorem in CompCert all the way to executable machine code. Finally, it would be possible to write a verified compiler from LLVM to Cminor, effectively making our scheme a drop-in replacement for the proposed [20]. Indeed, there have been recent efforts to formalize the semantics of LLVM in Coq [22], making this goal all the more realistic.

Acknowledgments

We wish to thank sincerely the anonymous referees and everyone who read drafts of this work, especially Lennart Beringer, Drew Dean, and Edward W. Felten. This work was supported in part by DARPA under grant FA8750-12-2-0293. Joshua Kroll was supported by the National Science Foundation Graduate Research Fellowship Program under grant number DGE-1148900.

References

[1] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," in *SOSP*, 1994.

[2] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy*, 2009.

[3] C. Small and M. Seltzer, "MiSFIT: Constructing safe extensible systems," *IEEE Concurrency*, vol. 6, no. 3, 1998.

[4] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with API integrity and multi-principal modules," in *SOSP*, 2011.

[5] J. Siefers, G. Tan, and G. Morrisett, "Robusta: taming the native beast of the JVM," in *CCS*, 2010.

[6] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *USENIX Security*, 2010.

[7] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, , and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *PLDI*, San Jose, CA, Jun 2011.

[8] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *USENIX Security*, 2006.

[9] J. A. Kroll and D. Dean, "BakerSFIeld: Bringing software fault isolation to x64," SRI International, Tech. Rep., Nov 2009.

[10] *Native Client Security Contest*, Google, May 2009. [Online]. Available: http://code.google.com/contests/nativeclient-security/

[11] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, 2009.

[12] ——, "A formally verified compiler back-end," *J. Automated Reasoning*, vol. 43, no. 4, 2009.

[13] Coq Development Team, "The Coq proof assistant," 2012. [Online]. Available: http://coq.inria.fr/

[14] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004.

[15] McCamant, S., "A Machine-Checked Safety Proof for a CISC-Compatible SFI Technique," MIT Computer Science and Artificial Intelligence Laboratory, Tech. Rep. MIT-CSAIL-TR-2006-035, 2006.

[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications." in *CCS*, 2005.

[17] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger SFI for the x86," in *PLDI*, 2012.

[18] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "ARMor: fully verified software fault isolation," in *ACM International Conference on Embedded Software*, 2011.

[19] Pilkiewicz, A. (2011) FPdNaCl: Formally proved Native Client inner sandbox validator. [Online]. Available: https://github.com/pilki/FPdNaCl

[20] *PNaCl: Portable Native Client*, Google, November 2011. [Online]. Available: http://www.chromium.org/nativeclient/pnacl

[21] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI*, 2011.

[22] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Formalizing the LLVM Intermediate Representation for Verified Program Transformations," in *POPL*, 2012.

[23] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software guards for system address spaces," in *OSDI*, 2006.

[24] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems, Second Edition*. Wiley Publishing, Inc., 2008.

[25] J. Rees and W. Clinger, "Revised report on the algorithmic language Scheme," *ACM Sigplan Notices*, vol. 21, no. 12, pp. 37–79, 1986.

[26] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *SOSP*, 1999.

[27] Chen, B. (2008) Native Client Issue Tracker: Inner Sandbox Escape (call memory dereference). [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=23

[28] G. Necula, "Proof-carrying code," in *POPL*, 1997.

[29] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *J. Automated Reasoning*, vol. 41, no. 1, 2008.

[30] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart, "The CompCert Memory Model, Version 2," INRIA, Tech. Rep. RR-7987, June 2012.

[31] A. McCreight, T. Chevalier, and A. Tolmach, "A certified framework for compiling and executing garbage-collected languages," in *ICFP*, 2010.

[32] U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: a retrospective," in *Proceedings of the 1999 Workshop on New Security Paradigms*, 1999.

[33] J. Liedtke, "Improved address-space switching on Pentium processors by transparently multiplexing user address spaces." German National Research Center for Information Technology, Tech. Rep. 933, Sept. 1995.

[34] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *SOSP*, 2009.

[35] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *CCS*, 2011.

[36] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[37] J. Douceur, J. Elson, J. Howell, and J. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *OSDI*, December 2008.

[38] G. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *OSDI*, 1996.

[39] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ToPLaS*, vol. 21, no. 3, 1999.

[40] A. W. Appel, "Foundational proof-carrying code," in *IEEE Symposium on Logic in Computer Science*, 2001.

[41] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *POPL*, 2002.

[42] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX ATC*, 2002.

[43] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, "Cyclone: A type-safe dialect of C," *C/C++ User's Journal*, vol. 23, no. 1, January 2005.

[44] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management*, 2010.

[45] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[46] A. W. Appel, "Verified software toolchain," in *20th European Symposium on Programming*, vol. 6602, 2011.