ARMor: Fully Verified Software Fault Isolation

Lu Zhao
University of Utah, USA
luzhao@cs.utah.edu

Guodong Li
Fujitsu Laboratories of America, USA
gli@fla.fujitsu.com

Bjorn De Sutter
Ghent University, Belgium
bjorn.desutter@elis.ugent.be

John Regehr
University of Utah, USA
regehr@cs.utah.edu

ABSTRACT

We have designed and implemented ARMor, a system that uses software fault isolation (SFI) to sandbox application code running on small embedded processors. Sandboxing can be used to protect components such as the RTOS and critical control loops from other, less-trusted components. ARMor guarantees *memory safety* and *control flow integrity*; it works by rewriting a binary to put a check in front of every potentially dangerous operation. We formally and automatically verify that an ARMored application respects the SFI safety properties using the HOL theorem prover. Thus, ARMor provides strong isolation guarantees and has an exceptionally small trusted computing base—there is no trusted compiler, binary rewriter, verifier, or operating system.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*

General Terms

Verification

Keywords

Software Fault Isolation, ARM Executables, Program Logic, Automated Theorem Proving

1. INTRODUCTION

Isolation—the guarantee that one computation on a machine cannot affect other computations—is a fundamental system service supporting multiprogramming. Reliable isolation enables many useful kinds of coexistence; for example, users can safely run code downloaded from the Internet, servers belonging to mutually-untrusting parties can be run in different virtual machines on the same physical box, and embedded systems can be made smaller and cheaper by running critical and non-critical code on the same processor.

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

Isolation can be implemented in many ways, including using physical partitioning across processors, hardware-assisted address space management, type-safety at the programming language level, capability-based systems, and software fault isolation (SFI). Each of these methods has advantages and disadvantages. In this paper we focus on SFI, which is implemented by rewriting a compiled program to insert a dynamic check in front of every dangerous operation. An operation is *dangerous* if it may violate the security policy, for example by writing to out-of-bounds storage or attempting to execute unauthorized code. We have implemented an SFI system that guarantees *memory safety*: store operations are confined to predefined regions, and *control flow integrity*: execution may not escape a predetermined control flow graph.

In the context of critical embedded systems, a key advantage of SFI is that it potentially has the smallest *trusted computing base* (TCB) of any software-based isolation scheme. Notably, the TCB for SFI does not necessarily contain the compiler or operating system. Furthermore, the binary code of an executable can be fully formally verified using a mechanical theorem prover. We have done this, resulting in the creation of ARMor: the first fully verified, realistic implementation of SFI that we are aware of.

The ARMor toolchain operates as follows:

1. An ARM executable is created by a compiler.
2. An extension for Diablo [23] that we developed inserts a check before every potentially dangerous operation.
3. With a novel program logic framework, we use the HOL theorem prover [9] to automatically prove that the rewritten binary conforms to the memory safety and control flow integrity policies that we have written.

The TCB for ARMor contains only our memory safety and control flow integrity policies, a program logic, a well-vetted formal semantics for the ARM instruction set architecture (ISA), and HOL itself.

Formal verification at the binary level is useful because SFI can easily go wrong. First, the policies themselves are fairly subtle: verification exposes them to a large amount of additional scrutiny. Second, tricky source code errors can result in vulnerable implementations. For example, the authors of the Google Native Client encountered a situation where a routine refactoring of C code permitted the compiler to silently eliminate a crucial safety check [3]. Third,

UU IR Author Manuscript

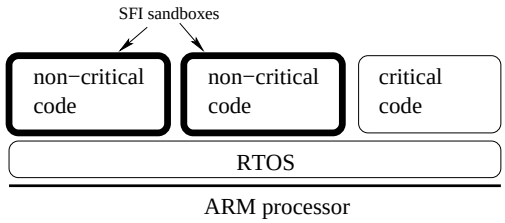

Figure 1: Multitasking embedded systems with fully verified isolation, enabled by ARMor

compiler bugs are not uncommon [25] and can undermine the safety policy.

The goal of ARMor is to provide a very high-confidence argument that its memory safety and control flow integrity policies are not violated, and that systems such as the one depicted in Figure 1 can be trusted. The key property is that larger, non-critical components (GUIs, network stacks, etc.) can be trustably isolated from smaller critical components (control loops, the RTOS, etc.) using SFI. A bug in application code or in the binary rewriter or in the compiler can result in verification failure, or is guaranteed by ARMor's proof that it cannot affect the rest of the machine—an ARMor-level trap that stops the faulting computation. Ideally these bugs are discovered during pre-deployment testing.

In ARMor, the safety policy is parameterized by a control flow graph (CFG): a description of legal control flow transfers in a list of instructions. If the CFG is erroneously computed, and is too small, the proof will fail. If the CFG is erroneously computed, and is too large, then ARMor's safety policy will provide a weaker safety guarantee than is desirable. In practice we use Diablo, a binary rewriter, to discover a CFG that is then passed to the safety policy. However, other methods, such as computing the CFG by hand, could also be used.

We present a high-level theorem that ARMor provides about an executable in Section 2. Next, we describe an SFI implementation in Section 3; Section 4 and Section 5 describe the instruction semantics and logic that ARMor uses. We illustrate the logic with a complete proof of an example in Section 6. We present results in Section 7, discuss related work in Section 8, and conclude in Section 9.

2. HIGH-LEVEL GUARANTEE

For a sandboxed ARM executable, ARMor guarantees that for every instruction of the executable, the set of memory addresses the instruction writes to is a subset of a predefined memory set, and the program counter (PC) following the execution of the instruction points to a successor instruction given in a predetermined CFG. More formally, we model a CFG as a function `succ`: given the address of an instruction, it returns the set of addresses of legal successor instructions. Each instruction has a unique memory address, and we use a pair (a, i) to represent an instruction i is stored at memory address a. Let `prog` be the set of paired instructions of an executable, `mem` be the set of predefined memory addresses, $ms(i)$ be the set of memory addresses an instruction writes to, and l and l' be values of the PC before and after the instruction executes. l is the address of i in order

```
block1: 0x0,  sub R1,R1,#0x1     // R1 <- R1 - 4
        0x4,  teq R2,#0x40000000 // test equality
        0x8,  bne +#16           // branch not equal
block2: 0xC,  strb R1,[R2]       // store a byte
        0x10, teq R1,#0x0        // test equality
        0x14, bne -#20           // branch not equal
block3: 0x18, add R1,R2,#0x4     // R1 <- R2 + 4
block4: 0x1C, b +#0              // branch to itself
```

(a) Program instructions

CFG policy `succ`	
input	output
0x8	{0x18, 0xC}
0x14	{0x0, 0x18}
0x18	{0x1C}
0x1C	{0x1C}
others	λa. {a+4}
Memory safety policy `mem`	
$\{a \mid 0x40000000 \leq a \wedge a < 0x40001000\}$	

(b) Safety policies

Figure 2: An example for illustrating ARMor's logic

to execute i, and l' is decided by the underlying instruction semantics, Section 4.2. ARMor proves the following:

THEOREM 1. $\forall(l, i) \in \mathit{prog}.\,(ms(i) \subseteq \mathit{mem}) \wedge (l' \in \mathit{succ}(l))$ *with respect to a given initial state.*

We take the following steps to prove it:

1. We choose an existing well-vetted formal semantics of the ARM ISA. It was developed independently by researchers at Cambridge [8, 14] and has been used as the basis for several previous projects and is believed to be correct. For example, Fox proved that the ARM semantics implemented one particular version of the ARM hardware [7].
2. We augment the existing semantics to create a safe instruction semantics by asserting the above safety properties, i.e., $(ms(i) \subseteq \mathtt{mem}) \wedge (l' \in \mathtt{succ}(l))$, for every instruction of a program. The instructions of the program are obtained from Diablo and compared with GCC objdump output. The CFG policy of the program is also obtained from Diablo.
3. We develop a novel program logic that discharges the assertions at two levels. The first level is a composition process used in Hoare logic reasoning, whose results are Hoare judgments for code blocks. The other level is a top-level program judgment that defines an implication relationship among the code block judgments.
4. After automatically generating the code block judgments, we use an abstract interpretation to discover the global relationship among them and use its result to prove the top-level program judgment.
5. The top-level program judgment is a global specification for the program with the safe instruction semantics. Because the semantics asserts the safety properties for every instruction of the program over all possible executions, the above theorem follows naturally.

We illustrate these steps with a very simple example shown in Figure 2.a, and we assume its safety policies in Figure 2.b.

UU IR Author Manuscript

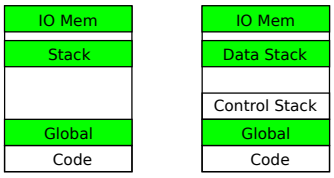

Figure 3: Left: the original memory layout of an executable. Right: ARMor's transformed layout.

We first use a Hoare logic to derive the precondition and postcondition of each block. For example, block1's postcondition when control flows from 0x8 to 0xC has $r2 = 0x40000000$. Next, we follow and verify the CFG of the program and derive the fact that the address of the store instruction satisfies $\{r2\} \subseteq$ `mem` for every execution of this program starting from every valid initial state. One key feature of this method is to reduce the proof of safety properties to linking and checking specific preconditions and postconditions of all blocks. Theorem 1 holds if and only if this procedure succeeds. We explain this process in detail throughout this paper.

3. SFI IMPLEMENTATION

We developed a binary transformation tool based on Diablo [23]. Diablo is a link-time binary rewriting framework that analyzes and transforms statically linked executables to achieve better performance, smaller code size, and improved security. Our tool utilized Diablo's framework and implemented three new transformations for an ARM executable emitted by GCC. These transformations ensure that either the invariants required by ARMor hold, or else a trap handler executes.

3.1 Protecting Return Addresses

Spilled return addresses of functions are normally pushed onto the stack, and this imposes a great danger for control flow integrity, because they can be overwritten by buffer overflow or pointer manipulation. We introduce a special region into the runtime memory of an executable to save the return addresses exclusively, which is named *control stack*, Figure 3. The original stack is called *data stack* for distinction. Our rewriting tool inserts code to save and restore return addresses to and from the control stack upon function calls and returns. This control stack provides invariants that are strong enough to prove assertions of control flow integrity about return addresses (Section 4.2). We patched GCC to reserve a dedicated register, R8, as the pointer of the control stack.

3.2 Checking Unknown Store Addresses

We used Diablo's linear constant analysis to find the store instructions whose addresses are unknown statically, and we inserted a call to an address checking routine before them. The routine takes the start and end addresses of the memory range to be written by an instruction and checks whether the range falls within a set of memory addresses. This set of addresses includes the data stack, the global data section, and I/O addresses as depicted in Figure 3. If the check succeeds, the routine simply returns, transferring control to the store instruction; otherwise, the routine aborts the program. Figure 4 illustrates this transformation.

```
stmia  R2,{R6,R7}
```

(a) Original code

```
mov    R0,R2
add    R1,R0,#0x8
bl     dguard
stmia  R2,{R6,R7}
```

(b) Instrumented code

```
if [R0, R1) falls in the global data section,
    the data stack, or I/O region
then return else abort
```

(c) Pseudo-code of the checking routine (R0 and R1 are used to pass parameters)

Figure 4: Checking unknown store addresses

```
        mov pc, R2        ;indirect jump
        ...
target:
        str R1, [R0]      ;target
```

(a) Original code

```
        mov  R0, 0x4          ;load mangled word pattern
        mov  R0, R0, ROR #1   ;right rotate 1 bit
        ldr  R1, [R2, #-4]    ;R1 <- Mem[R2 - 4]
        cmp  R0, R1           ;compare ids
        bne  invalid          ;abort if not equal
        mov  pc, R2           ;indirect jump
        ...
        0x8                   ;unique word pattern
target:
        str R1, [R0]          ;target
```

(b) A unique ID and the check for its presence, where `0x8` is `0x4` left rotated by 1 bit.

Figure 5: Constraining unknown jumps

Our rewriter unconditionalizes all conditional stores: it removes the condition of a store and inserts a conditional branch instruction before it with the same condition which jumps over the store when necessary. Next, it uses the same procedure discussed above to safe-guard the unconditional stores.

3.3 Constraining Indirect Jumps

Indirect jumps, such as those used to implement function pointers and jump tables, represent potential risks for the control flow integrity, since the value of a destination register might not coincide with the value given in the CFG policy.

We constrain these jumps by a method proposed by Abadi el at [1]. Our rewriter inserts, before a jump target in the given CFG, a unique identifier that is not present in the code of an executable. In the ARM ISA, the rewriter places the identifier in a datapool, which is a fragment of code memory storing constant data. Next, it inserts a piece of code right before the jumping instruction to check the presence of the identifier. The code loads a value mangled from the identifier by a fixed operation such as by left rotation with one bit, and restores the identifier by a reverse operation. If the restored value is the same as the identifier, then the target is correct. Figure 5 shows this transformation.

4. INSTRUCTION SEMANTICS

We first introduce the existing formal ARM semantics, from which we create a safe instruction semantics.

UU IR Author Manuscript

$\{\texttt{PC}\ p * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle\}$
$(p, \texttt{0xE5C21000})$ // `strb R1, [R2]`
$\{\texttt{PC}\ (p+4) * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f)\}$

Figure 6: Axiomatic semantics of `strb R1, [R2]`

4.1 ARM Semantics

The ARM semantics comes in as Hoare triples proven in the HOL theorem prover. Figure 6 shows the semantics for the store instruction `strb R1, [R2]`. As `w2w` converts a 32-bit word into an 8-bit word, the semantics states that after execution of the instruction, the value at memory address r2 is updated to the least significant byte of r1, and the PC is increased by 4. From our perspective, those Hoare triples are equivalent to instruction axioms in an axiomatic semantics, since we take those theorems as granted and build our logic on top of it. The detail of the design and implementation of the semantics is available in [7, 8, 14]. Here, we summarize important properties related to our cause.

Machine states include registers, memory cells, status flags, and the current program status register. For example, $\texttt{PC}\ p$ in the precondition of Figure 6 asserts that the program counter has value p and that p is word-aligned; $\texttt{R}\ 2\ r2$ and $\texttt{R}\ 1\ r1$ assert that registers R2 and R1 have values $r2$ and $r1$, respectively; $\texttt{MEMORY}\ dom\ f$ asserts that some set of memory addresses dom has value f (these are symbolic values). The $\mapsto$ operator is defined as: $(a \mapsto b)\ f = \lambda x.$ if $x = a$ then b else $f\ x$, namely, the result is a new function where a is mapped to b while other values stay unchanged. $((r2 \mapsto (\texttt{w2w}\ r1))\ f)$ means that only the value at address $r2$ is updated to $(\texttt{w2w}\ r1)$.

The $*$ operator is a separating conjunction [19]: (1) a triple only asserts the local state (the parts of state that are used by the instruction), and a global version may be achieved by using the frame rule (which adds resource assertions not used by a judgment onto it); (2) if a separating conjunction expression asserts a machine resource more than once (excluding a pure assertion), then its value is `false`.

$\langle\rangle$ represents a pure assertion [19], i.e., it does not assert any machine resource but serves as a predicate to specify the boolean relationship between variables. $\langle r2 \in dom \rangle$ states that $r2$ has to be in the domain of the memory function f in order for this transition to take place.

The pair $(p, \texttt{0xE5C21000})$ represents code assertion for the instruction, meaning that the value `0xE5C21000` is stored at memory address p.

Some boolean operators such as implication ($\Rightarrow$) and disjunction ($\vee$) are lifted to the separating conjunction level; $p \overset{*}{\Rightarrow} q$ means $\lambda s.\ (p\ s \Rightarrow q\ s)$; $p \overset{*}{\vee} q$ is $\lambda s.\ (p\ s \vee q\ s)$.

4.2 Safe Instruction Semantics

It is difficult to formalize memory safety or control flow integrity in Hoare logic, because a Hoare judgment only gives the pre- and post- states of code and hides intermediate states. In order to assert the properties for all states explicitly, we define a *safe instruction rule*, which creates a semantics that ensures both properties in every state.

$\{\texttt{PC}\ p * \texttt{R}\ 8\ k * \langle (p+4) \in \texttt{succ}(p) \rangle * \langle msafe \rangle *$
$\texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1\}$
$(p, \texttt{0xE5C21000})$ // `strb R1, [R2]`
$\{\texttt{PC}\ (p+4) * \texttt{R}\ 8\ k *$
$\texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * \texttt{MEMORY cm}\ cf *$
$\texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1\}$

Figure 7: The augmented theorem of `strb R1, [R2]`

Safe Instruction Rule. From an augmented instruction axiom, we define a new state transition relation `SAFE_INS` by HOL's inductive relation definition [13].

$$\frac{\begin{array}{l}\{\texttt{PC}\ l * \texttt{R}\ 8\ k * \langle l' \in \texttt{succ}(l) \rangle * \langle MemSafe \rangle * \\ \quad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * p\} \\ \quad (l,\ ins) \\ \{\texttt{PC}\ l' * \texttt{R}\ 8\ k' * \ \texttt{MEMORY dm}\ df' * \texttt{MEMORY cm}\ cf' * \texttt{MEMORY pm}\ pf * q\}\end{array}}{\begin{array}{l}\texttt{SAFE\_INS}\ (l,\ \texttt{R}\ 8\ k\ * \langle l' \in \texttt{succ}(l) \rangle * \langle MemSafe \rangle * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * p) \\ \quad (l,\ ins) \\ \quad (l',\ \texttt{R}\ 8\ k' * \ \texttt{MEMORY dm}\ df' * \texttt{MEMORY cm}\ cf' * \texttt{MEMORY pm}\ pf * q).\end{array}}$$

The augmented instruction axiom serves as the antecedent of the rule, where l is the value of the PC in the precondition, k is the value of the control stack pointer, R8, p and q represent other assertions that are not explicitly written out, and corresponding values of machine resources in the postcondition are marked with a prime $'$. As an example, Figure 7 shows the augmented version for the axiom in Figure 6.

In the augmented theorem, we divide the data memory of a program into three parts and use three separate assertions. The three parts are the writable data memory set (WD), the control stack (CS), and the datapool (DP). WD includes the data stack, the global data section, and the I/O addresses as depicted in Figure 3. WD and CS form the predefined memory set `mem` described in Theorem 1. Because the three parts are disjoint, we use three sets of memory addresses to represent them: `dm` represents WD, `cm` represents CS, and `pm` represents DP. We also use three separate assertions: $\texttt{MEMORY dm}\ df$ asserts WD, $\texttt{MEMORY cm}\ cf$ asserts CS, and $\texttt{MEMORY pm}\ pf$ asserts DP. We explicitly write out these memory assertions in order to utilize the feature provided by the separation logic [19]; that is, these memory addresses can not be asserted by other assertions, and doing so will result in a `false` value.

Formalization of safety properties. The augmented theorem has two important safety assertions. One is $\langle l' \in \texttt{succ}(l) \rangle$, which asserts control flow integrity. The other is $\langle MemSafe \rangle$, which asserts memory safety:

$$MemSafe = \text{if } \texttt{not}\ (\texttt{isStore}\ ins) \text{ then } \texttt{true} \text{ else}$$
$$ms(ins) \subseteq (\text{if } k = k' \text{ then } \texttt{dm} \text{ else } \texttt{cm})$$

where $ms(ins)$ is the set of memory addresses that the instruction writes to. Recall that R8 is the control stack pointer. The rationale of (if $k = k'$ then `dm` else `cm`) is that if the control stack pointer does not change during execution, then the instruction should write to `dm`; otherwise, it writes

$$\texttt{SAFE\_INS}\ (p, \texttt{R}\ 8\ k * \langle (p+4) \in \texttt{succ}(p) \rangle * \langle \{r2\} \subseteq \texttt{dm} \rangle * \cdots)$$
$$(p, \texttt{0xE5C21000})\ //\ \texttt{strb R1,[R2]}$$
$$(p+4, \texttt{R}\ 8\ k\ * \cdots)$$

Figure 8: Safe instruction semantics of `strb r1,[r2]`

to `cm`. This ensures that changes to the control stack can only be done through its pointer in R8.

In the example of Figure 7, only the value of the memory set `dm` is updated to $df' = ((r2 \mapsto (\texttt{w2w}\ r1))\ df)$, while the value of `cm` stays the same as in the precondition ($cf' = cf$); the assertion of the control flow integrity is $\langle (p+4) \in \texttt{succ}(p) \rangle$, and the memory safety assertion is

$$msafe = \text{if } \texttt{not}\ (\texttt{isStore 0xE5C21000}) \text{ then } \texttt{true} \text{ else}$$
$$ms(\texttt{0xE5C21000}) \subseteq (\text{if } k = k' \text{ then } \texttt{dm} \text{ else } \texttt{cm}).$$

The safe instruction rule plays very important roles in ARMor's logic framework. First, if we directly used an axiom with safety assertions in logic, it would not be clear what caused the absence of the assertions after they are simplified to `true` and removed from the precondition, because the axiom may not have assertions at all. With the new `SAFE_INS` relation created by the safe instruction rule, we are always assured that they have been discharged, as there is no instruction without having the safety assertions in this relation. As a result, when writing specifications (assertions) for memory safety and control flow integrity, we do not care about the exact program points where the specifications originate; we only care about if any specifications can be derived properly, i.e., if any assertions can be discharged. Second, our logic uses a flexible construct, a label predicate, to handle arbitrary jumps used in machine code, and it requires a new syntax which the axioms do not have.

After applying the safe instruction rule to the augmented theorem in Figure 7, we get the safe instruction semantics for `strb R1, [R2]` in Figure 8. For brevity, we have omitted the assertions for memory, R2 and R1. They are the same as in Figure 7. Notice that the memory safety assertion has been simplified, as this is a store instruction, and $k' = k$.

5. A NOVEL LOGIC FRAMEWORK

We developed a novel program logic that has a two-layer structure for reasoning about executables. The first layer is a Hoare-style logic that reasons about a code block with sequential execution structure such as a basic block by generating a local Hoare judgment. The novelty comes from the following. First, the second layer is a top-level program judgment stipulating the relationship among the Hoare judgments of code blocks: for every code block, the postconditions of its predecessor blocks imply its precondition. This idea originated from Floyd's inductive assertion [6], and we formalize it in this program logic. Second, we use a simple concept, *label predicate*, to connect the two layers. It is inspired by Tan and Appel's label continuation [22], but we use a much simpler, direct (instead of continuation) interpretation and do not have a complicated semantics and a complex soundness proof used by the latter. Third, our logic facilitates proof automation in two ways. One is the composition of code block judgments, which is automated by SML programming, SML being the meta-language of the HOL theorem prover. The other is discovering the relationship among these judgments automatically by an abstract interpretation.

5.1 Label Predicates

Our logic's assertion language is a set of label predicates. Informally, *a label predicate* is a pair of a label and a predicate, interpreted as that the predicate holds at the associated label. A set of label predicates means that there is a true label predicate in the set. Formally, the syntax of a label predicate is

lp	$\in$	$LabelPred$	$=$	$LabelExp \times StateAssert$
l	$\in$	$LabelExp$	$=$	word32
p	$\in$	$StateAssert$	$=$	separating conjunction expression

Its interpretation is defined by a semantic function `LP2SP`, and another function, `LPSET`, interprets a set of label predicates:

$$\texttt{LP2SP}\ (l,\ p) = \texttt{PC}\ l\ * p$$
$$\texttt{LPSET}\ P\ =\ \lambda s.\ (\exists lp.\ lp \in P \wedge (\texttt{LP2SP}\ lp)\ s)\,.$$

We denote the subsumption relation between two sets of label predicates as $\stackrel{lp}{\Rightarrow}$:

$$P \stackrel{lp}{\Rightarrow} Q \textbf{ iff } \quad (\texttt{LPSET}\ P) \stackrel{*}{\Rightarrow} (\texttt{LPSET}\ Q).$$

5.2 A Hoare Logic

This first layer is implemented by the following set of definitions that bridge the gap between `SAFE_INS` and a Hoare judgment.

First, a `step` relation implements a state transition:

$$\texttt{step}\ i\ s\ t \textbf{ iff}$$
$$\exists lp\ kq.\ (\texttt{SAFE\_INS}\ lp\ i\ kq) \wedge (\texttt{LP2SP}\ lp)\ s \wedge (\texttt{LP2SP}\ kq)\ t.$$

It reads that a transition from state s to state t by instruction i is equivalent to a transition from s to t by the safe semantics, indicating that the transition is safe.

Next, a sequence relation implements the concept of n-step execution:

$$\begin{aligned}&\texttt{seq}\ C\ sq\ s \textbf{ iff}\\ &\quad (sq\ 0 = s) \wedge\\ &\quad (\forall n.\ \texttt{if}\ \exists i \in C.\ \exists t.\ \texttt{step}\ i\ (sq\ n)\ t\\ &\qquad \texttt{then}\ \exists i \in C.\ \texttt{step}\ i\ (sq\ n)\ (sq\ (n+1))\\ &\qquad \texttt{else false})\end{aligned}$$

where C is a set of instructions, and sq is a mapping from integer to state. It reads that in an instruction set, if there exists an instruction that can take a state to the next, then do the transition; otherwise, execution gets stuck, indicated by the `false` value.

We first define a single-entry single-exit Hoare judgment:

$$\begin{aligned}&\texttt{sglspec}\ \{lp\}\ C\ \{kq\} \textbf{ iff}\\ &\qquad \forall r\ s.\ ((\texttt{LP2SP}\ lp) * r)\ \ s \Rightarrow \forall sq.\ \texttt{seq}\ C\ sq\ s\\ &\qquad\quad \Rightarrow \exists n.\ ((\texttt{LP2SP}\ kq) * r)\ \ (sq\ n).\end{aligned}$$

It reads that if the precondition lp holds for an initial state s, then n steps later, the postcondition kq holds for another state $(sq\ n)$. The universally quantified r forces any and only those resources used by the code to be included in the pre- and post-conditions.

Label predicate rules:

$$\frac{}{P \overset{lp}{\Rightarrow} P}\ \text{LPRefl} \qquad \frac{}{P \overset{lp}{\Rightarrow} (P \cup Q)}\ \text{LPExt}$$

Hoare rules:

$$\frac{\{P1\}\ C1\ \{Q1\} \qquad \{P2\}\ C2\ \{Q2\}}{\{P1 \cup P2\}\ (C1 \cup C2)\ \{Q1 \cup Q2\}}\ \text{Union}$$

$$\frac{\{P \cup M\}\ C\ \{Q \cup M\} \qquad \{M\}\ C\ \{Q\}}{\{P \cup M\}\ C\ \{Q\}}\ \text{Discharge}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{(l, p * r) | (l, p) \in P\}\ C\ \{(k, q * r) | (k, q) \in Q\}}\ \text{Frame}$$

$$\frac{\{P\}\ C\ \{Q\} \qquad R \overset{lp}{\Rightarrow} P}{\{R\}\ C\ \{Q\}}\ \text{Strengthen}$$

$$\frac{\{P\}\ C\ \{Q\} \qquad Q \overset{lp}{\Rightarrow} R}{\{P\}\ C\ \{R\}}\ \text{Weaken}$$

Label predicate merge rules:

$$\{P \cup \{(l, p)\} \cup \{(l, q)\}\}\ C\ \{Q\} = \{P \cup \{(l, p \overset{*}{\vee} q)\}\}\ C\ \{Q\}$$
$$\{P\}\ C\ \{Q \cup \{(l, p)\} \cup \{(l, q)\}\} = \{P\}\ C\ \{Q \cup \{(l, p \overset{*}{\vee} q)\}\}$$

Figure 9: Some proven inference rules

We extend it to a multiple-entry multiple-exit Hoare judgment:

$$\texttt{SPEC}\ \{P\}\ C\ \{Q\}\ \textbf{iff}\ \texttt{sglspec}\ (\texttt{LPSET}\ P)\ C\ (\texttt{LPSET}\ Q)$$

where P and Q are sets of label predicates. Informally, it means that if there exists a true label predicate in the precondition, then there exists a true label predicate in the postcondition some steps later.

5.2.1 Instruction and Inference Rules

It is relatively straightforward to prove instruction and inference rules by definition substitutions. We list some important inference rules in Figure 9, omitting the leading relation marker `SPEC`. Most rules are standard in Hoare logic. An unusual rule is Discharge, which removes unnecessary intermediate label predicate entries in the postcondition. Label predicate merge rules are used to combine and split label predicate entries which have the same label.

5.2.2 Soundness

This Hoare logic is sound, because we derive all the rules from the formal safe semantics of the ARM ISA.

5.2.3 Composing Code Block Judgments

Like in a traditional Hoare logic, the reasoning process for a code block is to compose a block Hoare triple from instruction rules. The inference rules used in composition are Frame, Union, Discharge, Strengthen, and LPExt. We show their usage in composing the judgment of block2 of the example in Figure 2, starting with the first two instructions.

For this concrete example, the rule for the store instruction is instantiated and simplified to the following ($p = 0xC$, and $(0x10) \in \texttt{succ}(0xC) = \texttt{true}$):

$$\{(0xC, \texttt{MEMORY dm}\ df * \langle\{r2\} \subseteq \texttt{dm}\rangle * a1 * a2)\}$$
$$(0xC, \texttt{0xE5C21000})\ //\ \texttt{strb R1,[R2]}$$
$$\{(0x10, \texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * a1 * a2)\}$$
$$\text{where } a1 = \texttt{R}\ 8\ k * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf$$
$$a2 = \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1$$

and the simplified rule for the test instruction is:

$$\{(0x10, \texttt{R}\ 1\ r1 * \texttt{S sZ}\ z * \texttt{MEMORY dm}\ df * a1)\}$$
$$(0x10, \texttt{0xE3310000})\ //\ \texttt{teq R1,\#0x0}$$
$$\{(0x14, \texttt{R}\ 1\ r1 * \texttt{S sZ}\ (r1 = 0) * \texttt{MEMORY dm}\ df * a1)\}$$

where $\texttt{S sZ}\ z$ asserts that the zero status flag (`sZ`) has value z. For brevity, we have omitted the assertions for other status flags.

First, we find all state assertions used by these two rules and use the Frame rule to add the assertions of resources that are not used by an instruction to that instruction. For example, we add $\texttt{R}\ 2\ r2$ to the second rule and the assertions of status flags to the first rule. In this process, we need to instantiate the free variables of the second rule to the corresponding values in the postcondition of the first rule. For example, *df* in the second rule is instantiated to $((r2 \mapsto (\texttt{w2w}\ r1))\ df)$, because the second instruction starts with the ending state of the first instruction. Next, we apply the Union rule to compose the two rules together, and use the Discharge rule to remove the intermediate entry in the postcondition, which is the entry with label $0x10$. Finally, we use the LPExt and the Strengthen rules to remove the intermediate entry from the precondition to get the judgment for the first two instructions.

By repeating this procedure, we compose the judgment of block2, and the final results are:

$$\{(0xC, \texttt{MEMORY dm}\ df * \langle\{r2\} \subseteq \texttt{dm}\rangle * \langle r1 \neq 0\rangle * \texttt{S sZ}\ z * a1 * a2)\}$$
$$\text{block2} \qquad (1)$$
$$\{(0x0, \texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * \texttt{S sZ}\ (r1 = 0) * a1 * a2)\}$$

$$\{(0xC, \texttt{MEMORY dm}\ df * \langle\{r2\} \subseteq \texttt{dm}\rangle * \langle r1 = 0\rangle * \texttt{S sZ}\ z * a1 * a2)\}$$
$$\text{block2} \qquad (2)$$
$$\{(0x18, \texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * \texttt{S sZ}\ (r1 = 0) * a1 * a2)\}.$$

Notice that block2 has two separate judgments with each for a branch condition. The branch conditions, $\langle r1 \neq 0\rangle$ and $\langle r1 = 0\rangle$, originate from the branch instruction `bne -#20`, which has two separate axioms.

5.2.4 Discharging and Pushing Up Safety Assertions

The composition process handles a safety assertion in two ways. The first is to discharge a safety assertion if possible, for example, the assertions of control flow integrity are discharged due to simplification using the `succ` function. If a safety assertion cannot be discharged, such as the memory assertion of the store instruction, then it is pushed up to the precondition of the composed judgment. We discharge those up-pushed assertions in the second layer of the logic. It is noteworthy that this composition procedure is completely mechanical and does not require complicated rule selections. We automated it by SML programming.

UU IR Author Manuscript

5.3 Well-Formed Code Blocks

A multiple-entry Hoare judgment does not model a basic block well, because a basic block has only one entry address. We define a well-formed code block as a single-entry multiple-exit Hoare judgment by imposing two conditions: (1) there is only one entry address for the code; (2) the label of a label predicate in the precondition must be the entry address. Formally, it is

$$\texttt{WF\_CBL}\ P\ C\ Q\ \textbf{iff}$$
$$(\texttt{SPEC}\ \{P\}\ C\ \{Q\}) \wedge (\forall (l,p) \in P.\ \ l = \mathbb{L}(C))$$

where $\mathbb{L}(C)$ returns the entry address of an instruction set.

5.4 The Top-Level Program Judgment

This layer connects the Hoare judgments of code blocks to form the semantics of a program.

We first define an *implication* relation between a precondition and a postcondition at the program level.

$$Q \overset{P}{\Rightarrow} R\ \textbf{iff}\ \ \forall (l,p) \in R.\ \forall (k,q) \in Q.$$
$$(k = l) \Rightarrow \left(\{(k,q)\} \overset{lp}{\Rightarrow} \{(l,p)\}\right).$$

It reads that a set of label predicates Q implies another set of label predicates R at the program level if and only if for every label predicate lp in R, if a label predicate kq in Q has the same label with lp, then the singleton set of kq should imply the singleton set of lp.

We define the top-level judgment of a program as

$$\texttt{PROG\_SPEC}\ prog\ predecessor\ bspec\ \textbf{iff}$$
$$\exists kspec.$$
$$\forall cbl \in prog.$$
$$(\texttt{WF\_CBL}\ (bspec\ cbl)\ cbl\ (kspec\ cbl)) \wedge$$
$$(\forall cbl' \in (predecessor\ cbl).\ (kspec\ cbl') \overset{P}{\Rightarrow} (bspec\ cbl))$$

where *prog* is the set of code blocks of a program. The second to the last line of the definition requires that each code block is well-formed. The *predecessor* models the CFG policy at the code blocks level by a function: given a code block, it returns the set of predecessor code blocks. *bspec* and *kspec* are two global specifications on code blocks. The former is a mapping from code blocks to their preconditions, and the latter is a mapping from code blocks to their postconditions. The last line of the definition specifies that if a code block is a predecessor of another code block, then the postcondition of the former implies the precondition of the latter.

This logic does not compose over code blocks, so it does not need rules for composing any control flow transfers. As a result, it can handle arbitrary jumps commonly seen in executables. The requirement of the postcondition of a block implying the precondition of a successor block in the CFG policy holds for arbitrary control flow transfers including finite or infinite loops.

Our soundness argument for this program semantics says that execution never gets stuck if and only if the program is safe, which is captured by the given global specification (*bspec*). An intuitive argument is that when control reaches the end of a code block, it resumes on one of its successor blocks because of the implication relation. Formally, we may derive a program specification `PROG_SPEC` *prog predecessor bspec* if and only if: starting from the initial state s of a program, if the execution reaches the label of a code block, $\mathbb{L}(cbl)$, then the precondition defined by *bspec* on the block is ensured to be true. This shows that *bspec* is an accurate specification for the program. The theorem is

$$\forall s\ sq\ n\ cbl.$$
$$\texttt{seq}\ C\ sq\ s\ \wedge (bspec\ \texttt{entryBlock}\ s) \wedge \texttt{LABEL\_IN}\ (\mathbb{L}(cbl))\ (sq\ n)$$
$$\Rightarrow (bspec\ cbl)\ (sq\ n)$$

where `LABEL_IN` specifies that a state has a label, or the control reaches to the state: $\texttt{LABEL\_IN}\ l\ t$ **iff** $\exists p.\ (\texttt{LP2SP}(l,p))\ t$, and C is the set of instructions of the program $(\bigcup prog)$.

5.5 Discharging Safety Assertions Globally

In general, there is not an automated reasoning process for finding the two global specifications. However, for verifying the shallow safety properties presented in this paper, we describe an abstract interpretation that automatically discovers them.

This is a flow-sensitive backward analysis for discharging the safety assertions globally. Its domain is the power set of all concrete safety assertions occurring in the judgments of code blocks, and each judgment is a node. The transfer function runs as follows: when a node has an incoming safety assertion, it tries to derive the assertion from the label predicates in the postcondition of the node and whose label is the same as the incoming assertion; if it succeeds, which means the assertion is true, it does nothing; otherwise, it propagates the assertion along the flow, hoping that other nodes can discharge the assertion. In pseudo-code, it is

$$transfer\ (node, \Sigma_{in}):$$
$$\text{foreach}\ (l, assert)\ \text{in}\ \Sigma_{in}$$
$$\text{foreach}\ (l', p) \in \texttt{postcondition}(node)$$
$$\texttt{if}\ \ l' = l\ \ \texttt{and}\ \ (\texttt{not}\ \ (p\ \text{implies}\ assert))\ \ \texttt{then}$$
$$\Sigma_{out}(node) = \Sigma_{out}(node) \cup \{(\mathbb{L}(node), assert)\}$$

where Σ_{in} and Σ_{out} are the in-state and out-state of the analysis.

When the fixed point computation successfully terminates, we take the safety assertions that are propagated to a node and use the Frame rule to add them to the node. This makes the postcondition of a node's predecessor imply the precondition of the node.

6. PROVING THE EXAMPLE

We illustrate the reasoning process in the second layer by proving the example in Figure 2. This process has four steps and starts with the Hoare judgments of code blocks. The first step is to convert the local judgments of code blocks to the global version. The second is to find the two global specifications: *bspec* and *kspec*. The third is to prove the well-formedness of code blocks. Finally, we prove the `PROG_SPEC` judgment by instantiating its definition with the two specifications found, the program code and the CFG policy. The first step is simple, in which we apply the Frame rule to add unused resource assertions to block judgments. So is the third step, since it is a straightforward proof by definition. Therefore, we focus on the second and the last step.

We composed the Judgment 1 and 2 in Section 5.2.3 for block2. We use the same procedure to compose the judgments of other blocks and give the results below. For clarify of explanation, we assume that the judgments are already

in the global version, but we do not explicitly write out assertions of status flags and of other machine resources representing them with $\cdots$ unless necessary.

$$\{(0x0, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ z * \langle r2 \neq 0x40000000\rangle * \cdots)\}$$
$$\text{block1} \quad (3)$$
$$\{(0x18, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ (r1-1) * \texttt{S}\ \texttt{sZ}\ (r2 = 0x40000000) * \cdots\}$$

$$\{(0x0, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ z * \langle r2 = 0x40000000\rangle * \cdots)\}$$
$$\text{block1} \quad (4)$$
$$\{(0xC, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ (r1-1) * \texttt{S}\ \texttt{sZ}\ (r2 = 0x40000000) * \cdots\}$$

$$\{(0x18, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \cdots)\} \qquad \{(0x1C, \texttt{R}\ 2\ r2 * \cdots)\}$$
$$\text{block3} \quad (5) \qquad \text{block4} \quad (6)$$
$$\{(0x1C, \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ (r2+4) * \cdots\} \qquad \{(0x1C, \texttt{R}\ 2\ r2 * \cdots\}.$$

The central task is to discover the $\stackrel{P}{\Rightarrow}$ relation between a postcondition-and-precondition pair, which in turn boils down to discharging safety assertions. In this example, the safety assertion of block2, $\langle\{r2\} \subseteq \texttt{dm}\rangle$, can be discharged by the postcondition of Judgment 4, because it has the branch condition of $\langle r2 = 0x40000000\rangle$, and $\{0x40000000\} \subseteq \texttt{dm} = \texttt{true}$ (dm is the only predefined memory set). In order to get it formally, we frame the branch condition to the judgment itself. As a result, the assertion in the postcondition has $\texttt{R}\ 2\ r2 * \langle r2 = 0x40000000\rangle$, which implies $\langle\{r2\} \subseteq \texttt{dm}\rangle$. In our framework, this work is done by the abstract interpretation described in Section 5.5. There are no safety assertions in other blocks, so the relation between corresponding postcondition-precondition pairs is trivial.

Next, we merge the two judgments of the same block to form a single judgment for it. After merging, the branch conditions can be removed. For example, the two branch conditions of Judgment 3 and 4 are simplified to $\texttt{true}$ ($\langle(r2 \neq 0x40000000) \vee (r2 = 0x40000000)\rangle$). Afterwards, we build the two global specifications. Denote the preconditions of the four blocks as P1, P2, P3 and P4, respectively, and the postconditions of the blocks as Q1, Q2, Q3 and Q4, respectively. Then *bspec* = {blocki → Pi}, and *kspec* = {blocki → Qi}, for i=1,2,3,4. The other two parameters are *prog* = {blocki}, for i=1,2,3,4, and *predecessor* = {block1 → {block2}, block2 → {block1}, block3 → {block1, block2}, block4 → {block3, block4}}.

Finally, we instantiate the definition of PROG_SPEC with these terms. Since we have already found the $\stackrel{P}{\Rightarrow}$ relation between postcondition-and-precondition pairs, we are able to prove it by rewriting the definition step-by-step. An example of the relation is Q1 $\stackrel{P}{\Rightarrow}$ P2 as discussed above; the relation for other postcondition-precondition pairs holds similarly.

7. IMPLEMENTATION AND RESULTS

We implemented the SFI mechanisms in C using Diablo, and the logic framework in the HOL theorem prover. Although we developed C code and the abstract interpretation, neither of them is trusted. What we trust is a formal proof of the top-level program judgment. If there are errors in the SFI implementation or in the abstract interpretation, proving PROG_SPEC will fail. The purpose of the SFI implementation is to provide necessary invariants that make the proof succeed; without them or with buggy transformations, the proof will simply fail. The purpose of the abstract interpretation is to automate the discovery of the two global specifications that define the top-level judgment; without it, depending on human efforts to find any global invariants in machine code is daunting and very inefficient, if possible.

We applied ARMor to automatically prove the memory safety and the control flow integrity of ARM executables including our programs and MiBench programs [10]. The proven MiBench programs are BitCount and StringSearch. BitCount has 293 machine words in its code section, and StringSearch has 1104 machine words in its code section. It took 2.5 hours to prove BitCount and 8 hours to prove StringSearch on a 2.7 GHz Core i7 machine. These programs were compiled with GCC 3.3.2 with optimization level `-Os` and run on a development board based on a Philips LPC2129 processor, which implements the ARM7TDMI architecture. To the best of our knowledge, this is the first time that such realistic programs have been automatically verified in a high-order theorem prover, providing the highest-level guarantee that can be achieved by today's computer technologies.

7.1 Trusted Computing Base

Our TCB includes the formalization of safety properties, the definitions of the program logic, the formal semantics of ARM ISA, the HOL theorem prover and hardware. Among them, we contributed the first two, whose definitions are 58 lines in HOL.

We also proved useful inference rules and theorems, and these proof scripts are about 600 lines, but they are not in the TCB, because their correctness is guaranteed by the reasoning system of HOL. Our total HOL/SML scripts have about 8000 lines; about half supports proof automation, a quarter is the implementation of abstract interpretations, and the other quarter is supporting functions.

We compare ARMor with other work that uses sandboxing techniques to isolate untrusted binary code such as Gleipnir [1, 5], PittSFIeld [12], and Native Client [20, 26] in Table 1. Some of the projects are quite big, and we only compare the sandboxing parts in terms of the size of TCB and verification methods used.

The Gleipnir project developed CFI and XFI. For CFI, it performed theoretical analysis at the language level [2], describing formal semantics for a simplified instruction set and for attack models, with final theorems that established the correctness of its mechanisms. However, this work was checked with human endeavor on paper and not carried out for any sandboxed code, which means that any implementation must be trusted. XFI used a static verifier to check the presence of CFI and memory guards. The verifier was a 3000-line C++ program, which brings itself and a compiler in its TCB. PittSFIeld also used a verifier, but as an improvement in verification, it formalized semantics in ACL2 for a very small subset of instructions and for the verifier constraints, and under this semantics, it mechanically proved that its mechanisms could guarantee the confinement of untrusted code [11]. NativeClient also relies on its verifier to ensure safety, and many testing efforts were made to ensure the correctness of the verifier. Its verifier has 600 C statements for x86, and the size of verifiers for the ARM and x86-64 architectures was not reported.

It is noteworthy that only ARMor ensures formalized safety properties in binary code with an automatic machine-checked proof, and this insurance is deeply rooted in a formal realistic semantics of the ARM ISA.

	Gleipnir (CFI/XFI)	PittsFIeld	NativeClient	ARMor
TCB	verifier implementation and compilation (compiler)			formal definitions of safety properties and logic, ARM semantics and HOL
Size of verifier (definitions)	3000-line commented C++ (XFI)	N/A for verifier, 500-line ACL2	600 C statements for x86, unknown for ARM & x86-64	58-line HOL plus existing definitions for ARM semantics
Formal methods	human-checked proof at language level	machine-checked proof at language level	N/A	automatic machine-checked proof
Formalized elements	semantics for small subset of instructions and attack models	semantics for small subset of instructions and verifier constraints	N/A	safety properties and program logic plus existing ARM semantics

Table 1: Comparison of verification and TCB (hardware omitted)

7.2 Influence of Formalization

Formalizing safety properties and proving them for an executable expose a large amount of information about the executable, which gives us useful knowledge about ensuring the properties and helps us correct errors in the SFI implementation.

Simplifying Proof. We initially thought that proving memory safety and control flow integrity would be tricky, because they mutually depend on each other: the memory safety needs the control flow integrity to ensure that the store checks cannot be circumvented, while the control flow integrity depends on the memory safety to guarantee that the memory locations storing jump targets are not overwritten. In practice, introducing the control stack and giving a smaller writable data memory set `dm` as discussed in Section 4.2 are strong enough to prove the control flow integrity for most code cases in ARM executables. For example, targets of switch jumps are stored in the datapool, which is not included in the set of given writable addresses. As a result, after we formalize the datapool memory as a separate heap assertion, the control flow integrity of switch statements can be proven. Introducing the control stack solved the problems of overwriting return addresses. The solution of function pointers depends on how the pointers are used. If they are not meant to be changed after being placed in a table, we exclude these addresses from the given set of writable addresses and handle them the same way as for switches. If the function pointers are allowed to be overwritten with different values, a more complicated proof scheme is needed. So far, we have not considered this situation.

Locating Errors in SFI Implementation. ARMor is not designed to find bugs, but the failure of a proof reveals useful information about possible issues in binary code. An example is that the link register, R14, in the ARM ISA may be used as a scratch register in a computation. Our initial implementation of the control stack only considered loading values into PC. As a result, the control flow integrity assertion failed, when R14 was used as a scratch register. By looking at the values used in the assertion and that of the R14, we found the reason of failure and considered instructions that load a value into R14.

7.3 Overhead of Safety Checks

We measured the performance overhead of SFI implementations for those programs we proved, and it ranges from 5% to 240%. For example, BitCount has 10% slowdown, and StringSearch has 240% slowdown. The high overhead was caused by the address checking routine discussed in Section 3.2, because it is a rather lengthy function with several load and comparison instructions. We used this suboptimal implementation, because our goal was to provide a very high-confidence argument for strict memory safety and strict control flow integrity about binary code, not to reduce overhead; this routine was the most direct way to implement a check. In addition, we have not optimized the implementation. If we use alternative SFI implementations which provide less strict safety policies, we can reduce the overhead dramatically as illustrated in [20] and verify a less stringent safety requirement.

7.4 Future Work

In order to verify other safety properties by reusing ARMor's framework, we have parameterized the safe instruction semantics. This makes the framework capable of proving a class of safety properties that can be formalized at every instruction of a program. For a different safety property, we only need to formalize it in the antecedent of the safe instruction rule, and the rest of the framework stay unchanged.

The top-level program judgment is flat in terms of organizing code block judgments, and this can cause scalability issue when programs become bigger. We are working on a hierarchical structure for this layer, where the concept of function is introduced to group code blocks into tree-like proof units. This scheme can potentially verify more programs and reduce proof time.

8. RELATED WORK

SFI dates back to Wahbe et al. [24], and more recent implementations have low overhead [20]. However, existing SFI work does not formally verify a sandboxed binary code. This means that an implementation and its compilation must be fully trusted as discussed in Section 7.1.

Boyer and Yu made the first attempt to verify small real-world executables with symbolic execution, but their specifications and proofs were done manually [4]. Myreen et al. developed a traditional Hoare logic for machine code programs [15] and a decompiler to reuse proofs for multiple architectures [16]. Tan and Appel developed a compositional logic for reasoning about arbitrary control flows and proved typing rules for the foundational proof-carrying code project [22], but their logic has the complexities mentioned in Section 5.

Proof-carrying code uses a VCG-based approach to verify programs without formalizing the method itself, resulting in a large TCB which includes a verifier for a code consumer [17].

UU IR Author Manuscript

Shao's group developed XCAP to verify low-level code [18]. Their method is posterior in the sense that if there is a correct top-level code specification for the code of interest, then the method can verify its correctness by checking it at each instruction interactively. This is similar to the last step of ARMor, in which the specifications are instantiated and verified.

Seo et al. used the result of an abstract interpretation to guide the construction of Hoare logic proofs [21], but their abstract interpreter generated redundant information which was removed manually.

9. CONCLUSION

We developed ARMor: a sandboxing system for ARM binaries based on software fault isolation (SFI) that is suitable for use in small embedded systems. Fault isolation is achieved by rewriting object code to place a safety check in front of every potentially dangerous operation. The isolation guarantees provided by ARMor are formally verified in the HOL theorem prover in terms of a well-vetted formal semantics for the ARM instruction set architecture. Thus, ARMor's trusted computing base is extremely small, including only the specifications of memory safety and control flow integrity, a program logic, the instruction semantics, and HOL itself. We do not trust an operating system, compiler, binary rewriter, verifier or any other large object. ARMor's proofs are constructed automatically using a novel program logic framework that integrates results from Hoare-style reasoning and from an abstract interpreter.

10. ACKNOWLEDGMENTS

We thank Dominique Chanet and Jonas Maebe for their help in using the Diablo framework. We thank Magnus Myreen and Anthony Fox for their help in using the HOL theorem prover. This research was primarily supported by an award from DARPA's Computer Science Study Group.

11. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control flow integrity: Principles, implementations, and applications. In *Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Intl. Conf. on Formal Engineering Methods*, pages 111–124, 2005.

[3] C. L. Biffle. Undefined behavior in Google NaCl. `http://code.google.com/p/nativeclient/issues/detail?id=245`.

[4] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43:166–192, January 1996.

[5] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–32, 1967.

[7] A. Fox. Formal specification and verification of ARM6. In *Proc. of the 16th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 25–40, Rome, Italy, Sept. 2003.

[8] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proc. of the Intl. Conf. on Interactive Theorem Proving (ITP)*, Edinburgh, UK, July 2010.

[9] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of Workshop on Workload Characterization*, pages 3–14, Austin, TX, Dec. 2001. `http://www.eecs.umich.edu/mibench`.

[11] S. McCamant. A Machine-Checked Safety Proof for a CISC-Compatible SFI Technique. In *MIT CSAIL Technical Report*, May 2006.

[12] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. of the 15th USENIX Security Symp.*, Aug. 2006.

[13] T. F. Melham. A package for inductive relation definitions in HOL. In *Proc. of the 1991 Intl. Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357, 1992.

[14] M. O. Myreen, A. C. J. Fox, and M. J. C. Gordon. A Hoare logic for ARM machine code. In *Proc. of the IPM Intl. Symp. on Fundamentals of Software Engineering (FSEN)*, 2007.

[15] M. O. Myreen and M. J. C. Gordon. A Hoare logic for realistically modelled machine code. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 568–582, 2007.

[16] M. O. Myreen, K. Slind, and M. J. C. Gordon. Machine-code verification for multiple architectures—An application of decompilation into logic. In *Proc. of the Intl. Conf. on Formal Methods in Computer-Aided Design*, 2008.

[17] G. C. Necula. Proof-carrying code. In *Proc. of the 24th Symp. on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, Jan. 1997.

[18] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. of the 33rd Symp. on Principles of Programming Languages (POPL)*, pages 320–333, Charleston, SC, USA, Jan. 2006.

[19] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th IEEE Symp. on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[20] 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 *Proc. of the 19th USENIX Security Symp.*, Aug. 2010.

[21] S. Seo, H. Yang, and K. Yi. Automatic construction of hoare proofs from abstract interpretation results. In *Proc. of the 1st Asian Symp. on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.

[22] G. Tan and A. W. Appel. A compositional logic for control flow. In *Proc. of the 7th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 80–94, 2006.

[23] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: A reliable, retargetable and extensible link-time rewriting framework. In *Proc. of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, 12 2005.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, Dec. 1993.

[25] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[26] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53:91–99, Jan. 2010.

UU IR Author Manuscript