View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library
Author Manuscript
UU IR Author Manuscript
UU IR Author Manuscript

A Practical Logic Framework for Verifying Safety Properties of Executables

Lu Zhao Guodong Li John Regehr
University of Utah
{luzhao,ligd,regehr}@cs.utah.edu

***Abstract*—We present a novel program logic, $\mathcal{L}_f$, which is designed on top of a Hoare logic, but is simpler, more flexible and more scalable. Based on $\mathcal{L}_f$, we develop a framework for automatically verifying safety properties of executables. It utilizes a whole-program interprocedural abstract interpretation to automatically discover the specifications needed by $\mathcal{L}_f$ to prove a program judgment. We implemented $\mathcal{L}_f$ and the framework in the HOL theorem prover.**

I. INTRODUCTION

It is challenging to formally reason about executable code, because it lacks high-level abstractions such as types, data structures and structured control flow. Some research has addressed various issues in formally verifying machine code or low-level code [2], [14], [15], [17], [20], [22], [24]. However, it is still impractical or very inefficient to verify some critical safety properties about executables emitted by a production compiler such as GCC using theorem proving.

We present a novel program logic framework for automatically proving safety properties about real-world executables. Its theoretical foundation is a new program logic, $\mathcal{L}_f$, designed on top of a Hoare logic. A Hoare logic describes code with a triplet judgment: $\{p\}\ c\ \{q\}$, where p and q are the pre- and post-conditions of code c, specifying the states before and after the execution of the code, respectively [7], [10]. The reasoning process is to compose the judgment of a piece of code from judgments of smaller codes using inference rules, until the judgment of an entire program is achieved.

It is simple to compose judgments of sequential code with a sequencing rule [7]. However, it is difficult to handle unstructured control flows, because for each type of control flow transfers, it requires proving some inference rules and using them interactively [13]. For example, in order to specify a loop, it requires a loop rule and a termination proof; this is not acceptable for reasoning about embedded code where control loops are often infinite. In addition, the goal of composing a monolithic judgment for an entire program has not been shown useful in practice, because it is not scalable.

Our idea is keeping the simple part of a Hoare logic: composition of *code blocks*, which only have sequential control flows, such as a basic block or a super block; beyond code blocks, we design a novel scalable logic structure which does not require compositional rules and which makes it possible to automate the entire proof of safety properties. This resulted in $\mathcal{L}_f$ (a logic with hierarchical function judgments).

The structure of $\mathcal{L}_f$ has three layers. The bottom layer is instruction semantics. In principle, any sound formal semantics works, and there exists independently developed, well-vetted, realistic formal semantics for common instruction set architectures, such as ARM [4], [5] and x86 [18]. The middle layer is Hoare judgments of code blocks. We implement this layer by a Hoare logic that has compositional rules for code blocks.

The top layer, as the core of $\mathcal{L}_f$, is hierarchical function judgments. A function is roughly equivalent to a function constructed by a binary rewriting/analysis tool. Such tool can decompile an executable, construct a conservative control flow graph, and build functions by using call/return conventions, besides other functionalities [11], [23]. A function has code blocks and calls to other functions. We abstract the calls by *well-formed nodes*, which also have a precondition and a postcondition: the precondition is the initial condition of a callee, and the postcondition is the condition that holds when the callee returns. We define the relationship among the Hoare judgments of the code blocks and the callee nodes: for each node, the postconditions of its predecessors imply its precondition. The final judgment of a program is the judgment of the top-level or entry function.

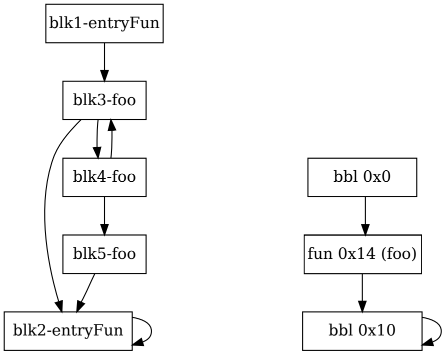
The hierarchical function judgments have several advantages over a traditional Hoare logic. First, it does not compose over code blocks, so it does not need rules for loops or any arbitrary jumps. Nor a termination proof. The definition of the function judgment handles both infinite and finite loops. Second, it naturally divides an executable into object-code functions, and the reasoning process examines one function at a time. As a result, it easily scales to an entire program. Third, it facilitates proof automation in two ways. One is the composition of judgments of code blocks, whose sequential structure is simple enough that we automate the composition by meta-language programming. The other is utilizing interprocedural abstract interpretations to automatically discover the relationship among Hoare judgments in verifying shallow safety properties.

The first two layers of $\mathcal{L}_f$ have been well studied in literature, and in this paper, we focus on the hierarchical function judgments (Section II) and an application of $\mathcal{L}_f$: automatic verification of safety properties with assistance of interprocedural abstract interpretations (Section III).

INSTITUTIONAL REPOSITORY THE UNIVERSITY OF UTAH
University of Utah Institutional Repository
Author Manuscript

```
<entryFun>
blk1: (0x0, 0xE3A0D441) //mov r13,#0x41000000
      (0x4, 0xE3A00000) //mov r0,#0
      (0x8, 0xE1A01000) //mov r1,r0
      (0xC, 0xEB000000) //bl foo (branch to foo)
blk2: (0x10,0xEAFFFFFE) //b +#0 (branch to blk2)
<foo>
blk3: (0x14,0xE2411001) //sub r1,r1,#0x1
      (0x18,0xE3320101) //teq r2,#0x40000000
      (0x1C,0x11A0F00E) //movne pc,r14 (ret neq)
blk4: (0x20,0xE5C21000) //strb r1,[r2] (str byte)
      (0x24,0xE3310000) //teq r1,#0x0 (test eq)
      (0x28,0x1AFFFFF9) //bne foo (branch neq)
blk5: (0x2C,0xe1a0f00e) //mov pc,r14 (return)
```

(a) Program code



(b) Program CFG (c) Abstracted function nodes

Fig. 1. An example

A. An Example

To explain $\mathcal{L}_f$ and to show how it works, we take the example of proving *memory safety*: store operations are confined to pre-defined regions, and *control flow integrity*: execution may not escape a pre-determined control flow graph (CFG), about the ARM executable shown in Figure 1.

For memory safety, we want to prove that it only writes to a memory section `mem` $= \{a | 0x40000000 \leq a \land a < 0x41000100\}$. For control flow integrity, we need a CFG policy (Figure 1.b). The CFG policy of an executable may be computed by a binary analysis tool, or by hand. How the CFG policy is computed is irrelevant to $\mathcal{L}_f$; what is critical is that $\mathcal{L}_f$ verifies that the executable respects the given CFG policy. We model the CFG policy by a function `succ`: given an address of an instruction, it returns the set of addresses where the control goes. Suppose the given CFG policy of the example is (computed by hand): `succ` $i = \{0xC \to \{0x14\}, 0x10 \to \{0x10\}, 0x1C \to \{0x20, 0x10\}, 0x28 \to \{0x14, 0x2C\}, 0x2C \to \{0x10\}, i \to \{i + 0x4\}\}$, where $\to$ means a parameter-return value pair. For brevity, we use $i \to \{i + 0x4\}$ to denote the PC relation inside a basic block without explicitly listing them (it is only used after other lookup fails).[1]

For this program, $\mathcal{L}_f$ establishes that the strb instruction in blk4 does not violate the memory safety, and the two indirect jumps at 0x1C and 0x2C follow the given CFG policy. We first compose the Hoare judgments of code blocks, and the results show what conditions are required in order for a block to execute safely. For example, blk4's precondition requires $\{r2\} \subseteq$ `mem`, and blk5 requires $r14 \in$ `succ`$(0x2C)$ (similar to blk3). Next, we reason about the relationship among the Hoare judgments of block 3, 4 and 5 (of function `foo`). We follow and verify the CFG of the function and derive the fact that the postcondition of blk3 (from 0x1C to 0x20) satisfies that register R2 has value $0x40000000$, which discharges the condition of blk4. However, the two assertions of the control flow integrity of blk3 and blk5 can not be discharged inside `foo`, and they are propagated to the precondition of `foo` by an abstract interpretation. We next abstract the function into a Hoare judgment, which can be fed into its caller, `entryFun`, and which behaves like the judgments of blk1 and blk2 (Figure 1.c). We reason about the relationship among the three Hoare judgments of `entryFun`: of blk1, of blk2, and of the abstracted `foo`, and we are able to discharge the two assertions of `foo`. The final result is a program judgment:

```
PROG_SPEC SAFE_INS entryFun 0x0 pred bspec
```

Informally, it means that the program holds its memory safety and control flow integrity with respect to a given specification. We will discuss in detail how we formally reach this conclusion (Section IV).

It is worth to point out that verifying memory safety is not always possible in real-world executables, because telling that every store operation is confined to pre-defined regions is undecidable in general. In order to make proof possible in practice, we insert necessary dynamic checks before dangerous stores whose addresses can not be determined statically, and this instrumentation is done by transforming an executable with a binary rewriter. We also get the CFG policy of the rewritten executable from the rewriter. These practical considerations affect if a proof attempt succeeds. For instance, if there was not blk3 in Figure 1.a, our proof of the memory safety would simply fail (in fact, any kind of verification attempt would fail), and we could not derive the above program judgment. However, they are not irrelevant to how $\mathcal{L}_f$ works, and we have omitted them.[2]

We implemented $\mathcal{L}_f$ and the framework for verifying safety properties in the HOL theorem prover [8]; we automatically proved the memory safety and control flow integrity of rewritten ARM executables, including some MiBench programs [9].

II. $\mathcal{L}_f$

We design the first layer of $\mathcal{L}_f$ to utilize an existing formal ARM semantics, and it comes in as proven Hoare triples in the HOL theorem prover [5], [13]. Figure 2 shows the semantics for a store instruction: strb r1, [r2]. It says that after execution of the instruction, the value at memory address r2 is updated to the least significant byte of r1 (`w2w` converts a 32-bit word into an 8-bit word), and the program counter (PC) increased

---

[1] If the given CFG policy is too small, or does not coincide with the executable, a proof attempt will fail.

[2] Determining which stores are dangerous and inserting dynamic checks for them are challenging by themselves and deserve their own research.

$$
\begin{aligned}
&\{\mathtt{PC}\ p * \mathtt{R}\ 2\ r2 * \mathtt{R}\ 1\ r1 * \mathtt{MEMORY}\ dom\ f * \langle r2 \in dom\rangle\}\\
&\ (p, \mathtt{0xE5C21000})\ \text{// strb r1,[r2]}\\
&\{\mathtt{PC}\ (p+4) * \mathtt{R}\ 2\ r2 * \mathtt{R}\ 1\ r1 *\\
&\qquad \mathtt{MEMORY}\ dom\ \left((r2 \mapsto (\mathtt{w2w}\ r1))\ f\right)\}
\end{aligned}
$$

Fig. 2. Axiomatic semantics of strb r1, [r2]

by 4. From our perspective, those theorems are equivalent to instruction axioms in an axiomatic semantics, since we take those theorems as granted and build our logic on top of it. This semantics has some important properties which we summarize below. For a full treatment, interested readers may refer to the references mentioned above.

- Machine states include registers, memory cells, status flags, and the current program status register. For example, $\mathtt{PC}\ p$ in the precondition of Figure 2 asserts that the program counter has value p and that p is word-aligned; $\mathtt{R}\ 2\ r2$ and $\mathtt{R}\ 1\ r1$ assert that registers R2 and R1 have values $r2$ and $r1$, respectively; $\mathtt{MEMORY}\ dom\ f$ asserts that some set of memory addresses dom has value f (these are symbolic values). The $\mapsto$ operator in the meta logic updates a function on a given value while keeping other values unchanged, whose definition is: $a \mapsto b = \lambda f\ c.$ (if $a = c$ then b else $f\ c$). Other machine state assertions include $\mathtt{S}\ t\ v$: one of the status flags t (carry $\mathtt{sC}$, negative $\mathtt{sN}$, overflow $\mathtt{sV}$, or zero $\mathtt{sZ}$) has value v.
- The $*$ operator is a version of separating conjunction which has important properties as in the separation logic [19]: (1) a triple only asserts the change of a local state (the parts of state that are used by the instruction), and a global version may be achieved by using the Frame rule; (2) if a separating conjunction expression asserts a machine resource more than once (excluding a pure assertion), then its value is $\mathtt{false}$.
- $\langle\rangle$ 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. $\langle r2 \in dom\rangle$ says that $r2$ has to be in the domain of the memory function f in order for this transition to take place.
- The pair, $(p, \mathtt{0xE5C21000})$, represents code assertion for the instruction, meaning that the value $\mathtt{0xE5C21000}$ is stored at the memory address of p.
- Some boolean operators such as implication ($\Rightarrow$) and disjunction ($\vee$) are lifted to the separating conjunction level. For example, $p \overset{*}{\Rightarrow} q$ means $\lambda s.\ (p\ s \Rightarrow q\ s)$; $p \veebar q$ is $\lambda s.\ (p\ s \vee q\ s)$.

A. Label Predicates

The assertion language in $\mathcal{L}_f$ is *a set of label predicates*. A label predicate is a pair of a label (instruction address) 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

$$
\begin{array}{cccll}
lp & \in & LabelPred & = & LabelExp \times StatePred\\
l & \in & LabelExp & = & \text{word32}\\
p & \in & StatePred & = & \text{separating conjunction expression}
\end{array}
$$

Its interpretation is defined by a semantic functions $\mathsf{LP2SP}$, and another function, LPSET, interprets a set of label predicates:

$$
\begin{aligned}
&\mathsf{LP2SP}\ (l,\ p) = \mathsf{PC}\ l\ * p\\
&\mathsf{LPSET}\ P\ =\ \lambda s.\ (\exists lp.\ lp \in P \wedge (\mathsf{LP2SP}\ \ lp)\ s)
\end{aligned}
$$

Let $\overset{lp}{\Rightarrow}$ be the subsumption relation between two sets of label predicates:

$$P \overset{lp}{\Rightarrow} Q\ \textbf{iff}\quad (\mathsf{LPSET}\ P) \overset{*}{\Rightarrow} (\mathsf{LPSET}\ Q)$$

We define the following rule in order to use the existing semantics in $\mathcal{L}_f$, and it converts an instruction axiom to another theorem that uses the syntax of label predicates. The reason for the conversion is that it is easy for the meta logic to operate on a pair, but it is difficult to operate on the $*$ operator. Another advantage of label predicates will become clear when we define the function judgment later.

$$\frac{\{\mathsf{PC}\ l * p\}\ ins\ \{\mathsf{PC}\ l' * q\}}{\mathtt{ORG\_INS}\ (l,\ p)\ ins\ (l',\ q)}\ \text{Ins}$$

B. A Hoare Logic

The middle layer of $\mathcal{L}_f$ is a multi-entry multi-exit Hoare judgment with a set of label predicates as its assertion language. We use a $\mathtt{step}$ relation to bridge state transitions in $\mathcal{L}_f$ and the existing axiomatic semantics.

$$\mathtt{step}\ \ ir\ i\ s\ t\ \textbf{iff}\ \ \exists p\ q.\ (ir\ p\ i\ q) \wedge (\mathsf{LP2SP}\ p)\ s \wedge (\mathsf{LP2SP}\ q)\ t$$

where ir is a parameterized relation of instruction transition, namely, an instruction semantics. It can be the existing instruction semantics introduced above ($\mathtt{ORG\_INS}$), or an augmented version for proving safety properties (to be described later). It says that a transition from state s to state t by instruction i under a given semantics is equivalent to a transition from s to t made by the instruction under the semantics.

Based on the $\mathtt{step}$ relation, we implemented a Hoare logic. Because Hoare logic has been well studied in literature, we have omitted its details. We summarize the important results below. Interested readers may refer to [6], [14] for detailed discussions.

- We write a Hoare judgment as $\mathtt{SPEC}\ ir\ \{P\}\ C\ \{Q\}$, where ir is the semantic parameter discussed above, P and Q are sets of label predicates, and C is a set of labeled instructions. Its interpretation is that if there exists a true label predicate in the precondition, then there exists a true label predicate in the postcondition some steps later.
- We proved many useful inference rules including those for composing code blocks, such as sequencing, frame, and strengthen, etc. For example, the LPMerge rules combine and split label predicate entries which have the same label (Figure 3.a. We have omitted the leading $\mathtt{SPEC}$).
- The Hoare logic has only one role in $\mathcal{L}_f$: composing judgments of code blocks, because it is simple and can be

INSTITUTIONAL REPOSITORY
THE UNIVERSITY OF UTAH
University of Utah Institutional Repository
Author Manuscript

$$ir\ \{P \cup \{(l,p)\} \cup \{(l,q)\}\}\ C\ \{Q\} = ir\ \{P \cup \{(l, p \veebar q)\}\}\ C\ \{Q\}$$
$$ir\ \{P\}\ C\ \{Q \cup \{(l,p)\} \cup \{(l,q)\}\} = ir\ \{P\}\ C\ \{Q \cup \{(l, p \veebar q)\}\}$$

(a) LPMerge rules

$$\{(0x20, \mathtt{MEMORY}\ dom\ df * \langle r2 \in dom\rangle * \langle r1 \neq 0\rangle * \mathtt{S}\ \mathtt{sZ}\ z * a1)\}$$
$$\text{blk4} \tag{1}$$
$$\{(0x14, \mathtt{MEMORY}\ dom\ \ ((r2 \mapsto (\mathtt{w2w}\ r1))\ df) * a2)\}$$
$$a1 = \mathtt{R}\ 2\ r2 * \mathtt{R}\ 1\ r1, \qquad a2 = \mathtt{S}\ \mathtt{sZ}\ (r1 = 0) * a1$$

$$\{(0x20, \mathtt{MEMORY}\ dom\ df * \langle r2 \in dom\rangle * \langle r1 = 0\rangle * \mathtt{S}\ \mathtt{sZ}\ z * a1)\}$$
$$\text{blk4} \tag{2}$$
$$\{(0x2C, \mathtt{MEMORY}\ dom\ \ ((r2 \mapsto (\mathtt{w2w}\ r1))\ df) * a2)\}$$

(b) The Hoare judgments of blk4

Fig. 3. Hoare logic rules and judgments

automated by meta-language programming. For example, Figure 3.b gives the Hoare judgments of blk4 of Figure 1.a.[3] Blk4 has two separate judgments with each for a branch condition, and the branch conditions, $\langle r1 \neq 0\rangle$ and $\langle r1 = 0\rangle$, originate from the branch instruction `bne foo`, which has two separate axioms [13]. The value of the `sZ` flag is set to $(r1 = 0)$ in the postconditions (we have omitted the assertions for other status flags).

- After composition, the assertion of a safety property is "pushed up" to the precondition of the code block containing the instruction, becoming the block's condition, e.g. $\langle r2 \in dom\rangle$ is now an assertion of the judgments of blk4. Branch conditions are a little different, because when we merge Judgments 1 and 2 with LPMerge rules, the two branch conditions become tautology $\langle (r1 \neq 0) \vee (r1 = 0)\rangle$ and can be removed from the merged judgment.

C. *Well-Formed Hoare Judgments*

In order to model a code block which has only one entry address, we define a *well-formed* Hoare judgment as a single-entry multi-exit Hoare judgment by imposing two constraints: (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

$$\mathtt{WF\_SPEC}\ ir\ P\ C\ Q\ \textbf{iff}$$
$$(\mathtt{SPEC}\ ir\ \{P\}\ C\ \{Q\}) \wedge (\forall (l,p) \in P.\ \ l = \mathtt{L}(C))$$

where $\mathtt{L}(C)$ returns the entry address of a code block.

D. *Hierarchical Function Judgments*

The central structure of $\mathcal{L}_f$ is recursive function judgments. The idea is that a function consists of code blocks and function calls; code blocks are specified by the well-formed Hoare judgment described above; we abstract a callee as a well-formed node, which behaves like a well-formed Hoare judgment in the caller, having a single-entry precondition, abstract code and a postcondition. We specify the relationship among these Hoare judgments as the following: for each node, the postcondition of its predecessor "implies" its precondition. The implication idea comes from Floyd's inductive assertion [3], and we formalize it here in order to define the function judgment:

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

It reads that a set of label predicates Q implies another set of label predicates R (at the function 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.

1) Function Judgments: We define the function judgment in Figure 4.a, where wf is a well-formed node relation, and it includes Hoare judgments of code blocks and Hoare abstractions of function calls. Figure 4.b defines such a relation. Ir is the parameterized instruction semantics discussed before, and $prog$ is a set of nodes of a function, including nodes of code blocks and nodes of callee abstractions. The definition requires that each node is well-formed (the second to the last line). $Entry$ is the entry address of the function, and $init$ is the initial condition of the function. $Exits$ is a set of pairs with each pair being an exit node and its associated exit condition. $Predecessor$ models the CFG policy at the node level by a function: given a node, it returns the set of predecessor nodes. $Bspec$ and $kspec$ are two specifications for all nodes of the function; the former is a mapping from nodes to their preconditions, and the latter is a mapping from nodes to their postconditions. The last line of the definition requires that if a node is a predecessor of another node, then the postcondition of the former implies the precondition of the latter. The first line of the definition body specifies that the initial condition of the function subsumes the $bspec$ at the entry node, and the second line stipulates that for every exit node, its $kspec$ subsumes the exit condition associated with that node. In a simple case, $\{(entry, init)\}$ is $(bspec\ (\mathtt{bbl}\ entry))$, and $(kspec\ e)$ is q.

`Bbl` is one of the two constructors for a user-defined data type `fun_node`, which represents the code of a code block or a function by its entry label:

$$\mathtt{bbl,fun\!:}\ \text{word32}\ \rightarrow \mathtt{fun\_node}$$

We use two constructors for human readability indicating that a node is a code block or a function abstraction; from the perspective of a type system, one constructor is enough.

2) Well-formed Nodes: The concept of a well-formed node plays a very important role in $\mathcal{L}_f$, and we define it in Figure 4.b by using the inductive relation definition of the meta-logic [12]. The Base rule says that the well-formed Hoare judgment of a code block is a well-formed node. The Induction rule says that from a function judgment (whose nodes are well-formed), we can get a new well-formed node whose precondition is the initial condition of the function, and whose postcondition is the big union of its exit conditions. `image` is a function defined as `image` $f\ s = \{f\ x | x \in s\}$, and `snd`

---

[3] A Hoare triple is written as $\{P\}\ C\ \{Q\}$, and our P, C and Q are sets which also use braces by convention. For clarity, we only use one pair of braces in writing pre- and post-conditions and do not use braces for the code.

$$\begin{array}{l}
\texttt{FUN\_SPEC}\ \mathit{wf}\ \mathit{ir}\ \mathit{prog}\ \mathit{entry}\ \mathit{init}\ \mathit{exits}\ \mathit{predecessor}\ \mathit{bspec}\ \mathit{kspec}\ \textbf{iff} \\
\quad (\{(\mathit{entry}, \mathit{init})\} \overset{lp}{\Rightarrow} (\mathit{bspec}\ (\texttt{bbl}\ \mathit{entry})))\wedge \\
\quad (\forall (e, q) \in \mathit{exits}.\ (\mathit{kspec}\ e) \overset{lp}{\Rightarrow} q)\wedge \\
\quad \forall \mathit{node} \in \mathit{prog}. \\
\qquad (\mathit{wf}\ \mathit{ir}\ (\mathit{bspec}\ \mathit{node})\ \mathit{node}\ (\mathit{kspec}\ \mathit{node}))\wedge \\
\qquad (\forall \mathit{pre} \in (\mathit{predecessor}\ \mathit{node}).\ (\mathit{kspec}\ \mathit{pre}) \overset{P}{\Rightarrow} (\mathit{bspec}\ \mathit{node}))
\end{array}$$

(a) Function judgment

$$\frac{\texttt{WF\_SPEC}\ \mathit{ir}\ \{(l, p)\}\ C\ \{Q\}}{\texttt{WF\_NODE}\ \mathit{ir}\ \{(l, p)\}\ (\texttt{bbl}\ l)\ \{Q\}}\ \text{Base}$$

$$\frac{\texttt{FUN\_SPEC}\ \texttt{WF\_NODE}\ \mathit{ir}\ \mathit{prog}\ \mathit{entry}\ \mathit{init}\ \mathit{exits}\ \mathit{predecessor}\ \mathit{bspec}\ \mathit{kspec}}{\texttt{WF\_NODE}\ \mathit{ir}\ \{(\mathit{entry}, \mathit{init})\}\ (\texttt{fun}\ \mathit{entry})\ (\bigcup(\texttt{image}\ \texttt{snd}\ \mathit{exits}))}\ \text{Induction}$$

(b) Well-formed node

$$\begin{array}{l}
\texttt{PROG\_SPEC}\ \mathit{ir}\ \mathit{prog}\ \mathit{entryProg}\ \mathit{predecessor}\ \mathit{bspec}\ \textbf{iff} \\
\qquad \exists \mathit{kspec}\ \mathit{exits}.\ \texttt{FUN\_SPEC}\ \texttt{WF\_NODE}\ \mathit{ir}\ \mathit{prog}\ \mathit{entryProg}\ (\lambda s.\texttt{T})\ \mathit{exits}\ \mathit{predecessor}\ \mathit{bspec}\ \mathit{kspec}
\end{array}$$

(c) Program judgment as the judgment of the top-level function

Fig. 4. Definitions of function judgments

returns the second element of a tuple. In the call graph of a program, the leaf functions, which do not have a callee, only have the `bbl` nodes; other functions have both `bbl` and `fun` nodes (Figure 1.c).

Figure 4.c defines the judgment of a program, which is simply the judgment of the top-level function. We have simplified the initial condition to $(\lambda s.\texttt{T})$ by focusing on the predicate of states instead of the contents of states.

3) Soundness: Our soundness proof says that a program never gets stuck under a given semantics throughout its execution. An intuitive argument is that when control reaches the end of a code block, it resumes on one of its successor blocks (including jumping to the entry block of another function) because of the implication relation. Formally, we may derive a function specification `FUN_SPEC` if and only if: starting from its initial state s, if the execution reaches the label of a code block, $\texttt{L}(n)$, then the precondition defined by bspec on the block is ensured to be true. We have omitted the theorem itself, since it uses some definitions of our Hoare logic which we did not show (We will have a detailed technical report available on-line if this paper is published).

III. AUTOMATIC VERIFICATION OF SAFETY PROPERTIES

We describe a specific application of $\mathcal{L}_f$: verifying safety properties. We present a framework that takes advantage of the hierarchical structure of $\mathcal{L}_f$ and that utilizes an interprocedural abstract interpretation to automate the verification. For the example of Figure 1, we first make assertions about the memory safety and the control flow integrity by defining a safe instruction semantics `SAFE_INS` and use it to instantiate the semantic parameter ir.

A. Safe Instruction Semantics

We augment an exiting instruction axiom to the following by asserting the safety properties mentioned before:

$$\begin{array}{l}
\{\texttt{PC}\ l * \texttt{MEMORY}\ \texttt{mem}\ \mathit{df} * \texttt{MEMORY}\ \texttt{cm}\ \mathit{cf} * \\
\qquad \langle \texttt{ms}(\mathit{ins}) \subseteq \texttt{mem}\rangle\ * \langle l' \in \texttt{succ}(l)\rangle * p\} \\
\quad (l, \mathit{ins}) \\
\{\texttt{PC}\ l' * \texttt{MEMORY}\ \texttt{mem}\ \mathit{df}' * \texttt{MEMORY}\ \texttt{cm}\ \mathit{cf} * p'\}
\end{array} \tag{3}$$

where l is the value of the PC, p represents other assertions that are not explicitly written out, and corresponding values in the postcondition are marked with a prime $'$.

1) Safety Assertions: Recall that `mem` is the set of predefined memory region mentioned in Section I-A, and `succ` is the CFG policy. $\texttt{ms}(\mathit{ins})$ is the set of memory addresses that an instruction, ins, writes to. $\langle \texttt{ms}(\mathit{ins}) \subseteq \texttt{mem}\rangle$ is the assertion for memory safety, and $\langle l' \in \texttt{succ}(l)\rangle$ is the assertion for the control flow integrity. The memory assertion is true for non store instructions, because $\texttt{ms}(\mathit{ins}) = \{\}$. $\texttt{MEMORY}\ \texttt{cm}\ \mathit{cf}$ asserts the data pool of ARM executables. A data pool is a set of memory addresses in the text section for storing constants, and our augmented theorem says that it cannot be changed ($\mathit{cf}' = \mathit{cf}$). The purpose of modeling the data pool is that some constants are useful in proving some properties. Figure 5.a shows the augmented theorem for the axiom in Figure 2.

2) The Safe Instruction Rule: We define a safe instruction rule, whose antecedent is the augmented theorem, and whose conclusion is a new relation `SAFE_INS`:

$$\frac{\text{theorem 3}}{\begin{array}{l}
\texttt{SAFE\_INS}\ (l,\ \texttt{MEMORY}\ \texttt{mem}\ \mathit{df} * \texttt{MEMORY}\ \texttt{cm}\ \mathit{cf} * \\
\qquad \langle \texttt{ms}(\mathit{ins}) \subseteq \texttt{mem}\rangle\ * \langle l' \in \texttt{succ}(l)\rangle * p) \\
\quad (l,\ \mathit{ins}) \\
\quad (l', \texttt{MEMORY}\ \texttt{mem}\ \mathit{df}' * \texttt{MEMORY}\ \texttt{cm}\ \mathit{cf} * p')
\end{array}}\ \text{SafeIns}$$

INSTITUTIONAL REPOSITORY THE UNIVERSITY OF UTAH
University of Utah Institutional Repository
Author Manuscript

$$\{\mathtt{PC}\ p * \mathtt{MEMORY}\ \mathtt{mem}\ df * \mathtt{MEMORY}\ \mathtt{cm}\ cf *$$
$$\langle \mathtt{ms}(\mathtt{0xE5C21000}) \subseteq \mathtt{mem}\rangle\ * \langle (p+4) \in \mathtt{succ}(p)\rangle * a1\}$$
$$(p, \mathtt{0xE5C21000})\ \text{// strb r1,[r2]}$$
$$\{\mathtt{PC}\ (p+4) * \mathtt{MEMORY}\ \mathtt{mem}\ ((r2 \mapsto (\mathtt{w2w}\ r1))\ df) *$$
$$\mathtt{MEMORY}\ \mathtt{cm}\ cf\ \ * a1\}$$
$$a1 = \mathtt{R}\ 2\ r2 * \mathtt{R}\ 1\ r1$$

(a) The augmented semantics of strb r1, [r2]

$$\mathtt{SAFE\_INS}\ (p, \langle (p+4) \in \mathtt{succ}(p)\rangle * \langle \{r2\} \subseteq \mathtt{mem}\rangle * \cdots)$$
$$(p, \mathtt{0xE5C21000})\ \text{// strb r1,[r2]}$$
$$(p+4, \cdots)$$

(b) Safe instruction semantics of strb r1, [r2]

Fig. 5. Safe instruction rule in action

This rule is critical, because if we directly use an instruction semantic with safety assertions in a logic, when the safety assertions are simplified to `true` and removed from the precondition, it is not clear what causes the absence of the assertions: that the axiom does not have the assertions at all, or that they have been discharged. With the new relation, `SAFE_INS`, we are always assured that they have been discharged; there is no instruction without having the safety assertions in this relation.

After applying this rule to the augmented theorem in Figure 5.a, we get the safe instruction semantics for the store strb r1, [r2], in Figure 5.b (We have omitted the assertions for memory, R2 and R1, since they are the same as in (a)).

It is noteworthy that this rule also provides flexibility in proving safety properties. For example, if we want to prove a different property, say, memory reads being confined to predefined regions, then we only need to formalize it as assertions in the augmented theorem. All the proven rules and definitions stay unchanged.

3) Instantiating the Semantic Parameter: We use the `SAFE_INS` relation to instantiate the semantic parameter ir in $\mathcal{L}_f$. This instantiation means that every instruction of a program over all possible executions has been asserted for the safety properties defined by the SafeIns rule. For example, the program judgment in Section I-A has this relation, indicating that every instruction of the program has been asserted for memory safety and control flow integrity.

By the definition of `PROG_SPEC SAFE_INS`, proving it boils down to finding the $\overset{P}{\Rightarrow}$ relation among nodes inside functions, which in turn reduces to finding global invariants that can discharge the safety assertions that the `SAFE_INS` relation has. There are two processes in $\mathcal{L}_f$ that discharge these safety assertions. One is the composition process (Section II-B). For example, the assertion $\langle (p+4) \in \mathtt{succ}(p)\rangle$ in Figure 5.b can be discharged for the instruction at address 0x20 (Figure 1.a) after we instantiate p to $0x20$ in composing. For the assertions that cannot be discharged by composition, they are pushed up to the precondition of the Hoare judgments of code blocks, and we use another method presented below to discharge them.

B. Interprocedural Safety Assertion Analysis

This is a backward context-sensitive and flow-sensitive analysis. Its domain is the power set of all concrete safety assertions occurring in the Hoare judgments of code blocks. It runs on a function at a time and computes, for a set of incoming safety assertions, the set of assertions that goes out of the function. A function has two types of nodes: *block nodes*: Hoare judgments of code blocks, and *function nodes*: abstract nodes for function calls. The transfer function works differently on a block node and a function node. For a block node, it runs as follows: when a node has an incoming safety assertion, it tries to derive the assertion from the label predicates in the postcondition whose labels are 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. For a function node, it suspends the computation in the caller and "dives into" the code of the callee. It merges the incoming assertions to the in-configuration of each exit node of the callee and computes the outgoing assertions for the callee. It takes the assertions going out of the callee as the new configuration of the function node and resumes the analysis in the caller. If the callee has other callees, it recursively dives into these callees to compute their outgoing configurations.

In simplified pseudo-code, the transfer functions are (Σ_{in} and Σ_{out} are the in- and out-configurations of a function):

```
transfer_block ($bnode, \Sigma_{in}$):
   foreach ($l, assert$) in $\Sigma_{in}(bnode)$
      foreach ($l', p$) $\in$ postcondition($bnode$)
         if $l' = l$ and (not ($p$ implies $assert$)) then
            $\Sigma_{out}(bnode)$ = $\Sigma_{out}(bnode) \cup \{(\mathtt{L}(bnode), assert)\}$

transfer_fun ($fnode, \Sigma_{in}$):
   merge $\Sigma_{in}(fnode)$ to the in-configuration
        of exit nodes of function of $fnode$ (fun_of_fnode);
   compute the states of fun_of_fnode until fixed point;
   $\Sigma_{out}(fnode) =$ the out-configuration of the entry
                    node of fun_of_fnode
```

This algorithm computes the global invariants—where a safety assertion can be discharged—in depth first search. In theory, it is exponential, but in practice, we use a cache for each function that records the outgoing assertion for a given incoming assertion. This makes an assertion traverse a function only once, reducing the complexity to polynomial. Our implementation also records or computes the following information:

- The location where a safety assertion is originated in a context-sensitive call graph and the path it traverses through;
- the location where a safety assertion gets discharged;
- for each block node, which safety assertions traverse along which call paths and their conversion theorems (equations that connect incoming assertions to corre-

sponding outgoing assertions).

This information is necessary for later proof automation that constructs function abstractions in depth first search for the top-level function. The automation is implemented by meta-language programming, in which we take the safety assertions that are propagated by a block node and use the Frame rule to add them to the node, generating a context-sensitive judgment along a call path. The framed Hoare judgments are able to imply the precondition of its successor nodes from their postcondition.

IV. PROVING THE EXAMPLE

We illustrate the verification process by proving the example in Figure 1. First, we compose Hoare judgments of code blocks by instantiating the semantic parameter ir with `SAFE_INS` and use the Frame rule to convert the local judgments to the global version. The results are shown below. For clarity, we have omitted the leading relation marker `SPEC SAFE_INS`. In addition, we have not explicitly written out unchanged assertions and less important assertions such as assertions of status flags; they are represented by $\cdots$.

$$\begin{array}{l}\{(0x0, \mathtt{REG}\ rf * \cdots)\}\\ \quad \text{blk1}\\ \{(0x14, \mathtt{REG}\ rf' * \cdots)\}\end{array} \quad (4) \qquad \begin{array}{l}\{(0x10, \mathtt{REG}\ rf * \cdots)\}\\ \quad \text{blk2}\\ \{(0x10, \mathtt{REG}\ rf * \cdots)\}\end{array} \quad (5)$$

where `REG` rf collectively asserts the values of registers from R1 to R14 (similar to `MEMORY`), and $rf' = ((R14 \mapsto 0x10)\ ((R0 \mapsto 0)\ ((R1 \mapsto 0)\ ((R13 \mapsto 0x41000000)\ rf))))$.

$$\begin{array}{l}\{(0x14, \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ z * \langle rf\ \ R14 \in \mathtt{succ}(0x1C)\rangle *\\ \qquad \langle rf\ \ R2 \neq 0x40000000\rangle * \cdots)\}\\ \quad \text{blk3}\\ \{(rf\ \ R14, \mathtt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * s1 * \cdots)\}\\ s1 = \mathtt{S}\ \mathtt{sZ}\ ((rf\ \ R2) = 0x40000000)\end{array} \tag{6}$$

$$\begin{array}{l}\{(0x14, \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ z * \langle rf\ \ R2 = 0x40000000\rangle * \cdots)\}\\ \quad \text{blk3}\\ \{(0x20, \mathtt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * s1 * \cdots)\}\end{array} \tag{7}$$

$$\begin{array}{l}\{(0x20, \mathtt{MEMORY}\ \mathtt{mem}\ df * \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ z *\\ \qquad \langle \{rf\ \ R2\} \subseteq \mathtt{mem}\rangle * \langle rf\ \ R1 \neq 0x0\rangle * \cdots)\}\\ \quad \text{blk4}\\ \{(0x14, \mathtt{MEMORY}\ \mathtt{mem}\ ((rf\ \ R2 \mapsto \mathtt{w2w}(rf\ \ R1))\ df) * s2 * \cdots)\}\\ s2 = \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ ((rf\ \ R1) = 0x0)\end{array} \tag{8}$$

$$\begin{array}{l}\{(0x20, \mathtt{MEMORY}\ \mathtt{mem}\ df * \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ z *\\ \qquad \langle \{rf\ \ R2\} \subseteq \mathtt{mem}\rangle * \langle rf\ \ R1 = 0x0\rangle * \cdots)\}\\ \quad \text{blk4}\\ \{(0x2C, \mathtt{MEMORY}\ \mathtt{mem}\ ((rf\ \ R2 \mapsto \mathtt{w2w}(rf\ \ R1))\ df) * s2 * \cdots)\}\end{array} \tag{9}$$

$$\begin{array}{l}\{(0x2C, \mathtt{REG}\ rf * \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle * \cdots)\}\\ \quad \text{blk5}\\ \{(rf\ \ R14, \mathtt{REG}\ rf * \cdots)\}\end{array} \tag{10}$$

There are safety assertions that are not discharged during composition: $\langle rf\ \ R14 \in \mathtt{succ}(0x1C)\rangle$ (Judgment 6), $\langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle$ (Judgment 10), and $\langle \{rf\ \ R2\} \subseteq \mathtt{mem}\rangle$ (Judgments 8 and 9).

Next, we examine the judgments of nodes in function `foo` (Judgments 6, 7, 8, 9, and 10) to see if these assertions can be discharged. $\langle \{rf\ \ R2\} \subseteq \mathtt{mem}\rangle$ can be discharged by the postcondition of Judgment 7, because it has the branch condition of $\langle (rf\ \ R2) = 0x40000000\rangle$, and $(\{0x40000000\} \subseteq \mathtt{mem}) = \mathtt{true}$. In order to get it formally, we frame the branch condition to the judgment itself. As a result, the assertion in the postcondition has $\langle (rf\ \ R2) = 0x40000000\rangle$, which implies the memory assertion of blk4. In our framework, this work is done by the abstract interpretation described in Section III-B.

The other two assertions cannot be discharged inside `foo` and are propagated to the judgment of blk3 by the analysis. After the analysis, we take the safety assertions propagated by a block node and frame them to the node judgment. We also merge the two judgments of the same block with the LPMerge rules. For example, we get the framed and merged judgment of blk3 in (11) (similar to blk4). The branch conditions form tautology after merging and are removed. The merged judgments have one entry in the precondition and two entries in the postcondition. It is easy to prove that they are well-formed Hoare judgments by definition (Section II-C).

$$\begin{array}{l}\{(0x14, \mathtt{REG}\ rf * \mathtt{S}\ \mathtt{sZ}\ z * \langle rf\ \ R14 \in \mathtt{succ}(0x1C)\rangle *\\ \qquad \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle * \cdots)\}\\ \quad \text{blk3}\\ \{(rf\ \ R14, \mathtt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * s1 * \cdots),\\ \ (0x20, \mathtt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * s1 *\\ \qquad \langle rf\ \ R2 = 0x40000000\rangle * \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle * \cdots)\}\end{array} \tag{11}$$

Next, we construct the terms needed for proving the function judgment of `foo`. Let P_i and Q_i be the precondition and postcondition of blk$_i$. After using the Base rule (Figure 4.b), we get three well-formed nodes, whose code is: `foo =` {`bbl 0x14, bbl 0x20, bbl 0x2C`}. We construct the two specifications of the function as: `foo_bspec` = {(`bbl 0x14`) → P_1, (`bbl 0x20`) → P_2, (`bbl 0x2C`) → P_3}, and `foo_kspec` = {(`bbl 0x14`) → Q_1, (`bbl 0x20`) → Q_2, (`bbl 0x2C`) → Q_3}. The exit specification is `foo_exits` = {(`bbl 0x14`, Q_1), (`bbl 0x2C`, Q_3) }. The initial condition has the two assertions that are not discharged by foo: `foo_init` = $\langle rf\ \ R14 \in \mathtt{succ}(0x1C)\rangle * \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle * \cdots$. With these terms, we are able to prove the judgment of function `foo`:

```
FUN_SPEC WF_NODE SAFE_INS foo 0x14 foo_init
  foo_exits foo_predecessor foo_bspec foo_kspec
```

where `foo_predecessor` is the predecessor relation of nodes: {`bbl 0x14` → {`bbl 0x20`}, `bbl 0x20` → {`bbl 0x14`}, `bbl 0x2C` → {`bbl 0x20`}}.

By applying the Induction rule (Figure 4.b), we get the well-formed node of function `foo`, whose three label predicate entries in the postcondition come from the postconditions of blk5 and blk3 (we only write out state predicates for clarity):

$$\begin{array}{l}\mathtt{WF\_NODE\ SAFE\_INS}\\ \{(0x14, \langle rf\ \ R14 \in \mathtt{succ}(0x1C)\rangle * \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle)\}\\ \ \mathtt{fun\ 0x14}\ \text{// foo}\\ \{(rf\ \ R14, \cdots),\ (rf\ \ R14, \cdots),\\ \ (0x20, \langle rf\ \ R2 = 0x40000000\rangle * \langle rf\ \ R14 \in \mathtt{succ}(0x2C)\rangle * \cdots)\}\end{array} \tag{12}$$

With this Hoare judgment of `foo`, we repeat the above reasoning process for function `entryFun`. Inside this function, the two assertions of `foo` are discharged by the postcondition of blk1, where $R14$ is $0x10$. As a result, we are able to prove the `PROG_SPEC` judgment given in Section I-A, where `entryFun` is the set of nodes of entryFun, `pred` is the predecessor relation of these nodes (Figure 1.c gives the pictorial representation of these two terms), and `bspec` is the mapping from the nodes to their preconditions.

V. RELATED WORK

Boyer and Yu made the first attempt to verify small real-world executables with symbolic execution, but their specifications and proofs were done manually [1]. Myreen et al. developed a traditional Hoare logic for machine code programs [14] and a decompiler to reuse proofs for multiple architectures [15]. Both the logic and the decompilation require structured code in order to compose a judgment or to develop a function. 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]. Their logic requires a complicated semantics and soundness proof.

Proof-carrying code uses a VCG-based approach to verify programs without formalizing the method itself [16].

Shao's group developed certified assembly programming to verify low-level code [17], [24]. It requires manually provided specifications of code, and the verification process is interactive. This is similar to the last step in our framework, in which the specifications are instantiated and verified.

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

VI. IMPLEMENTATION AND CONCLUSION

We implemented $\mathcal{L}_f$ and the framework for verifying safety properties in the HOL theorem prover and applied it to automatically prove the memory safety and the control flow integrity of rewritten ARM executables. The definition of our logic is about 60 lines in HOL, proof scripts of useful theorems are about 600 lines, and automating libraries are about 8000 lines including the interprocedural interpreter.

The ARM executables we proved include our test programs and MiBench programs [9]. The proven MiBench programs have text sections over hundreds of machine instructions, e.g. StringSearch has 1104 machine words. These programs can run on a development board based on the NXP LPC2129 chip, which contains an ARM7TDMI core and targets industrial automation.

REFERENCES

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

[2] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI)*, pages 401–414, June 2006.

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

[4] 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.

[5] 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.

[6] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[7] M. J. C. Gordon. *A Mechanized Hoare Logic of State Transitions*, pages 143–159. Prentice Hall International (UK) Ltd., 1994.

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

[9] 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.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[11] I. Jager, T. Avgerinos, E. Schwartz, and D. Brumley. Bap: Binary analysis platform. In *CAV*, 2011.

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

[13] 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.

[14] 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.

[15] 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 Formal Methods in Computer-Aided Design*, 2008.

[16] 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.

[17] 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.

[18] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 391–407, 2009.

[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] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, Mar. 2007.

[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] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.