AUSPICE: Automatic Safety Property Verification for Unmodified Executables

Jiaqi Tan, Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan

Department of Electrical & Computer Engineering, Carnegie Mellon University
tanjiaqi@cmu.edu,htay@andrew.cmu.edu,rgandhi@ece.cmu.edu,priya@cs.cmu.edu

Abstract. Verification of machine-code programs using program logic has focused on functional correctness, and proofs have required manually-provided program specifications. Fortunately, the verification of shallow safety properties such as memory and control-flow safety can be easier to automate, but past techniques for automatically verifying machine-code safety have required post-compilation transformations, which can change program behavior. In this work, we automatically verify safety properties for unmodified machine-code programs without requiring user-supplied specifications. We present our novel logic framework, AUSPICE, for automatic safety property verification for unmodified executables, which extends an existing trustworthy Hoare logic for local reasoning, and provides a novel proof tactic for selective composition. We demonstrate our fully automated proof technique on synthetic and realistic programs, and our verification completes in 6 hours for a realistic 533-instruction string search algorithm, demonstrating the feasibility of our approach.

1 Introduction

Interactive theorem proving using program logics is a promising technique for reasoning about executable (i.e. machine-code) programs, as it provides a succinct specification of the program. However, formally reasoning about machine-code is challenging as accounting for low-level details and writing proofs interactively can be tedious. Program logics have been developed to formally reason about the low-level state (e.g. registers, main memory) in machine-code programs: Myreen et al. developed a Hoare logic for realistically modeled machine-code [12]. These logics are designed to verify the correctness of programs, and hence must capture the complete execution state of the program, which requires manually supplied specifications e.g. loop invariants, function pre- and post-conditions. Hence, techniques for reasoning about program correctness ease the job of the proof author [13], but do not fully automate proof generation. Fortunately, verifying shallow safety properties can be easier, as we are only concerned with the parts of program state which affect our desired safety properties. Thus, there are more opportunities for proof automation. Zhao et al. [22] proposed a program logic for automatically verifying safety properties in executables, but programs must be compiled with a modified compiler and safety checks must

be added post-compilation [23], thus developers cannot observe how the safety checks added to their programs may change them.

In this paper, we present a novel logic framework, AUSPICE, for automatically verifying safety properties for **unmodified** machine-code programs: programs generated by an unmodified compiler, without any post-compilation transformations (e.g. binary rewriting) applied to them. Thus, any safety checks must be added as source-code statements. This enables developers to gain assurance of their program's behavior and safety from observing the added safety checks. Our contributions are: (i) a novel logic framework, AUSPICE, for automatically verifying safety properties in *unmodified* machine-code programs, (ii) a program logic, $\mathcal{L}_{\mathcal{LR}}$, which enables *local reasoning* to ensure that safety properties are asserted and checked for every instruction in a machine-code program, (iii) a proof tactic for *selective composition* which enables the automatic verification of safety properties without manual inputs, and (iv) an empirical evaluation of AUSPICE on verifying real-world machine-code. *To the best of our knowledge, AUSPICE is the first logic framework which enables the fully automated proving of safety properties for unmodified ARM machine-code programs,* avoiding the post-compilation transformations required by [22, 23]. We verify safety properties for ARM machine-code programs, although our technique can be applied to other architectures. This paper discusses details of AUSPICE as used in [3] to automatically verify safety properties formally for a novel software fault isolation [19] technique, while [3] focuses on the technique for software fault isolation.

Intuition. Our safety property verification uses Hoare logic to reason about machine-code. Hoare logic was designed to reason about program correctness, hence, typical Hoare logic proofs must reason about the "global" effects of programs, i.e. capture all possible values of program state. Our first intuition is that our safety properties at each instruction are affected only by the program state immediately before the instruction runs. This enables us to consider only a subset of program state and perform "local" reasoning (§3), and avoid requiring manually supplied specifications. Second, previous efforts to automate safety property verification [23] relied on binary rewriting to insert safety checks. In unmodified machine-code, safety checks must be implemented entirely in source-code. Our second intuition is that, when verifying safety properties for unmodified machine-code, safety checks inserted in program source can span a larger part of a program than our "local" scope of reasoning described above. Hence, we develop a novel proof tactic, *selective composition* (§4), which uses the Hoare logic Compose rule (§2.4) to help us reason about safety properties using additional contextual information not available in purely local reasoning.

1.1 Problem Statement

Goals. The main objective of our logic framework is to automatically prove safety properties for machine-code programs which have been compiled using an unmodified compiler, with no post-compilation modifications (e.g. binary rewriting). The goals of our logic framework are: (i) to use an independently developed, trustworthy logic so that our approach is trustworthy; (ii) to fully

automate our proof by not requiring manual inputs from the user, and (iii) to formalize the notion of safety for the execution of a machine-code program.

Non-Goals. We do not intend to prove the correctness of machine-code programs. We are not concerned with the security and privacy of applications implemented by the machine-code. In this paper, we present our logic framework for automatic safety property verification for executables; we address how these safety properties are achieved in unmodified executables in [3].

Scope. We choose to verify safety properties for the machine-code of programs rather than their source-code so that (i) we do not need to trust the compiler used, thus minimizing our Trusted Computing Base (TCB), and (ii) our verification does not need access to the source-code of the program. We require no modifications to the compiler used to generate the executables which we verify. Our logic framework currently targets ARM machine-code programs, although our techniques can be applied to machine-code for other architectures by (i) parameterizing the Hoare logic [12] with a different instruction semantics, and (ii) defining execution safety for the target architecture. We verify safety properties for user programs running on a commodity operating system (currently Linux).

Assumptions. Our logic framework uses the trustworthy formalization of the ARM Instruction Set Architecture (ISA) developed by Myreen et al. [11] at Cambridge University (the "Cambridge ARM model"). Thus, our verification inherits the assumptions and limitations of this model. We assume that the behavior of the program being verified is not affected by exceptions, interrupts, and page table operations, as these are not modeled in the model. We are also unable to verify safety properties in the presence of system calls, as the model does not capture the effects of specific system calls on user programs (we intend to explore verifying programs with system calls in future). We assume that the compiler and program obey the ARM-THUMB Procedure Call Standard (ATPCS) [1], which specifies the behavior for function calls/returns, and that the OS correctly isolates concurrently executing user programs. We also assume that the target program being verified was compiled with a well-known, unmodified compiler with well-known function prologues and epilogues, and that the machine-code contains function boundaries. We also require programs to be statically compiled so that all code to be executed is present, and that programs are not recursive.

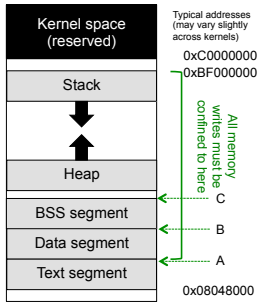
2 Background

2.1 ARM Architecture

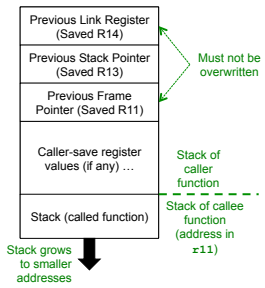
First, we review aspects of the ARM architecture pertinent to defining execution safety for ARM machine-code programs. ARM is an RISC, load/store architecture, and data instructions operate only on register contents but not memory [2]. There are 6 processor modes, and we focus on the user mode, which an operating system (OS) runs applications in. The remaining modes handle various types of exceptions, including system calls. Each ARM processor mode has a different set of visible registers, and we focus on only the registers visible in user mode:

registers `r0` through `r15`, and the status register (`CPSR`). We also consider the ATPCS [1], which specifies conventions for procedure calls and returns. By the convention in the ATPCS, registers `r13`, `r14`, `r15` store the stack pointer (`SP`), link register (`LR`) for return addresses and program counter (`PC`). We highlight these registers for their impact on control-flow safety. The state of an ARM processor comprises the registers `r0` to `r15`, the processor status register `CPSR`, and the processor's main memory (modeled as an array of 2^{32} byte-addressed bytes).

2.2 Safety properties for ARM machine-code programs



(a) Memory safety: Linux Process Memory Layout



(b) Control-flow safety: Function Activation Record

Fig. 1. Safety properties

Next, we discuss the memory and control-flow safety properties we wish to prove for ARM machine-code. We instantiate these safety properties in the context of user programs running on a specific OS (Linux in our current implementation), as our goal is to provide isolation for user programs running in an OS. The main goal of our safety properties is to ensure that (i) a running program cannot affect other running programs, and (ii) the running program does not exhibit any unexpected or emergent behaviors not already present in its machine-code.

Memory Safety. The goal of our memory safety policy is to prevent a program from modifying OS-addressable memory (thus preventing it from modifying OS data-structures in privilege escalation attacks), and from modifying its own instructions to prevent self-modification. In a multiprogramming OS such as Linux, each user program runs as a separate process, each with its own virtual memory with a common layout. In processors with 32-bits of addressable memory, each process has a 4 GB memory space, with the upper 1GB typically reserved for the OS. Figure 1(a) illustrates this layout. Our memory safety policy requires that all memory writes be restricted to the area between the start of the process's stack space (marked `0xBF000000`) and the start of the text segment of the code.

Control-flow Safety. The goal of our control-flow safety is to ensure that there are no unexpected control-flow transfers, that only instructions in the `text` section of the program are executed. We also require that there can be no control-flow hijacks with modified function return addresses. Our memory safety policy partially ensures control-flow safety by preventing the modification of the `text` section. Our control-flow safety is also enforced by protecting the return addresses for function calls saved on the program stack. First, we consider the ATPCS [1] convention. Registers `r11` and `r13` store the frame pointer (stack base address) and stack pointer (stack top address) respectively, while `r14` stores

the return address of the current function. When a function call is made, the caller function first saves its current values of `r11`, `r13`, and `r14` on the stack (whose base address is stored at `r11`), before loading the address of the next instruction (i.e. the return address) to (`r14`). Also, the ATPCS specifies that the stack grows downwards to lower addresses. Thus, to prevent control-flow hijacks, we must ensure that all memory writes are to addresses smaller than the current function's frame pointer (`r11`).

2.3 Hoare logic for ARM machine-code programs

We use the HOL4 theorem prover [16], and the Hoare logic [12] for ARM machine-code programs developed at Cambridge [11], to prove safety theorems. The Cambridge ARM model has been extensively tested and validated [6], providing us with a strong foundation for our logic. The Cambridge ARM model uses Hoare triple theorems and separation logic [15] to describe the behavior of each instruction, and the model captures realistic details of ARM instructions, which we illustrate briefly. The model decompiles each ARM instruction to a Hoare triple theorem of the form `(p) c (q)`, with `p` and `q` representing the state of the processor before (pre-state) and after (post-state) executing code `c` respectively[1]. Then, the theorem `(p) c (q)` informally means that for a processor in state `p` before running `c`, after running `c`, the processor will have state `q`. The pre- and post-states `p` and `q` contain assertions about the values of machine resources e.g. registers, status flags and the program counter. They can also contain pure boolean assertions which describe relationships among the values of machine resources, which encode relationships which hold true before or after the execution of the instruction. The theorem for the ARM instruction `0xE5832000`, with the mnemonic "`str r2 [r3]`", is shown below:

```
|- SPEC ARM_MODEL (aR 3w r3 * aR 2w r2 * aPC p * aMEMORY df f)
   {(p,0xE5832000w)} (aR 3w r3 * aR 2w r2 * aPC (p+4w) * aMEMORY df ((r3=+r2) f))
```

`SPEC` indicates that the theorem is a Hoare triple, while `ARM_MODEL` stores the ARM-specific instruction semantics [11]. The pre-state shows that the registers `r2` and `r3` contain the (symbolic) values *`r2`* and *`r3`* respectively, the main memory contains the map *`f`* with domain *`df`*, and the program counter has some address *`p`* before running the instruction. After running the instruction, the values of registers `r2`, `r3` remain unchanged, the program counter advances to *`p+4`*, while the memory has been updated to store the value that was in register `r2` at the address given by the value that was in register `r3`. The `*` operator is the separating conjunction [15] which asserts all other resources are unchanged.

2.4 Composition rule in Hoare logic

$$\frac{SPEC\ x\ p\ c_1\ q \quad SPEC\ x\ q\ c_2\ r}{SPEC\ x\ p\ (c_1 \cup c_2)\ r}\ \texttt{COMPOSE} \qquad \frac{SPEC\ x\ p\ c\ q}{SPEC\ x\ (p * r)\ c\ (q * r)}\ \texttt{FRAME}$$

[1] In Hoare logic, `p`, `q` are named pre-, post-condition, but we use the terms pre-, post-state as we call the boolean conditions imposed by a branch the pre-condition.

The Compose rule of Hoare logic [8] is shown above, which extends single instruction Hoare triple theorems to describe multiple instructions. One critical detail of this rule is that to apply the Compose rule to compose two Hoare triple theorems, the pre-state of the second theorem must be equal to the post-state of the first theorem. Conceptually, when instruction i_1 executes, followed by instruction i_2, as i_2 is executing immediately after i_1, the processor state observed just before i_2 executes is exactly the processor state after i_1 executes.

Pre-composition Tactic. A typical proof tactic for composing Hoare triple theorems for sequential instructions, i_1, i_2, with i_1 running immediately before i_2, into a single Hoare triple theorem, is given by the following steps: (i) Using the Frame rule (shown above), add machine state assertions in i_1, but not in i_2, to i_2's theorem; (ii) Using the Frame rule, add machine state assertions in i_2, but not in i_1, to i_1's theorem; (iii) Instantiate free variables in i_2 with the post-state machine resource values from i_1. We call these steps the pre-composition tactic. After carrying out the above theorem manipulation steps, the manipulated theorems i_1' and i_2' for both instructions will now have the post-state of i_1' matching the pre-state of i_2', allowing us to directly apply the Compose rule in Hoare logic.

For instance, consider the two instructions, i_1 ("`mov r3,r4`"), followed by i_2 ("`sub r2, r3, #16`"). We illustrate the use of the Compose rule to obtain a theorem describing the behavior of a program (or its fragment), $i_1 i_2$. The Hoare triple theorems for each of the two instructions are shown respectively:

```
|- SPEC ARM_MODEL
   (aR 3w r3 * aR 4w r4 * aPC p) {(p,0xE1A03004w)}
     (aR 3w r4 * aR 4w r4 * aPC (p + 4w))

|- SPEC ARM_MODEL
   (aR 2w r2 * aR 3w r3 * aPC p) {(p,0xE2432010w)}
     (aR 2w (r3 - 16w) * aR 3w r3 * aPC (p + 4w))
```

Thus, in composing the two theorems i_1, i_2 in our above example, our pre-composition tactic will carry out the following steps on the theorems i_1, i_2: (i) Use the Frame rule to add `aR 2w r2` to i_1 to get i_1'; (ii) Use the Frame rule to add `aR 4w r4` to i_2 to get i_2'; (iii) Instantiate the value of p to $p + 4w$, and $r3$ to $r4$ in i_2' to get i_2''; (iv) Apply Compose rule to theorems i_1', i_2'' to obtain:

```
|- SPEC ARM_MODEL (aR 3w r3 * aR 4w r4 * aPC p * aR 2w r2)
    {(p,0xE1A03004w); (p + 4w,0xE2432010w)}
    (aR 2w (r4 - 16w) * aR 3w r4 * aPC (p + 8w) * aR 4w r4)
```

The pre-composition tactic prepares two suitable Hoare triples for reasoning about the effects of code on the same pre-state (i.e. pre-state of the first Hoare triple) by placing them in the same context (i.e. describing the effects of the code in both triples in terms of the pre-state variables of the first Hoare triple).

3 Design: The $\mathcal{L}_{LR}$ Program Logic

Next, we describe the design of our logic framework for automatically verifying safety properties, and discuss the rationale behind our design decisions. Our

logic framework needs to fulfill three tasks. First, it needs to specify safety assertions for each instruction. A safety assertion of an instruction specifies the conditions which must be true before the instruction is executed for our memory and control-flow safety properties to hold. Second, it needs to ensure that the Hoare triple theorems for every instruction are encoded with their safety assertions. Third, it needs to define, formally, the requirements for a program to possess our desired safety properties.

$$\frac{\texttt{SPEC}\ x\ (\langle ms \wedge cfi_1 \wedge cfi_2 \rangle * p)\ \{(\texttt{offset}, \texttt{ins})\}\ q}{\texttt{MEMCFISAFE}\ x\ ((\texttt{MCSAt}\ offset\ ms\ cfi_1\ cfi_2)\ *\ p)\ \{(\texttt{offset}, \texttt{ins})\}\ q}\ \texttt{MEM\_CFI\_SAFE}$$

$$\frac{\texttt{MEMCFISAFE}\ x\ p\ c_1\ q \quad \texttt{MEMCFISAFE}\ x\ q\ c_2\ r}{\texttt{MEMCFISAFE}\ x\ p\ (c_1 \cup c_2)\ r}\ \texttt{MEM\_CFI\_SAFE\_COMPOSE}$$

$$\frac{\texttt{MEMCFISAFE}\ x\ p\ c\ q}{\texttt{MEMCFISAFE}\ x\ (p * r)\ c\ (q * r)}\ \texttt{MEMCFISAFE\_FRAME}$$

Fig. 2. Logic rules for $\mathcal{L}_{LR}$

3.1 Individual Instructions: Safety Assertion Specification

Figure 2 shows the `MEM_CFI_SAFE` rule for augmenting the Hoare triple theorem of a single instruction with its safety assertion. This rule overcomes the challenge of reasoning about safety properties at every instruction using Hoare logic. We add our safety assertions as a pure boolean condition to the pre-state of an instruction's Hoare triple. Then, when the Compose rule (§2.4) is applied to compose theorems of multiple instructions, the pre-states of successor instructions (q in the Compose rule) will be hidden, thus hiding our augmented safety assertions. Also, safety assertions which hold can be simplified to true and eliminated from the Hoare triple. Thus, for a Hoare triple describing a sequence of instructions, we cannot tell if the theorem contains safety assertions for every instruction.

The `MEM_CFI_SAFE` rule overcomes this challenge by ensuring that the Hoare triple for every instruction has been augmented with its safety assertions. This rule has two features. First, `MEM_CFI_SAFE` can be instantiated only from single instruction Hoare `SPEC` theorems, because code c in the `SPEC` theorem in the rule antecedent admits only a single instruction with the machine word `ins` located at address `offset`. The second rule which generates the safe `MEMCFISAFE` theorem, `MEM_CFI_SAFE_COMPOSE`, does not admit Hoare triple `SPEC` theorems, and only allows the composition of `MEMCFISAFE` theorems. Second, the `MEM_CFI_SAFE` rule can be instantiated only when the pre-state is augmented with our safety assertion, the pure boolean conjunction, $ms \wedge cfi_1 \wedge cfi_2$, in its pre-state. Thus, the `MEMCFISAFE` relation indicates the resulting Hoare triple has been augmented with our safety assertion in its instruction pre-state. `MCSAt` is a syntactic relation which associates our safety assertion, $ms \wedge cfi_1 \wedge cfi_2$, with the address $offset$ which the assertion applies to. We also add the safety assertions ms, cfi_1, cfi_2 to the hypotheses of the theorem, to indicate that they are undischarged.

Safe instruction semantics are sound. Our safe instruction semantics, in the form of `MEMCFISAFE` theorems, are a special form of Hoare triple theorems.

$$
\begin{aligned}
&\vdash \texttt{FUN\_SAFE}(addr, NODES, FUNCS, CFG_{pred}, CFG_{succ}, ICFG_{callpred}, ICFG_{callsucc},\\
&\quad ICFG_{retpred}, ICFG_{retsucc}, assns_{entry}, postcond_{exit}, prestate, poststate)\\
&\Leftrightarrow ((\forall node \cdot node \in NODES \Rightarrow (\texttt{min}(node, addr) = addr)) \wedge\\
&\quad (\forall node, pred \cdot node \in NODES \Rightarrow pred \in CFG_{pred}(node) \Rightarrow\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, pred, node, x, c_1, p, q) \wedge\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, node', x, c_1, q, r) \wedge (pd_1 \Rightarrow assn_2)) \wedge\\
&\quad (\forall node, succ \cdot node \in ICFG_{callsucc}(succ) \Rightarrow succ \in ICFG_{callpred}(node) \Rightarrow\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, node, succ, x, p, q) \wedge\\
&\qquad \texttt{FUN\_SAFE}(succ, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_2, pd_2, q, r) \wedge (pd_1 \Rightarrow assn_2)) \wedge\\
&\quad (\forall node, pred \cdot node \in ICFG_{retsucc}(pred) \Rightarrow pred \in ICFG_{retpred}(node) \Rightarrow\\
&\qquad \texttt{FUN\_SAFE}(pred, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_1, pd_1, p, q) \wedge\\
&\qquad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, node', x, q, r) \wedge (pd_1 \Rightarrow assn_2))
\end{aligned}
$$

Fig. 3. `FSI` rule: Judgment for Interprocedural Function Safety

They are augmented to ensure that every instruction described in an `MEMCFISAFE` theorem has an associated safety assertion, added to it as a pure boolean condition in the pre-state of the instruction's theorem. We proved the following theorem: $\vdash \forall x\ p\ c\ q\ \cdot\ \texttt{MEMCFISAFE}\ x\ p\ c\ q\ \Rightarrow \texttt{SPEC}\ x\ p\ c\ q$. Informally, our safety-augmented Hoare triple theorems retain a direct correspondence to the Hoare triple theorems proven by the Cambridge ARM model. Hence, our safe instruction semantics inherits the soundness of the Cambridge ARM model.

3.2 Sequential Code Blocks

Next, we describe how we obtain safety-augmented Hoare triple theorems for basic blocks of sequential code (safe basic block theorems). A basic block is a sequence of instructions which execute sequentially, with a single entry and single exit instruction. The two rules (Fig. 2) we need for building safe basic block theorems are `MEM_CFI_SAFE_COMPOSE`, and `MEMCFISAFE_FRAME` (proved using the Frame rule in separation logic). These two rules allow us to inductively build up a safe basic block theorem from safety theorems for individual instructions. The process of building up a safety theorem for a basic block of sequential code is the same as that of composing Hoare triple theorems (§2.4), except that only safety-augmented Hoare triple theorems can be composed. This process is repeated recursively for every instruction in a basic block to obtain a single safe theorem for the basic block. Our safe basic block theorems have the same semantics as Cambridge ARM Hoare triples, as proved in §3.1.

3.3 Function Judgment for Local Reasoning

Global vs. Local Reasoning. In a typical correctness proof for a program using Hoare logic, we would repeatedly apply the Compose rule to the Hoare triple for every instruction in the program to obtain a single Hoare triple describing the entire program. This is a "global reasoning" process which identifies

the final values of all registers, main memory, etc. at the end of the program's execution. In the presence of loops and function calls, loop invariants and pre- and post-conditions for functions will need to be manually provided.

For safety assertions to hold in a program, we only need to ensure that the safety assertions for each instruction hold locally at that instruction. For the safety assertions at a given instruction i_2 to hold, we consider every instruction i_1 that can execute before i_2. Then, the post-state of each instruction i_1 must have machine resources values that result in the safety assertions at i_2 being true. The reasoning behind this process is analogous to the mechanics of the pre-composition process before applying the Compose rule in Hoare logic (§2.4). As long as the machine resource values from the post-states of predecessor instructions i_1 enable the safety assertion at i_2 to be true, the safety assertion holds. In addition, any pure boolean condition from the post-state of predecessor instructions i_1 will also apply to the pre-state of instruction i_2. Hence, safety properties hold on a per-instruction basis. To check if a safety assertion holds for an instruction, we only need to perform "local reasoning" by considering the post-state values and boolean conditions of all predecessor instructions.

Safe Function Judgment. We define the `FUN_SAFE` rule (Fig. 3), which defines what it means for a function to be safe. This rule encodes our "local reasoning" process for verifying that safety assertions hold. First, we rearrange `MEMCFISAFE` theorems to form `HOARE_WITH_ASSERT` theorems, which make explicit the hypotheses (which are the undischarged safety assertions) of the theorems, and rearrange machine resource expressions into a tuple for pattern-matching later.

$$\vdash \texttt{HOARE\_WITH\_ASSERT}(pd, assn, pc_{pre}, pc_{post}, x, c, p, q) \Leftrightarrow$$
$$assn \Rightarrow (\texttt{MEMCFISAFE}\ x\ (\texttt{aPC}\ pc_{pre}\ *\ p\ *\ \texttt{precond}\ pd)\ c(\texttt{aPC}\ pc_{post}\ *\ q)))$$

A function is comprised of basic blocks of instructions in the function. In a function's intra-procedural control-flow graph (CFG), the nodes are basic blocks of the function's instructions, and the edges are control transfers within the function. In a function's inter-procedural CFG, the nodes are (i) basic blocks which call other functions, (ii) basic blocks which are return-sites from callee functions, and (iii) callee functions, while edges are function calls or returns.

Arguments to the `FUN_SAFE` relation. To formally specify the requirements for a function to be safe, we consider the safety assertions which must be discharged at each edge in both the intra- and inter-procedural CFGs. First, the `FUN_SAFE` relation is parameterized by the function address $addr$, a set of addresses of basic blocks in the function $NODES$, a set of addresses of callee functions $FUNCS$, and 6 maps CFG and $ICFG$ specifying the predecessors and successors of edges in the function's intra- and inter-procedural CFGs. `FUN_SAFE` also records the safety assertions of the function, $assns_{entry}$, the conditions which the function guarantees hold at its exit $postcond_{exit}$, as well as the machine resource pre-state $prestate$ and post-state $poststate$. The first clause specifies that the address of the function is the lowest basic block address for the function.

Intra-procedural safety requirements. The second clause specifies that for each intra-procedural CFG edge, the safety assertions of the instruction at the destination of each edge must be discharged by the post-condition of the instruc-

tion at the source of the edge, i.e. ($pd_1 \Rightarrow assn_2$). Also, in the spirit of the Hoare Compose rule, we require that the post-state of the predecessor instruction q, is equal to the pre-state of the successor instruction.

Inter-procedural safety requirements. The third and fourth clauses specify the requirements for inter-procedural CFG edges. The third clause specifies that for call edges, the safety assertions of the called function must be discharged by the post-condition of the calling basic block, i.e. $pd_1 \Rightarrow assn_2$. The fourth clause specifies that for return edges, the safety assertions of the basic block which is the return site for the function must be discharged by the post-condition of the returning function, i.e. $pd_1 \Rightarrow assn_2$. In both clauses, we require that the post-state of the predecessor node must equal the pre-state of the successor node.

Compositional reasoning for functions. Although the `FSI` rule appears to be recursively defined without a base case, this rule actually collapses to include only the first and second clauses for functions which do not call any other functions. This implies that our safety property proving requires the CFG of the program to have no cycles, i.e. we are unable to analyze recursive programs.

4 Implementation: Proofs using $\mathcal{L}_{LR}$

We describe the implementation of our automatic safety property verification. Our framework consists of 128 lines of HOL4 definitions and 11.8 KLOC of proof scripts in ML. We illustrate how safety properties are automatically specified for each instruction, and describe our *selective composition* proof tactic which prepares our safe basic block theorems for automated proving, and our abstract interpretation framework which automatically discharges proof obligations.

4.1 Automatic Safety Property Specification

To illustrate the safety assertions we augment instructions with, consider the instruction word `0xE5832000` (`str r2 [r3]`) located at address `0x81E0`. We first obtain the following Hoare logic theorem from the decompiler:

```
|- SPEC ARM_MODEL (aR 3w r3 * aR 2w r2 * aPC 0x81E0 * aMEMORY df f)
  {(0x81E0,0xE5832000w)} (aR 3w r3 * aR 2w r2 * aPC 0x81E4 * aMEMORY df ((r3 =+ r2) f))
```

Suppose the `text` section of this program lies in the range $[0x80B4, 0x85F4]$. This instruction writes to the byte locations $r3, r3+1, r3+2, r3+3$. Thus, we set the first conjunct in the safety assertion ms to $\{r3+3; r3+2; r3+1; r3\}$ $\subseteq$ $\{addr \mid \texttt{0x85F8} \leq addr \wedge addr \leq \texttt{0xBF000000}\}$ which asserts that the memory locations written to are in our allowed safe region. Then, the first control-flow safety conjunct, cfi_1 is set to $\exists pc.pc = \texttt{0x81E4} \wedge pc \in \{addr \mid \texttt{0x80B4} \leq addr \wedge addr \leq \texttt{0x85F4}\}$, which asserts that the address of the next instruction to be executed lies in the `text` section of the binary. Next, the second control-flow safety conjunct, cfi_2 is set to $\{r3+3; r3+2; r3+1; r3\}$ $\subseteq$ $\{addr \mid addr < \texttt{r11}\}$, which asserts that the memory locations written to cannot overwrite the saved link register (`lr`, stored in register `r11`) value on the stack.

4.2 Selective Composition Proof Tactic

Next, we discuss the steps for automatically proving that safety properties hold using $\mathcal{L}_{LR}$. After augmenting single instruction theorems with safety assertions (§3.1) and obtaining safe basic block theorems (§3.2), we need to prove that the antecedents in the `FSI` rule (Fig. 3) hold. Each of the top-level conjuncts of `FSI` requires either an `HOARE_WITH_ASSERT` theorem for safe basic blocks or an `FUN_SAFE` theorem for safe functions. We also need to prove that the pre-condition pd_1 of each predecessor basic block or function discharges the safety assertion $assn_2$ in the successor, based on the program's CFG.

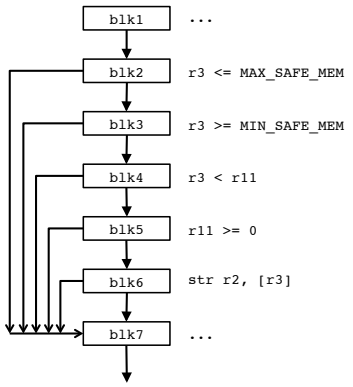


Fig. 4. Possible structure for program with safe `str r2 [r3]`.

From §4.1, we can see that the safety assertion at each instruction contains three conjuncts: one for memory-safety and two for control-flow safety. In a safe program, for the theorem of a given instruction i_2, its predecessor (safe basic block or function) theorem i_1 should have a pre-condition which implies the safety assertion of i_2. Observe that the safety assertion for each instruction has three conjuncts, and each of the range conjuncts (ms and cfi_1 in §4.1) is specified by two conjuncts: one each for the lower and upper bounds of the valid memory locations written to. Thus, the safety assertion at each instruction comprises multiple conjuncts. However, in a machine-code program, each basic block can only carry out one of the "elementary" arithmetic comparison operations (one of $<$, $>$, $\leq$, $\geq$, etc.), because each $cmp*$ instruction is a branch and will mark the end of the basic block it belongs to. Hence, information from multiple predecessor basic blocks are required to discharge the safety assertion at each instruction.

Forward propagation of branch conditions. In §3.3, we noted that we must use a *local reasoning* process to ensure our proof process is automatic, because global reasoning would require manually specified information. However, our safety assertions contain multiple conjuncts, whereas each basic block in machine-code can provide only one conjunct in its pre-condition. To enable our proof process to use pre-conditions from predecessors which are more than one edge away from a given basic block in the program CFG, we selectively "propagate" the pre-conditions of basic blocks forward. We call this process "selective composition", where we apply the pre-composition tactic (§2.4) forward to successor theorems under certain conditions.

To illustrate the process of selective composition, consider the following example. Consider the store instruction from above, `str r2 [r3]`. Figure 4 shows the CFG of the possible structure of the basic blocks in a program with safety checks to ensure that the store instruction is safe. Then, we need the pre-conditions from basic blocks $blk_2, blk_3, blk_4, blk_5$ to be available at blk_5 to dis-

charge the safety assertion at blk_6. At each of the nodes $blk_2, blk_3, blk_4, blk_5$, there are two Hoare triple theorems: one where each blk_i executes blk_{i+1} next (for $i \in \{2,3,4,5\}$), and one where the safety check fails, and each blk_i goes on to execute blk_7. However, we do not compose $blk_2, blk_3, blk_4, blk_5$ to form a single Hoare triple theorem, because the resulting block of code will have multiple exits, which is not captured by our safe basic block theorem (the `MEM_CFI_SAFE_COMPOSE` rule), which only admits single exit blocks. Instead, we iteratively carry out the steps in the pre-composition tactic (§2.4) for basic blocks $blk_2, blk_3, blk_4, blk_5$. This enables us to select to place the analysis of the machine-code in blocks $blk_2, blk_3, blk_4, blk_5, blk_6$ in the context of the pre-state values of machine resources in blk_2. This then allows us to discharge the safety assertion at blk_6 with the combined pre-conditions of $blk_2, blk_3, blk_4, blk_5$ at blk_5. We call this process "selective composition" because we are only carrying out the pre-composition process without applying the composition rule. Note that this selective composition process succeeds only when the target basic block which the pre-conditions are being propagated forward to have only one predecessor basic block. Only then is the pre-condition from the predecessor block blk_i the only pre-condition that will apply at the successor block blk_{i+1}.

Local use of global information. Next, we describe the second instance of *selective composition*. Recall that for control-flow safety, we require that the address of each instruction executed must be within the `text` section of the program. The address of the next instruction to be executed can be statically determined at every point of the program except where a function returns to its caller. Consider a typical machine-code instruction for returning from a function call `pop {pc}`. Control is being returned from the function by restoring the saved link register value from the stack to the program counter. The instruction will be specified by the Hoare triple theorem:

```
|- SPEC ARM_MODEL (aPC p * aR 13w r13 * aMEMORY df f)
   {(p,0xE8BD8000w)} (aPC (f r13) * aR 13w (r13 + 4w) * aMEMORY df f)
```

Here, `aMEMORY df f` is an assertion that the main memory is the map `f` which when applied to an address *addr*, returns the word stored at *addr*, and *df* is a set specifying the address domain of f. Thus, in the post-state of this instruction, we can see that the next instruction to be executed is at address `f r13`. However, the memory map f does not contain any information that enables us to determine the value of `f r13`. The return address for a (non-leaf) function is saved to the stack in the function prologue before any instructions in the function. An example of such an instruction is `push {lr}`, with the following Hoare triple:

```
|- SPEC ARM_MODEL (aR 14w r14 * aR 13w r13 * aPC p * aMEMORY df f)
   {(p,0xE92D4000w)} (aR 14w r14 * aR 13w (r13 - 4w) * aPC (p + 4w)
   * aMEMORY df ((r13 - 4w =+ r14) f)
```

The memory in the post-state of the function is `((r13 - 4w =+ r14) f)`, which contains the value of the link register, `r14`, at the top of the stack, at the address `r13 - 4`. Hence, the information we need to discharge the control-flow safety assertion at the function exit is the memory expression at the post-state of the function prologue, and the new value of register `r13`. After substituting the post-state memory and register `r13` values of the function prologue into the return

instruction, the program counter in the return instruction post-state will contain `((r13 - 4w =+ r14) f) (r13 - 4w)` which simplifies to `r14`, and the safety assertion simplifies to $\texttt{r14} \in \{\texttt{addr} \mid \texttt{0x85F8} \leq \texttt{addr} \wedge \texttt{addr} \leq \texttt{0x85F4}\}$, which can be discharged by any caller of the function, which supplies a concrete value of `r14`. Again, we can use the pre-composition tactic to substitute the value of the memory (and registers) at the post-state of the function prologue into every subsequent basic block in the function. As long as the prologue precedes every instruction in the function, and the function does not alter the callee-saved registers until the its epilogue, our proof steps will not affect the memory expression.

4.3 Automatic Discharge of Proof Obligations

There are two ways to discharge the safety assertions of a theorem. First, for a given safety theorem, the pure boolean conditions of the pre-state of the theorem preceding it may imply the safety assertion holds for the current theorem. Second, if the former does not hold, then the safety assertion is added to the hypotheses of the preceding instruction, and the Frame rule is used to add the undischarged assertion to the theorems of the preceding instructions. We use abstract interpretation [4] to identify safety assertions which cannot be discharged. At each instruction, our analysis records the safety assertions which need to be framed to the safe instruction theorem for that instruction.

We use a flow-sensitive backwards fixed-point analysis. Our analysis proceeds across all nodes in reverse topological order in each iteration. At each node, the analysis checks that for each predecessor node, the instruction theorem for that node has pure boolean conditions which can discharge the safety assertions at the current node. For safety assertions the predecessor cannot discharge, our analysis adds the assertion to the predecessor node, propagating the assertion backwards up the CFG. Our analysis is also inter-procedural, but its analysis is context-insensitive, so that the analysis information for each function is the union of the information resulting from all its callers. This does not affect our analysis, as it generates a conservative safety theorem for a function so that it can satisfy the safety requirements of all its call-sites and return-sites.

In the general case, this analysis may not terminate. If there are safety assertions being propagated which have values that change with a loop, the analysis will not terminate. This is because the free variable instantiation at loop boundaries will generate new safety assertions to be framed whenever the assertion is propagated across the loop boundary. We prevent the assertion analysis from running forever by (i) recording the propagation path of safety assertions, and (ii) aborting the analysis if a cycle is detected in this path. Then, we inform the user that we are unable to prove the safety properties for the program.

4.4 Discussion: Soundness

The semantics of our single instruction (§3.1) and basic block (§3.2) safety theorems are sound, as we proved they have the same semantics as the Cambridge

ARM model [12] (§3.1), which is sound. At the function level, the construction of our `FSI` rule ensures that we have captured all the possible relationships between basic blocks and functions being called and returned from. The `FUN_SAFE` theorem requires that its CFG predecessor and successor maps are correct for the safety verification to be sound. Standard algorithms exist for constructing control-flow graphs for programs, so we do not envision the requirement for accurate CFGs to be a problem. Our selective composition proof tactic does not need to be correct, as the soundness of our verification depends only on the soundness of our logic. Any bugs in our proof tactic will only result in a failure by our system to prove the safety of an safe executable.

5 Evaluation

We aim to show that we can verify real-world programs, and we pick programs with constructs that are challenging to verify. We also measure the runtime of our verification to show the feasibility of our verification. Our test programs were compiled using an unmodified version of the `gcc` toolchain for the ARMv7 architecture with `-O0` optimization. Figure 5 summarizes our test programs. `sort` implements the Bubble Sort algorithm which has a doubly-nested loop, which can be challenging to verify. `sort` also contains 2 other functions to test our ability to verify safety in programs with multiple function calls/returns. `memcpy` is an implementation of the the C library function which we developed, and shows we can verify a real-world function. `stringsearch` is an application in the MiBench commercially-representative embedded benchmark suite [7], and it implements the Boyer-Moore string search algorithm, and demonstrates our verification on real-world programs.

Test Program	CFG edges	Instructions	Functions	Description
`memcpy`	27	116	2	Real-world `memcpy`
`sort`	112	337	5	Nested loops, function calls/returns
`stringsearch`	153	530	5	Boyer-Moore string search (MiBench [7])

Fig. 5. Test programs, their sizes, and the purpose of each test.

	Cambridge ARM Decompiler	Safe Basic Blocks	Abstract Interpretation	Safe Function	Total Proof Time
memcpy	1.3 mins	2.7 mins	5.7 mins	6.7 mins	16.4 mins
sort	2.5 mins	11.2 mins	36 mins	73 mins	122.7 mins
stringsearch	2.8 mins	15.3 mins	327.6 mins	17.8 mins	363.5 mins

Fig. 6. Verification runtime.

Figure 6 shows the time taken to verify the safety of each of our test programs. We carried out the verification on an 2.6 GHz Core i7 system. The verification of our simple `memcpy` test-case took 16 minutes, while the `sort` test-case was more demanding and took about 2 hours to verify as it had a doubly-nested loop with multiple function calls. `stringsearch`, took about 6 hours to verify,

as its inter-procedural CFG was complex, with function call-chains that were 3 calls deep. However, this was significantly faster than ARMor [23], which took 8 hours to verify the same `stringsearch` test program on a computer with similar specifications. We believe these are reasonable times for verifying safety properties, as programs only need to be verified once when they are first installed.

6 Related Work

Many techniques have been developed for verifying machine-code programs using logic. Certified assembly programming uses a Hoare logic with separation logic to build certified libraries [21, 14], but they require specifications to be manually annotated in programs, and their verification is interactive. Tan and Appel [17] developed a program logic to reason about multiple-entry, multiple-exit machine-code fragments for reasoning about unstructured control-flows in executables for Foundational Proof Carrying Code (FPCC). They require programmers to use a special compiler to generate machine-code programs annotated with types [10], while we verify unmodified executables compiled using an off-the-shelf compiler. Executables have also been verified without using a program logic, although concise theorems cannot be proven. Xu et al. [20] verify safety properties for machine-code using static-analysis. Thakur et al. [18] perform model-checking on machine-code without requiring a precomputed, fixed, inter-procedural CFG.

XFI [5] and ARMor [23], are software fault isolation [19] implementations which ensure that (x86 and ARM, respectively) executables possess memory and control-flow safety properties, and they verify that the machine-code programs they process possess memory and control-flow safety properties. XFI requires modules being verified to be annotated with hints. PittSFIeld [9] also provides software fault isolation for x86 executables, but it verifies that its safety rewriting is correct, as opposed to verifying that the executables it processes are safe. ARMor [23] is closest to our work. They require machine-code to be compiled with a modified compiler, after which the program must undergo binary rewriting to insert safety checks. In contrast, we can prove safety properties automatically for unmodified executables by using our novel logic framework with our *selective composition* proof tactic.

7 Conclusion and Future Work

We have presented a novel logic framework, AUSPICE, for automatically verifying safety properties in unmodified ARM machine-code programs. Our framework consists of a program logic, $\mathcal{L}_{LR}$, which uses a subset of a trustworthy Hoare logic for ARM executables [11, 12], and extends it for *local reasoning*, and the *selective composition* proof tactic, which fully automates the verification of safety properties. We demonstrated the feasibility of our fully automated safety property verification on one synthetic and two real-world (including a real-world benchmark [7]) examples. In future, we intend to validate our approach on more programs, and expand our verification to programs with system calls.

Acknowledgment.

The authors thank Lu Zhao for his assistance with ARMor [23, 22], and Magnus Myreen for his assistance with the Cambridge ARM model [11, 12].

References

1. The ARM-THUMB Procedure Call Standard (2000), http://infocenter.arm.com/help/topic/com.arm.doc.espc0002/ATPCS.pdf
2. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2014)
3. Anonymous: (closely related work). In: (Under review) (2015)
4. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL (1977)
5. Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., Necula, G.: XFI: Software Guards for System Address Spaces. In: OSDI (2006)
6. Fox, A.: Formal specification and verification of ARM6. In: TPHOLs (2003)
7. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: IEEE WWC Workshop (2001)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10) (Oct 1969)
9. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC Architecture. In: USENIX Security (2006)
10. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A Realistic Typed Assembly Language. In: Workshop on Compiler Support for System Software (WCSSS) (1999)
11. Myreen, M., Fox, A., Gordon, M.: Hoare Logic for ARM Machine Code. In: Fundamentals of Software Engineering (FSEN) (2007)
12. Myreen, M., Gordon, M.: Hoare Logic for Realistically Modeled Machine Code. In: TACAS (2007)
13. Myreen, M., Gordon, M., Slind, K.: Machine-code verification for multiple architectures: An application of decompilation into logic. In: FMCAD (2008)
14. Ni, Z., Shao, Z.: Certified Assembly Programming with Embedded Code Pointers. In: POPL (2006)
15. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE LICS (2002)
16. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: TPHOLs (2008)
17. Tan, G., Appel, A.: A Compositional Logic for Control Flow. In: VMCAI (2006)
18. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: CAV (2010)
19. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient Software-Based Fault Isolation. In: SOSP (1993)
20. Xu, Z., Miller, B., Reps, T.: Safety Checking of Machine Code. In: PLDI (2000)
21. Yu, D., Hamid, N., Shao, Z.: Building Certified Libraries for PCC: Dynamic Storage Allocation. In: ESOP (2003)
22. Zhao, L., Li, G., Regehr, J.: A Practical Logic Framework for Verifying Safety Properties of Executables. In: LOLA (2011)
23. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: ARMor: Fully Verified Software Fault Isolation. In: EMSOFT (2011)