AUSPICE: Automatic Safety Property Verification for Unmodified Executables

Jiaqi Tan(✉), Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan

Department of Electrical and Computer Engineering,
Carnegie Mellon University, Pittsburgh, USA
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 isolation 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. Our novel logic framework, AUSPICE, for automatic safety property verification for unmodified executables, extends an existing trustworthy Hoare logic for local reasoning, and provides a novel proof tactic for selective composition. We demonstrate our automated proof technique on synthetic and realistic programs. Our verification completes in 6 h for a realistic 533-instruction string search algorithm, demonstrating the feasibility of our approach.

1 Introduction

Interactive theorem proving using logic 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. 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 [16]. 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, and function pre-/post-conditions. Hence, techniques for reasoning about program correctness ease the job of the proof author [17], 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. [27] proposed a program logic for automatically verifying safety properties in executables, but programs must be compiled with a modified

A. Gurfinkel and S.A. Seshia (Eds.): VSTTE 2015, LNCS 9593, pp. 202–222, 2016.
DOI: 10.1007/978-3-319-29613-5_12

compiler and safety checks must be added post-compilation [27], 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). 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 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 ARMor [27]. We currently target ARM machine-code programs, although our technique can be applied to other architectures.

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 (Sect. 3), and avoid requiring manually supplied specifications. Second, previous efforts to automate safety property verification [27] 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 a program's source-code 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* (Sect. 4), which uses the Hoare logic Compose rule (Sect. 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, and we are not concerned with the security and privacy of applications implemented by the machine-code.

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 technique can be applied to machine-code for other architectures by (i) parameterizing the Hoare logic [16] 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. [15] 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 are also unable to verify programs with concurrent behavior, nor unstructured control-flow jumps (e.g., `longjmp` and switch statements). 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.

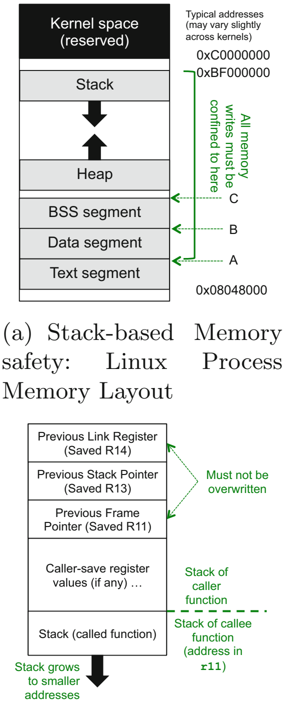
2 Background

2.1 ARM Architecture

First, we review aspects of the ARM architecture pertinent to defining execution safety for ARM machine-code programs. ARM is a 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

Next, we discuss the safety properties we wish to prove for ARM machine-code programs. At a high-level, we wish to prove that the execution of a machine-code program is isolated from any harmful effects of potentially malicious user input. Specifically, we wish to prove that the control-flow of a machine-code program cannot be hijacked and changed at runtime due to user input, and that new behaviors cannot be introduced through code injection or modification. We do this by proving that (i) machine-code loaded to memory cannot be overwritten; that (ii) function-return addresses saved to the program's stack cannot be changed; and that (iii) only machine-code initially loaded to memory is executed. Concretely, at the machine-code level, we prove two safety policies, *memory isolation*, and *control-flow isolation*, which together provide the machine-code safety properties sufficient to show that our desired high-level safety property holds. This is similar to Control-Flow Integrity [3], except that we disallow the use of arbitrary function pointers on a program's heap. We instantiate these safety properties in the context of user programs running in an OS (currently Linux), as our goal is to provide isolation for user programs running in an OS.


Kernel space (reserved)
Typical addresses (may vary slightly across kernels)
0xC0000000
0xBF000000
Stack
Heap
All memory writes must be confined to here
BSS segment
Data segment
Text segment
C
B
A
0x08048000
Previous Link Register (Saved R14)
Previous Stack Pointer (Saved R13)
Previous Frame Pointer (Saved R11)
Must not be overwritten
Caller-save register values (if any) ...
Stack of caller function
Stack (called function)
Stack of callee function (address in r11)
Stack grows to smaller addresses

(a) Stack-based Memory safety: Linux Process Memory Layout

(b) Stack-based Control-flow safety: Function Activation Record

Fig. 1. Safety properties

Memory Isolation. The goal of our memory safety policy, *memory isolation*, is to prevent a program from modifying its own instructions to prevent the introduction of new behaviors through self-modification. In a multiprogramming OS such as Linux, each user program runs as a separate process 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 1 GB 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 Isolation. The goal of our control-flow safety policy, *control-flow isolation*, 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 via modified function-return addresses. Our memory safety policy partially ensures control-flow isolation by preventing the modification of the `text` section. Our control-flow isolation 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, before loading 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`) (Fig. 1(b)).

2.3 Hoare Logic for ARM Machine-Code Programs

We use the HOL4 theorem prover [20], and the Hoare logic [16] for ARM machine-code programs [15] developed at Cambridge, to prove safety theorems. The Cambridge ARM model has been extensively tested and validated [9], providing us with a strong, trustworthy foundation for our logic. The Cambridge ARM model uses Hoare triple theorems and separation logic [19] 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)`, where `p` and `q` are predicates describing 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 a state satisfying `p` before running `c`, after running `c`, the processor will have state satisfying `q`. The predicates `p` and `q` are pre- and post-state 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 are true before or after an instruction executes. The theorem for the ARM instruction `0xE5832000` (mnemonic "`str r2 [r3]`") is:

$$\begin{aligned}\vdash\ & \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 3w}\ \mathit{r3} * \texttt{aR 2w}\ \mathit{r2} * \texttt{aPC}\ p * \texttt{aMEMORY}\ \mathit{df}\ f\\ & * \texttt{cond}((\mathit{r3}\ \&\&\ \mathit{3w} = 0w) \wedge (\mathit{r3} \in \mathit{df})))\ \{(p, \mathit{0xE5832000w})\}\\ & (\texttt{aR 3w}\ \mathit{r3} * \texttt{aR 2w}\ \mathit{r2} * \texttt{aPC}\ \mathit{(p{+}4w)} * \texttt{aMEMORY}\ \mathit{df}\ \ ((\mathit{r3} = +\mathit{r2})\ f))\end{aligned}$$

`SPEC` indicates that the theorem is a Hoare triple, while `ARM_MODEL` is the ARM-specific instruction semantics [15]. `aR 2w` and `aR 3w` are expressions which assert that a given register stores the specified value, where `2w` and `3w` indicate the register number whose value is being asserted, and the suffix `w` indicates the register number is a fixed-width word. Then, the pre-state shows that the registers `r2`

[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.

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, and the program counter advances to *p+4*. Also, `=+` is the map-update operator, hence `r3 =+ r2` indicates that 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 expression `cond`$((r3 \ \&\& \ 3w = 0w) \wedge (r3 \in df))$ is an assertion which specifies our memory alignment requirement for writes to the address *r3*, and that *r3* is in the domain of the memory map *f*. `*` is the separating conjunction [19] which asserts all other resources are unchanged.

2.4 Composition Rule in Hoare Logic

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

The Compose rule of Hoare logic [11] 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, so the processor state 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. This is similar to the "shift" operation described by Myreen et al. [15]. 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:

$$\vdash \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 3w}\ \mathit{r3} * \texttt{aR 4w}\ \mathit{r4} * \texttt{aPC}\ p)\ \{(p, \mathit{0xE1A03004w})\}$$
$$(\texttt{aR 3w}\ \mathit{r4} * \texttt{aR 4w}\ \mathit{r4} * \texttt{aPC}\ \mathit{(p{+}4w)})$$

$$\vdash \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 2w}\ \mathit{r2} * \texttt{aR 3w}\ \mathit{r3} * \texttt{aPC}\ p)\ \{(p, \mathit{0xE2432010w})\}$$
$$(\texttt{aR 2w}\ \mathit{(r3 - 16w)} * \texttt{aR 3w}\ \mathit{r3} * \texttt{aPC}\ \mathit{(p{+}4w)})$$

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:

$$\begin{aligned}&\vdash \texttt{SPEC ARM\_MODEL } (\texttt{aR 3w } \mathit{r3} * \texttt{aR 4w } \mathit{r4} * \texttt{aPC } p * \texttt{aR 2w } \mathit{r2})\\ &\quad \{(p, \mathit{0xE1A03004w}); (p + \mathit{4w}, \mathit{0xE2432010w})\}\\ &\quad (\texttt{aR 2w } (\mathit{r4} - \mathit{16w}) * \texttt{aR 3w } \mathit{r4} * \texttt{aPC } (p{+}\mathit{8w}) * \texttt{aR 4w } \mathit{r4})\end{aligned}$$

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: (1) 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 safety properties to hold. (2) Ensure that the Hoare triple theorems for every instruction are encoded with their safety assertions. (3) Define, formally, the requirements for a program to possess our desired safety properties.

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

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

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

Fig. 2. Logic rules for $\mathcal{L}_{LR}$. $\mathit{ms}, \mathit{cfi_1}, \mathit{cfi_2}, \mathit{cfi_3}$ are safety assertions for memory and control-flow isolation respectively. MCSAt is a syntactic label to group safety assertions.

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 (Sect. 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. Also, the second rule which generates safe MEMCFISAFE theorems, 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 safety assertions for every instruction described. 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. 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 (Sect. 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 Sect. 3.1.

$$
\begin{array}{l}
\vdash \forall \mathit{addr}, \mathit{NODES}, \mathit{FUNCS}, \mathit{CFG}_{\mathit{pred}}, \mathit{CFG}_{\mathit{succ}}, \mathit{ICFG}_{\mathit{callpred}}, \mathit{ICFG}_{\mathit{callsucc}}, \\
\quad \mathit{ICFG}_{\mathit{retpred}}, \mathit{ICFG}_{\mathit{retsucc}}, \mathit{assns}_{\mathit{entry}}, \mathit{postcond}_{\mathit{exit}}, \mathit{prestate}, \mathit{poststate} \;\cdot \\
\quad \mathtt{FUN_SAFE}(\mathit{addr}, \mathit{NODES}, \mathit{FUNCS}, \mathit{CFG}_{\mathit{pred}}, \mathit{CFG}_{\mathit{succ}}, \mathit{ICFG}_{\mathit{callpred}}, \mathit{ICFG}_{\mathit{callsucc}}, \\
\qquad\qquad \mathit{ICFG}_{\mathit{retpred}}, \mathit{ICFG}_{\mathit{retsucc}}, \mathit{assns}_{\mathit{entry}}, \mathit{postcond}_{\mathit{exit}}, \mathit{prestate}, \mathit{poststate}) \Leftrightarrow \\
(\quad (\forall \mathit{node} \cdot \mathit{node} \in \mathit{NODES} \Rightarrow (\mathtt{min}(\mathit{node}, \mathit{addr}) = \mathit{addr})) \\
\;\wedge\; (\forall \mathit{min} \cdot \mathit{min} \in \mathit{NODES} \Rightarrow (\mathit{CFG}_{\mathit{pred}}(\mathit{min}) = \emptyset \wedge \mathit{ICFG}_{\mathit{callpred}}(\mathit{min}) = \emptyset \wedge \mathit{ICFG}_{\mathit{retpred}}(\mathit{min}) = \emptyset) \\
\quad \Rightarrow (\forall \mathit{node} \cdot (\mathit{node} \in \mathit{NODES} \Rightarrow \mathit{node} \neq \mathit{min}) \Rightarrow (\mathtt{min}(\mathit{node}, \mathit{min}) = \mathit{min})) \\
\qquad \Rightarrow \exists \mathit{pd}_1, x, c_1, p_1, q_1 \cdot \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_1, \mathit{assns}_{\mathit{entry}}, \mathit{min}, \mathit{node}, x, c_1, p_1, q_1) \wedge \\
\qquad\quad (\mathit{prestate} = \mathtt{aPC}\ \mathit{min} * p_1)) \\
\;\wedge\; (\forall \mathit{out} \cdot \mathit{out} \in \mathit{NODES} \Rightarrow (\mathit{CFG}_{\mathit{succ}}(\mathit{out}) = \emptyset) \\
\quad \Rightarrow (\forall \mathit{funcnode} \cdot (\mathit{funcnode} \in \mathit{FUNCS} \Rightarrow \mathit{out} \notin \mathit{ICFG}_{\mathit{callsucc}}(\mathit{funcnode}))) \\
\qquad \Rightarrow \exists \mathit{pd}_1, \mathit{assn}_1, \mathit{node}, x, c_1, p_1, q_1 \cdot \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_1, \mathit{assn}_1, \mathit{out}, \mathit{node}, x, c_1, p_1, q_1) \wedge \\
\qquad\quad (\mathit{poststate} = q_1) \wedge (\mathit{pd}_1 \Rightarrow \mathit{postcond}_{\mathit{exit}})) \\
\;\wedge\; (\forall \mathit{node}, \mathit{pred} \cdot \mathit{node} \in \mathit{NODES} \Rightarrow \mathit{pred} \in \mathit{CFG}_{\mathit{pred}}(\mathit{node}) \Rightarrow \exists \mathit{pd}_1, \mathit{assn}_1, x, c_1, p, q, \mathit{pd}_2, \\
\qquad \mathit{assn}_2, c_2, r, \mathit{node}' \cdot \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_1, \mathit{assn}_1, \mathit{pred}, \mathit{node}, x, c_1, p, q) \wedge \\
\qquad \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_2, \mathit{assn}_2, \mathit{node}, \mathit{node}', x, c_2, q, r) \wedge (\mathit{pd}_1 \Rightarrow \mathit{assn}_2)) \\
\;\wedge\; (\forall \mathit{node}, \mathit{succ} \cdot \mathit{node} \in \mathit{ICFG}_{\mathit{callsucc}}(\mathit{succ}) \Rightarrow \mathit{succ} \in \mathit{ICFG}_{\mathit{callpred}}(\mathit{node}) \Rightarrow \\
\quad \exists \mathit{pd}_1, \mathit{assn}_1, x, c_1, p, q, \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_1, \mathit{cfg}_2, \mathit{cfg}_3, \mathit{cfg}_4, \mathit{cfg}_5, \mathit{cfg}_6, \mathit{assn}_2, \mathit{pd}_2, r \;\cdot \\
\qquad \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_1, \mathit{assn}_1, \mathit{node}, \mathit{succ}, x, c_1, p, q) \wedge \\
\qquad \mathtt{FUN_SAFE}(\mathit{succ}, \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_1, \mathit{cfg}_2, \mathit{cfg}_3, \mathit{cfg}_4, \mathit{cfg}_5, \mathit{cfg}_6, \mathit{assn}_2, \mathit{pd}_2, q, r) \wedge (\mathit{pd}_1 \Rightarrow \mathit{assn}_2)) \\
\;\wedge\; (\forall \mathit{node}, \mathit{pred} \cdot \mathit{node} \in \mathit{ICFG}_{\mathit{retsucc}}(\mathit{pred}) \Rightarrow \mathit{pred} \in \mathit{ICFG}_{\mathit{retpred}}(\mathit{node}) \Rightarrow \\
\quad \exists \mathit{pd}_1, \mathit{assn}_1, x, c_2, p, q, \mathit{pd}_2, \mathit{assn}_2, r, \mathit{node}', \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_1, \mathit{cfg}_2, \mathit{cfg}_3, \mathit{cfg}_4, \mathit{cfg}_5, \mathit{cfg}_6 \;\cdot \\
\qquad \mathtt{FUN_SAFE}(\mathit{pred}, \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_1, \mathit{cfg}_2, \mathit{cfg}_3, \mathit{cfg}_4, \mathit{cfg}_5, \mathit{cfg}_6, \mathit{assn}_1, \mathit{pd}_1, p, q) \wedge \\
\qquad \mathtt{HOARE_WITH_ASSERT}(\mathit{pd}_2, \mathit{assn}_2, \mathit{node}, \mathit{node}', x, c_2, q, r) \wedge (\mathit{pd}_1 \Rightarrow \mathit{assn}_2)) \quad)
\end{array}
$$

Fig. 3. FSI rule: judgment for interprocedural function safety

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 instruction i_2 to hold, we consider every instruction i_1 that can execute immediately before i_2. The machine-resource values in the post-state of each i_1 must satisfy the safety assertions at i_2. This is analogous to the pre-composition process (Sect. 2.4). As long as the machine-resource values in the post-states of predecessor instructions i_1 enable the safety assertion at i_2 to be true, the safety assertion holds. Also, 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 and boolean conditions of all predecessor instructions.

Safe Function Judgment. We define the FUN_SAFE rule (Fig. 3), which encodes what it means for a function to be safe. This rule encodes our "local reasoning" process for verifying that safety assertions hold. Thus, proving that the machine-code of a given function is safe involves proving that the FUN_SAFE theorem holds for the function. First, we rearrange MEMCFISAFE theorems to form HOARE_WITH_ASSERT theorems, which make explicit the hypotheses (i.e., undischarged safety assertions) of the theorems, and rearrange machine resource expressions into tuples for pattern-matching.

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

A function is comprised of basic blocks of instructions in the function. In a function's intra-procedural control-flow graph (CFG), nodes are basic blocks of the function's instructions, while 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. 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. We walk through each of the 6 conjunct clauses in the FSI rule in Fig. 3.

Arguments to the FUN_SAFE Relation. 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, for a function, the safety assertions $assns_{entry}$, the conditions which hold at its exit $postcond_{exit}$, and the machine resource pre-state $prestate$ and post-state $poststate$.

Function Entry and Exit Specifications. The first clause states that the address of the function is the lowest basic block address for the function. The second clause states that the safety assertions $assns_{entry}$ and pre-state $prestate$ of the function are specified by the entry basic-block of the function. The third clause states that the function's guaranteed exit condition $postcond_{exit}$ and post-state $poststate$ are specified by the exit basic-block of the function.

Intra-procedural Safety Requirements. The fourth 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 instruction 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 fifth and sixth clauses specify the requirements for inter-procedural CFG edges. The fifth 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 ($pd_1 \Rightarrow assn_2$). The sixth 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 ($pd_1 \Rightarrow assn_2$). 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 four 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. Algorithm 1 summarizes the overall workflow of the AUSPICE safety property proof process. First, AUSPICE computes basic blocks and extracts function boundaries from the machine-code of the program (Line 14). Next, AUSPICE obtains the Hoare triple theorems from the Cambridge ARM model for each machine-code instruction (Line 15), adds safety assertions to the Hoare triple theorem for each instruction (Sect. 4.1), and composes the individual instructions' theorems into a single Safe Basic Block theorem for each basic block (Line 16). AUSPICE's proof process takes place on a per-function basis beginning from the entry-function. For each function, all callee functions called by that function are analyzed before the function itself is analyzed (Line 3). Next, AUSPICE applies the Selective Composition tactic (Sect. 4.2) to the safe basic block theorems to propagate branch conditions and function prologue information to the appropriate theorems for the function (Lines 6 and 7). The main process for

Algorithm 1. Overall AUSPICE Workflow

```
1: function SafeFunctionAnalysis(function_name, bb_safe_thms list)
2:     cfg ← Compute Control-Flow Graph for function_name
3:     for all callee functions, callee do
4:         SafeFunctionAnalysis(callee, bb_safe_thms)
5:     end for
6:     bb_safe_thms ← SC-FwdPropagate-BranchConds(bb_safe_thms, cfg)
7:     bb_safe_thms ← SC-FwdPropagate-FuncPrologue(bb_safe_thms, cfg)
8:     assertion_info ← SafetyAssertionAnalysis(bb_safe_thms, cfg)
9:     bb_safe_thms ← AugmentTheorems(bb_safe_thms,assertion_info)
10:    safety_theorem ← FSI_Rule(bb_safe_thms, cfg)
11:    return safety_theorem
12: end function
13: function AUSPICE((addr, instr) list)                ▷ List of machine-code instructions
14:    (bb list) ← Compute basic blocks in program
15:    (bb_instr_thms list) ← Obtain Hoare triple theorem for each instr in each bb
16:    (bb_safe_instr_thms list) ← map ($\lambda x$.AddSafetyAssertions($x$)) bb_instr_thms
17:    bb_safe_thms ← map ($\lambda x$.ComposeSafeInstrs($x$)) bb_safe_instr_thms
18:    return SafeFunctionAnalysis(main,bb_safe_thms)
19: end function
```

discharging safety proof obligations is the `SafetyAssertionAnalysis` function (Line 8), which implements the proof search process using abstract interpretation (Sect. 4.3). Then, the results of the assertion analysis are applied to each of the basic blocks' theorems, and the `FSI_rule` function (Line 10) generates the `FUN_SAFE` safety theorem for the target function being proved to be safe.

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:

$$\begin{aligned}\vdash\ & \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 3w}\ \mathit{r3} * \texttt{aR 2w}\ \mathit{r2} * \texttt{aPC}\ (\mathit{0x81E0}) * \texttt{aMEMORY}\ \mathit{df}\ f\\ & * \ \texttt{cond}((\mathit{r3}\ \&\&\ \mathit{3w} = 0w) \wedge (\mathit{r3} \in \mathit{df})))\,\{(\mathit{0x81E0}, \mathit{0xE5832000w})\}\\ & (\texttt{aR 3w}\ \mathit{r3} * \texttt{aR 2w}\ \mathit{r2} * \texttt{aPC}\ (\mathit{0x81E4}) * \texttt{aMEMORY}\ \mathit{df}\ ((\mathit{r3} = +\mathit{r2})\ f))\end{aligned}$$

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 $\{r3+3; r3+2; r3+1; r3\} \subseteq \{addr \mid \texttt{0x85F8} \le addr \wedge addr \le \texttt{0xBF000000}\}$ 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 $\exists pc.pc = \texttt{0x81E4}\ \wedge\ pc \in$ $\{addr \mid \texttt{0x80B4} \le addr \wedge addr \le \texttt{0x85F4}\}$, 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

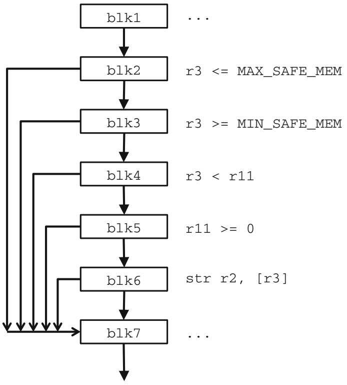


Fig. 4. Possible structure for program with safe `str r2 [r3]`.

Next, we discuss the steps for automatically proving that safety properties hold using $\mathcal{L}_{LR}$. After augmenting single instruction theorems with safety assertions (Sect. 3.1) and obtaining safe basic block theorems (Sect. 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 a `HOARE_WITH_ASSERT` theorem for safe basic blocks or a `FUN_SAFE` theorem for safe functions. We also need to prove that the pre-condition pd_1 of each predecessor CFG node discharges the safety assertion $assn_2$ in the successor CFG node.

From Sect. 4.1, we can see that the safety assertion at each instruction contains three conjuncts: one for memory-isolation and two for

Algorithm 2. Selective Composition: Branch-condition Forward Propagation

```
function SC-FwdPropagate-BranchConds($bb\_safe\_thms$ list)
    $info$ map ← $\emptyset$                    ▷ Conditions to propagate to each CFG node
    procedure PropagateOneStep($info$ map, $last\_info$ map, $cfg$)
        for all $node \in cfg$ do
            $curr\_node\_preds$ ← FindPreds($cfg$,$node$)
            $pred\_preconds$ ← ($map$ ($\lambda x$.GetThmPreconds(x)) $curr\_node\_preds$)
            $last\_info\_preconds$ ← ($map$ ($\lambda x.last\_info[x]$) $curr\_node\_preds$)
            if $length(curr\_node\_preds) == 1$ then
                $info[node] = pred\_preconds \bigcup last\_info\_preconds$
            end if
        end for
    end procedure
    repeat
        $last\_info$ ← $info$
        $info$ ← PropagateOneStep($info$, $last\_info$, $cfg$)
    until $last\_info == info$
    return $info$
end function
```

control-flow isolation. 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 Sect. 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 Sect. 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 (Sect. 2.4) forward to successor theorems under certain conditions.

To illustrate the process of selective composition, consider, for example, the store instruction `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 discharge 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 apply the pre-composition tactic (Sect. 2.4) for basic blocks $blk_2, blk_3, blk_4, blk_5$. This lets us 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 carry 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}.

Algorithm 2 describes the Selective Composition tactic for the forward propagation of branch-conditions in pseudocode. The tactic uses a fixed-point intraprocedural static-analysis over the Hoare triple theorems of a function. The static-analysis identifies branch-conditions to propagate forward from each theorem to its successor theorems (Lines 3 to 16; FindPreds returns the predecessors for a given node in the CFG of the function, while GetThmPreconds returns the pre-conditions for a given Hoare triple theorem). The analysis also ensures that branch-conditions are propagated forward only when the target node has only one predecessor in the CFG (Line 8). The analysis returns the branch-conditions to add to each node's theorem in the program's CFG (Line 17).

Local Use of Global Information. Next, we describe the second instance of *selective composition.* Recall that for control-flow isolation, 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:

$$\vdash \texttt{SPEC ARM\_MODEL}\ (\texttt{aPC}\ p * \texttt{aR 13w}\ r13 * \texttt{aMEMORY}\ df\ f\ *\ \texttt{cond}(((f\ r13)\ \&\&\ 3w = 0w) \wedge$$
$$((f\ r13) \in df)))\ \{(p,\ 0xE8BD8000w)\}\ (\texttt{aPC}\ (f\ r13) * \texttt{aR 13w}\ (r13{+}4w) * \texttt{aMEMORY}\ df\ f)$$

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:

$$\begin{aligned}\vdash\ & \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 14w}\ \mathit{r14} * \texttt{aR 13w}\ \mathit{r13} * \texttt{aPC}\ p * \texttt{aMEMORY}\ \mathit{df}\ f\\ & \texttt{cond}(((f\ (\mathit{r13} - \mathit{4w}))\ \&\&\ \mathit{3w} = 0w) \wedge ((f\ (\mathit{r13} - \mathit{4w})) \in \mathit{df})))\ \{(p, \mathit{0xE92D4000w})\}\\ & (\texttt{aR 14w}\ \mathit{r14} * \texttt{aR 13w}\ \mathit{(r13 - 4w)} *\ \texttt{aPC}\ \mathit{(p{+}4w)} * \texttt{aMEMORY}\ \mathit{df}\ ((\mathit{r13} - \mathit{4w} = {+}\mathit{r14})\ f))\end{aligned}$$

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. As long as the prologue precedes every instruction in the function, and the function does not alter the callee-saved registers until its epilogue, this substitution is valid. 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. Unlike the forward-propagation of branch-conditions, a fixed-point analysis is not required, and we directly substitute the information from the function prologue in every subsequent basic block in the function.

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 [7] 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 the CFG of a function in reverse topological order in each iteration. Each CFG node is a basic block in the function, and each node is associated with a safe basic-block theorem (Sect. 3.2). 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's theorem. For safety assertions which the predecessor node's theorem cannot discharge, our analysis adds the assertion to the predecessor node's theorem, propagating the assertion backwards up the CFG. Our analysis

Algorithm 3. Safety Assertion Analysis

```
 1: function SafetyAssertionAnalysis(bb_safe_thms map, cfg)
 2:     info map ← $\emptyset$
 3:     procedure AssertionAnalysisStep(info map, last_info map, cfg)
 4:         for all node ∈ cfg do
 5:             for all pred ∈ FindPreds(cfg, node) do
 6:                 pred_preconds ← GetThmPreconds(pred) $\bigcup$ last_info[pred]
 7:                 node_asserts ← GetThmAsserts(node) $\bigcup$ last_info[node]
 8:                 for all assert ∈ node_asserts do
 9:                     if PROVE(pred_preconds, assert) == False then
10:                         info.term[pred] ← info.term[pred] $\bigcup$ assert
11:                         a_path ← FindAssertPath(last_info.path[node], assert)
12:                         info.path[pred] ← info.path[pred] $\bigcup$ a_path
13:                         AbortIfAssertPathIsCycle(a_path)
14:                     end if
15:                 end for
16:             end for
17:         end for
18:     end procedure
19:     repeat
20:         last_info ← info; info ← AssertionAnalysisStep(info, last_info, cfg)
21:     until last_info == info
22:     return info
23: end function
```

is also inter-procedural, and context-sensitive. Each function is summarized at its call-site by a `FUN_SAFE` theorem for that particular call-site.

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: the free variable instantiation at loop boundaries will generate new safety assertions to be framed whenever the assertion is propagated across the back-edge of the loop. 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 on this path. Then, we inform the user that we are unable to prove our safety properties for the program.

Algorithm 3 describes our static-analysis algorithm. `GetThmPreconds` (Line 6) and `GetThmAsserts` (Line 7) return the pure boolean conditions in the pre-state and the safety assertion at a node's theorem respectively. `PROVE` tries to discharge the given safety assertion, *assert*, using the given conditions *pred_preconds* from the predecessor theorem, and returns true if it can discharge the safety assertion, and false otherwise (Line 9). If the safety assertion cannot be discharged, it is added to the analysis information for the node's predecessor node (Line 10), so that it will be framed to the predecessor node's theorem after the analysis. The analysis information also records the path along which each assertion is propagated in *info.path* (Line 12). Then, the analysis checks if there is a cycle along the propagation path of the assertion (Line 13) in the function `AbortIfAssertPathIsCycle`, and terminates the AUSPICE proof process if a cycle is found. This is because if a cycle is found along which the pre-composition tactic causes the safety assertion term to change with each iteration, the analysis is likely to not terminate as it will keep adding new safety assertion terms to the analysis information on each successive iteration of the analysis.

5 Discussion

Soundness of Proof Rules. AUSPICE's proof rules for single instruction (Sect. 3.1) and basic block (Sect. 3.2) safety are sound, because we derive our `MEM_CFI_SAFE`, `MEM_CFI_SAFE_COMPOSE`, and `MEMCFISAFE_FRAME` proof rules from the Hoare triples for machine-code programs in the Cambridge model, which Myreen et al. have shown to be sound [16]. Also, using the HOL4 proof assistant to define our proof rules further ensures they are sound. In addition, we proved (Sect. 3.1) that safe single instruction and basic block theorems in AUSPICE derived from our proof rules have the same instruction semantics as the ARM machine-code semantics defined by the trustworthy, validated Cambridge ARM model [9,15].

Correctness of Safety Rule. Next, we give a brief, informal argument of the correctness of our proof rule for safe programs. The `FUN_SAFE` theorem (Fig. 3) can be proven for a program if and only if safety assertions are specified for every instruction, and if these safety assertions hold before that instruction begins executing (except for the first instruction, which relies on the OS to correctly initialize the processor state for the program). We argue this by Structural Induction on the Control-Flow Graph (CFG) of a program. Each node in our CFG of a function in a program is either a single-entry, single-exit basic block with sequentially executing code, or a (callee) function called by the function.

Base Case. The `MEM_CFI_SAFE` rule (Sect. 3.1) ensures every instruction's theorem contains our safety assertions (Sect. 2.2). The `MEM_CFI_SAFE_COMPOSE` rule ensures every basic block's theorem is built up only from single-instruction theorems with added safety assertions. The requirement that post-states of predecessor theorems and pre-states of successor theorems must be equal in `MEM_CFI_SAFE_COMPOSE` ensures every basic block's theorem accumulates the safety assertions for every composed safe instruction theorem. Then, for a program with only a single instruction or basic-block, if the OS correctly initializes the processor state, the safety assertions will hold for the single instruction or single basic block.

Inductive Case. We take the CFG of a function, G, and partition its vertices into a single vertex, g, and all other vertices, G'. By the Inductive Hypothesis, the `FUN_SAFE` theorem holds for G'. Then, consider the edges E connecting G' to g. In the absence of function pointers and unstructured jumps (`longjmp`), the edges E are either (i) intra-procedural control-flow transfers between basic blocks in the function, (ii) function calls from a basic block in the function to a callee function, or (iii) function returns from a callee function to a basic block in the function. Then, for `FUN_SAFE` to be true, the fourth to sixth conjunct clauses of the `FUN_SAFE` rule must be true, so the pre-conditions of the theorems of all predecessor vertices to g in the CFG discharge the safety assertions at g, making the safety assertions at g hold, for any type of possible control-flow transfer to g. Thus, our `FUN_SAFE` rule ensures that we have captured all the possible control-flow transfers in a machine-code program. For `FUN_SAFE` to be correct, we require correct CFG predecessor and successor maps, which are straightforward to compute without function pointers and unstructured jumps.

Limitations. Our machine-code safety properties (Sect. 2.2) have been formulated to ensure they can be automatically proven to hold in machine-code programs. These properties are sufficient, but not strictly necessary to meet our high-level goal of preventing the control-flow of a machine-code program from being hijacked due to user input. The strictness of our safety properties helps automate the verification process. Our requirement that machine-code instructions do not alter memory addresses greater than the current function's frame pointer address also prevents functions from modifying any variables passed by reference from the stacks of caller functions. Instead, functions passing data by reference must store this data either on the program's heap, or use memory allocated in the program's `data` section. We believe this is a small inconvenience to enable the fully automated verification of safety properties.

6 Evaluation

We aim to show that we can verify real-world programs, and we pick programs with constructs which are challenging to verify. We also measure our runtime to show the feasibility of our verification. Our test programs were compiled using an unmodified `gcc` toolchain for the ARMv7 architecture with `-O0` optimization. Figure 5 summarizes our test programs. `sort` implements Bubble Sort, which has a doubly-nested loop, and also contains 2 other functions to exercise our inter-procedural analysis. `memcpy` is an implementation of 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 [10], and it implements the Boyer-Moore string search algorithm, and demonstrates our verification on real-world programs.

Test Program	Instructions	Functions	Description
`memcpy`	116	2	Real-world `memcpy`
`sort`	337	5	Nested loops, function calls/returns
`stringsearch`	530	5	Boyer-Moore string search (MiBench [10])

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

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 majority of

	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.

the verification time is spent in the abstract interpretation (Sect. 4.3) and the proof of the safe function theorem (Sect. 3.3). We believe these are feasible times for verifying safety properties, as programs only need to be verified once on installation.

7 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 [18,26], but specifications must be manually annotated in programs, and verification is interactive. Tan and Appel [21] developed a program logic for multi-entry, multi-exit machine-code fragments to reason about unstructured control-flows in executables in Foundational Proof Carrying Code (FPCC). They require a special compiler to generate machine-code annotated with types [13], while we verify unmodified executables compiled using an off-the-shelf compiler. iTalX [22] infers types for x86 assembly programs, reducing the amount of type annotations required from a modified compiler. Executables have also been verified without using a program logic, although concise theorems cannot be proven. Bedrock [6] provides "mostly-automated" verification for generic program properties, and provides memory safety as a side-effect, for programs written using its idealized machine language, from which concrete architectures can be targeted. Xu et al. [25] verify safety properties for machine-code using static-analysis. RevGen [5] decompiles machine-code to the intermediate representation of the LLVM compiler framework, enabling other analyses to be reused, whereas we use a validated model of ARM machine-code. Thakur et al. [23] perform model-checking on machine-code without requiring a precomputed, fixed, inter-procedural CFG. Sequoll [4] also performs model-checking on machine-code programs, and like our work, uses the Cambridge ARM model [15], but it uses temporal logic to reason about worst-case execution time (WCET) in the NuSMV model-checker, whereas our approach uses Hoare logic. XFI [8] and ARMor [27], are software fault isolation (SFI) [24] implementations which ensure and verify that (x86 and ARM, respectively) executables possess memory and control-flow safety properties. XFI requires modules being verified to be annotated with hints. PittSFIeld [12] verifies that its SFI safety rewriting for x86 binaries is correct, as opposed to verifying that the executables it produces are safe. RockSalt [14] also provides verified SFI by providing a verified checker which checks that programs are isolated, whereas our work produces a safety proof for each program. ARMor [27] 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 using our logic framework and *selective composition* proof tactic.

8 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 [15,16], 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 [10]) examples. In future, we intend to validate our approach on more programs, and expand our verification to programs with system calls.

Acknowledgment. We thank Lu Zhao for his help with ARMor [27], Magnus Myreen for his help with the Cambridge ARM model [15,16], and Xinyu Zhuang for his feedback.

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. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow Integrity. In: ACM CCS (2005)
4. Blackham, B., Heiser, G.: Sequel: a framework for model checking binaries. In: IEEE RTAS (2013)
5. Chipounov, V., Candea, G.: Enabling sophisticated analyses of x86 binaries with RevGen. In: HotDep (2011)
6. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI (2011)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
8. Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., Necula, G.: XFI: software guards for system address spaces. In: OSDI (2006)
9. Fox, A.: Formal specification and verification of ARM6. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 25–40. Springer, Heidelberg (2003)
10. 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)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)

12. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: USENIX Security (2006)
13. 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)
14. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: PLDI (2012)
15. Myreen, M.O., Fox, A.C.J., Gordon, M.J.C.: Hoare logic for ARM machine code. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 272–286. Springer, Heidelberg (2007)
16. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 568–582. Springer, Heidelberg (2007)
17. Myreen, M., Gordon, M., Slind, K.: Machine-code verification for multiple architectures: an application of decompilation into logic. In: FMCAD (2008)
18. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL (2006)
19. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: IEEE LICS (2002)
20. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
21. Tan, G., Appel, A.W.: A compositional logic for control flow. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 80–94. Springer, Heidelberg (2006)
22. Tate, R., Chen, J., Hawblitzel, C.: Inferable object-oriented typed assembly language. In: PLDI (2010)
23. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)
24. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient software-based fault isolation. In: SOSP (1993)
25. Xu, Z., Miller, B., Reps, T.: Safety checking of machine code. In: PLDI (2000)
26. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for PCC: dynamic storage allocation. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 363–379. Springer, Heidelberg (2003)
27. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: ARMor: fully verified software fault isolation. In: EMSOFT (2011)