AUSPICE-R: Automatic Safety-Property Proofs for Realistic Features in Machine Code

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

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

Abstract. Automatically generating proofs of safety properties for software is important as software becomes safety-critical, e.g., in medical devices and automobiles. While current techniques can automatically prove safety properties for machine code, they either: (i) do not support user-mode programs in an operating system, (ii) do not support realistic program features such as system calls, or (iii) have been demonstrated only on programs of limited sizes. We present AUSPICE-R, which automates safety-property proof generation for user-mode ARM machine code containing system calls, and greatly improves the scalability of automated safety-property proof generation. AUSPICE-R uses an axiomatic approach to model system calls, and leverages idioms in compiled code to optimize its proof automation. We demonstrate AUSPICE-R on (i) simple working versions of common text utilities that perform I/O, and (ii) embedded programs for the Raspberry Pi single-board-computer containing hardware I/O. AUSPICE-R automatically proves safety up to 12x faster, and supports programs 3x larger, than prior techniques.

1 Introduction

Interactive theorem proving (ITP) is a promising approach for reasoning about programs, as it produces succinct proofs. While ITP has required manual user inputs in "heavy-weight" [21] proofs of functional correctness, recent work [27, 24] has automated "light-weight" proofs for single classes of safety properties (e.g., Software Fault Isolation (SFI) [25], Control-Flow Integrity (CFI) [6]) using ITP for machine code, eliminating the need for manual user inputs. Reasoning about machine code provides a foundational approach for verification, as machine code proofs are not affected by miscompilation bugs that may cause safety problems [26], as compared to proofs about source-code. However, current approaches for automating safety proofs for machine code are limited: they either target embedded programs running directly on a processor without an operating system (OS) [27], or they do not support proofs for user-mode machine code containing system calls (syscalls) [24]. As embedded systems become more powerful, it is increasingly common for them to run full-fledged OSes. Then, applications run as user-mode programs [17], which need syscalls to perform useful tasks. In addition, current approaches are limited in the scale of programs for which they can feasibly generate safety proofs, due to their long proof times.

Modeling and proving safety properties about syscalls in machine code is challenging as syscall behavior occurs in two processor modes: in the user-mode where the syscall is invoked, and in the supervisor mode where the syscall is serviced by an OS kernel. However, our safety properties focus on user-mode behavior, and OS kernels are complex. Hence, we wish to ensure that our safety proofs for user-mode machine code are modular, and avoid needing to prove that syscalls are correctly serviced by the OS kernel. In addition, current approaches for automating safety proofs face scalability challenges, due to the large number of logic terms manipulated. Hence, we wish to improve the scalability of automatically generating safety proofs by reducing the computation required to generate a safety proof.

In this paper, we present AUSPICE-R, an automated safety-property proof-generation framework (for CFI [6]) for user-mode ARM machine code that supports syscalls, and scales up to larger programs than prior techniques. AUSPICE-R extends our earlier work, AUSPICE [24]. First, to ensure modularity in its safety proofs for machine code with syscalls, AUSPICE-R treats syscalls as *black-boxes*: We focus on the inputs to and the user-mode-visible effects of syscalls; we model the effects of syscalls using axioms that capture the specified (e.g., in the syscall API of the OS) behavior of the syscall. This lets us reason about syscalls in user-mode machine code without having to verify the behavior of the underlying OS. Second, we optimize the proof automation in AUSPICE-R to improve the times taken to prove safety, and to increase the size of programs for which safety can be proved. These optimizations leverage common conventions found in `gcc`-emitted machine code to speed up AUSPICE's analysis.

Our contributions are: (i) an axiomatic approach to modeling syscall behavior with the goal of automating safety property proofs, (ii) a delayed algorithm for performing safety property analysis to support syscalls, (iii) optimizations to AUSPICE-R's analysis that leverage idioms in compiled machine code, and (iv) an evaluation of AUSPICE-R on programs containing syscalls that perform both file and hardware I/O, that are significantly larger than in prior techniques.

2 Problem Statement

Goals. AUSPICE-R's goals are to: (i) fully automate safety property proofs for machine code containing system call (syscall) invocations, (ii) formalize the user-mode-visible effects of a syscall while assuming that the underlying OS services the syscall "correctly" (we discuss "correct" next), (iii) construct a formalization that is sound with respect to the trustworthy Hoare Logic for ARM machine code [18, 19] that we build on, and (iv) work with programs compiled by unmodified commodity compilers (e.g., `gcc`), i.e., we disallow compiler modifications.

Scope. In this work, we target machine code programs for the ARM platform, as ARM is the dominant platform for embedded systems [4]. We consider programs that run in user-mode on the Linux operating system (OS) in this work, and we focus specifically on the safety property of Control-Flow Integrity (CFI) [6].

In this work, we describe how to automate safety proofs for machine code with syscalls; we address how to enforce these safety properties in source-code in [23].

Assumptions. AUSPICE-R extends AUSPICE [24], which builds on the Hoare Logic for ARM machine code developed at Cambridge University [18, 19] (which we refer to as the Cambridge ARM model). Hence, safety proofs in AUSPICE-R inherit some of the assumptions and limitations of AUSPICE and the Cambridge ARM model. Specifically, we assume that our target machine code programs:

1. Have behavior that is not affected by hardware exceptions, interrupts, or page-table operations (not modeled by Cambridge ARM model),
2. Do not contain recursive function calls (unsupported by AUSPICE),
3. Have no floating-point instructions (not modeled by Cambridge ARM model),
4. Have no `goto` nor `longjmp` statements (unsupported by AUSPICE),
5. Do not contain explicit function pointers (unsupported by AUSPICE),
6. Contain only sequential execution behavior (multi-threaded behavior and concurrency are not modeled by the Cambridge ARM model),
7. Are statically compiled and linked, so that all executable code is present,
8. Are compiled with an unmodified version of `gcc` at `-O0` optimization, and obey the ARM-THUMB Procedure Call Standard (ATPCS) [3], and
9. Have well-defined function prologues and epilogues.

We also assume that our target programs run in an OS that isolates user processes, preventing attackers from modifying the memory of a process, and that the OS and physical security of the host are not compromised. We assume that the underlying OS (Linux in this paper) "correctly" services syscall invocations by correctly restoring the context (program counter, register, and memory) of the user process at the end of every syscall invocation, and providing its specified (e.g., in Section 2 of the Linux Programmer's Manual [2]) functionality. Recent work has verified the functional correctness of microkernels for realistic OSes [11], making this assumption a realistic one. We also assume that each syscall is invoked via an assembly wrapper with a C function prototype that sets up the arguments for the syscall, whose name identifies the invoked syscall. This is the convention by which common C libraries give programmers access to syscalls.

Non-Goals. We do not verify that the OS correctly services syscalls. We consider only direct (e.g., register and memory) effects of syscalls on CFI, but not OS state not directly observed by a user-mode process (e.g., file descriptor mappings, user-space memory mappings in `mmap`). We do not verify arbitrary safety properties: we focus on the safety properties in §3.1 that ensure CFI.

3 Background

We summarize our prior work, AUSPICE [24], before we present AUSPICE-R's extensions. We describe the safety properties proved automatically by AUSPICE (§3.1) and the Cambridge ARM model which AUSPICE is based on (§3.2), and AUSPICE's proof rules (§3.3) and proof automation algorithm (§3.4).

3.1 Safety Properties of Interest

The main goal of AUSPICE is to prove that a machine code program possesses Control-Flow Integrity (CFI) [6]. CFI requires that the execution of a program follows a path in a Control-flow Graph (CFG) that is "determined ahead of time" [6]. For a program which has CFI with respect to its load-time CFG (i.e., its execution follows the CFG describing the instructions loaded from disk), attackers cannot change the program's execution in unintended ways (e.g., by supplying malformed or malicious inputs), nor inject instructions to be executed.

AUSPICE proves CFI for a program by proving three safety properties: that (1) loaded program instructions in memory cannot be overwritten, (2) function-return addresses saved to the program's stack cannot be overwritten, and (3) only instructions at initially loaded addresses can be executed. AUSPICE instantiates these safety properties as safety assertions at each instruction to be proved.

The above three safety properties are necessary and sufficient (in the absence of `goto` and `longjmp` statements in C and explicit function pointers) to ensure CFI holds for an ARM machine code program. To see why this is the case, consider how the three safety properties together prevent a machine code program's CFG from being changed at run-time: Property (1) prevents CFG nodes from being changed by preventing loaded instructions in memory from being modified; Property (2) prevents CFG edges from being changed by preventing function return addresses from being changed; Property (3) prevents CFG nodes from being added by preventing the injection and running of new instructions. Hence, the 3 safety properties that AUSPICE proves for each instruction in a program are sufficient to prove that CFI holds (given our assumptions in §2).

3.2 Hoare Logic for ARM Machine Code: Cambridge ARM Model

Next, we describe the Hoare Logic we use to reason about ARM machine code. The Cambridge ARM model [18] specializes a Hoare Logic [19] to reason about low-level details of ARM machine code. This model is formalized in higher-order logic and mechanized in the HOL4 [22] proof system, and captures low-level details of processor state as seen by ARM machine code programs, namely: values of registers and status flags, data values stored in memory, and the value of the program counter (`pc`). The model represents the behavior of each instruction using a Hoare triple theorem:

$$\vdash \texttt{SPEC}\ x\ \{\texttt{p}\}\ \texttt{c}\ \{\texttt{q}\}$$

"`SPEC`" indicates the theorem is a Hoare triple; "x" is a tuple that defines the next-step relation and other relations for the instruction set architecture (ISA) modeled in the triple, and is instantiated with the "`ARM_MODEL`" tuple of relations in the Cambridge ARM model [18] (other ISAs are also supported in the Cambridge model [19], but are beyond the scope of this paper). Informally, the theorem reads: if assertion `p` holds for the current processor state, and instruction `c` is executed, then `q` will hold for the resulting processor state. We refer to `p` and `q` as the pre-state and post-state assertions of instruction `c`.

Processor state assertions `p` and `q` either assert the value of a processor state element (namely register values, status flags, memory, and the program counter value), or are pure boolean assertions about logical variables. Pure boolean assertions can be *pre-conditions* (labelled `precond`$(\cdot)$), which are predicates known to hold before an instruction executes (e.g., statements in the body of "`if (i == 0) { ... }`" have the pre-condition "$i = 0$"), or *assumptions* (labelled `cond`$(\cdot)$).

State assertions can assert the values of multiple resources (e.g., multiple registers) using the separating conjunction $*$ [20]. Note that $*$ in the Cambridge ARM model prevents assertions about repeated processor resources (e.g., the same register cannot be asserted about twice), but not memory locations. Instead, processor memory is treated as a single resource in the model. Memory is represented as a map from 32-bit addresses to the bytes stored at each address. AUSPICE uses the following proved rules from the Cambridge ARM model:

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

Note that there are no side-conditions restricting the form of r in the Frame rule above, as machine resource values are asserted using register relations (see §4.1) rather than variables in the Hoare logic of the Cambridge ARM model [19], effectively turning all symbolic variables into single-static assignment variables.

3.3 AUSPICE: Hoare Logic-based Safety Property Proofs

Next, we describe AUSPICE's proof rules which enable safety properties to be proved automatically. Then, we describe AUSPICE's abstract interpretation algorithm for automating safety proofs.

AUSPICE defines proof rules to hierarchically build up, in a bottom-up fashion, to a whole-program definition of safety with respect to its 3 safety properties. First, AUSPICE defines proof rules for its safety properties to hold at the single instruction and basic block levels (Fig. 1). The `MEM_CFI_SAFE` rule constructs a "safe instruction" theorem by requiring each instruction's Hoare triple to be augmented with safety assertions (as assumptions) for AUSPICE's three safety properties. Our three safety properties (§3.1) are concretely instantiated in the predicates *ms*, cfi_1, cfi_2 respectively with the safe ranges for memory addresses written to, and the value of the program counter after each instruction runs. The code in the `MEM_CFI_SAFE` rule, "$\{(\mathit{offset}, \mathit{ins})\}$", enforces that safe instruction theorems can be constructed only from a Hoare triple for a single instruction with instruction word "*ins*" at address "*offset*" in the program. Then, the `MEM_CFI_SAFE_COMPOSE` rule builds up to a "safe basic block" theorem by allowing only theorems for safe instructions, and theorems composed from smaller safe basic blocks, to be composed. The `MEMCFISAFE_FRAME` rule lifts the `FRAME` rule in the Cambridge ARM model to reason about safe basic blocks.

Each `MEMCFISAFE` (i.e., safe instruction and safe basic block) theorem is also sound with respect to the Cambridge ARM model, as AUSPICE proved:

$$\vdash \forall x\ p\ c\ q\ \cdot\ \texttt{MEMCFISAFE}\ x\ p\ c\ q\ \Rightarrow \texttt{SPEC}\ x\ p\ c\ q$$

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

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

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

Fig. 1. AUSPICE logic rules for single instruction and basic block level safety.

Next, AUSPICE defines the `FUNSAFE` rule (Fig. 2), which enables *local reasoning* about safety properties at the function (and whole-program) levels. AUSPICE's local reasoning principle [24] states that the safety properties at each instruction depend only on the program state immediately before that instruction runs: thus, for a program to be safe, we only need to ensure that the safety assertions at each instruction hold given the pre-conditions of all its predecessor instructions. Informally, if the `FUN_SAFE` theorem for a function holds, then the machine code of the function is safe with respect to AUSPICE's three safety properties. The safety of a (machine code) function is defined by the `FUN_SAFE` relation, with respect to: (i) *addr*, the address of the function, (ii) *nodes*, a set of addresses of the function's CFG nodes (i.e., its basic blocks), (iii) *funcs*, a set of addresses of callee functions, (iv) $\mathit{cfg}_{\mathit{pred}}, \mathit{cfg}_{\mathit{succ}}$, maps of CFG predecessors/successors of each node, (v) *assns*, the safety assertions of the function's entry node, (vi) *postcond*, the pre-conditions of the function's exit node, and *p*,*q*, the pre-state/post-state of the function's entry/exit nodes respectively.

Then, the 6 conjunct clauses of the `FUNSAFE` rule specify the requirements that need to hold for the function to be safe. The requirements for the function to be safe are instantiated according to the function's CFG. The first 3 conjunct clauses define: (i) the address of the function, as given by its entry node with the smallest address, (ii) the entry node of the function to have no CFG predecessors, and (iii) the exit node of the function to have no CFG successors.

The 4th to 6th conjunct clauses define the requirements for the function to be safe for all control-flow transfers, that are either: (i) intra-procedural, (ii) inter-procedural function calls, or (iii) inter-procedural function returns. The `FUNSAFE` rule is instantiated with one clause for each CFG edge. Each conjunct clause begins with a description of the CFG predecessor/successor relationships for the kind of control-flow transfer described (e.g., $\forall n, \mathit{pred}, \mathit{succ} \cdot \{n, \mathit{pred}\} \subseteq \mathit{nodes} \Rightarrow \mathit{pred} \in \mathit{cfg}_{\mathit{pred}}(n) \Rightarrow n \in \mathit{cfg}_{\mathit{pred}}(\mathit{succ})$ for intra-procedural control-flow transfers). In each conjunct clause, the `MEMCFISAFE` and `FUN_SAFE` terms describe the behavior of the basic blocks or functions in the clause, and the predicate $r' \Rightarrow s'$ is the requirement for the pre-conditions r' of each predecessor basic block to discharge the safety assertions s' at each basic block. Note also

$$
\begin{aligned}
\vdash\quad & \forall \mathit{addr}, \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_{\mathit{pred}}, \mathit{cfg}_{\mathit{succ}}, \mathit{assns}, \mathit{postcond}, p, q \ \cdot \\
& \texttt{FUN\_SAFE}(\mathit{addr}, \mathit{nodes}, \mathit{funcs}, \mathit{cfg}_{\mathit{pred}}, \mathit{cfg}_{\mathit{succ}}, \mathit{assns}, \mathit{postcond}, p, q) \\
\Leftrightarrow\ & (\forall n \cdot n \in \mathit{nodes} \Rightarrow \texttt{min}(n, \mathit{addr}) = \mathit{addr}) \\
\bigwedge\ & (\forall \mathit{min} \cdot \mathit{min} \in \mathit{nodes} \Rightarrow (\mathit{cfg}_{\mathit{pred}}(\mathit{min}) = \emptyset) \Rightarrow \exists x, p', q', c' \cdot \\
& \quad \big(\texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ \mathit{min} * \texttt{cond}(\mathit{assns}) * p'\big)\ c'\ \big(q'\big)\big)) \\
\bigwedge\ & (\forall \mathit{out} \cdot \mathit{out} \in \mathit{nodes} \Rightarrow (\mathit{cfg}_{\mathit{succ}}(\mathit{out}) = \emptyset) \Rightarrow \exists x, p', q', c', r' \cdot \\
& \quad \big(\texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ \mathit{out} * \texttt{precond}(r') * p'\big)\ c'\ \big(q'\big)\big) \wedge (r' \Rightarrow \mathit{postcond})) \\
\bigwedge\ & (\forall n, \mathit{pred}, \mathit{succ} \cdot (\{n, \mathit{pred}\} \subseteq \mathit{nodes} \Rightarrow \mathit{pred} \in \mathit{cfg}_{\mathit{pred}}(n) \Rightarrow n \in \mathit{cfg}_{\mathit{pred}}(\mathit{succ}) \Rightarrow \\
& \exists x, r', p', c_1, q', s', c_2, q'', r' \cdot \\
& \quad \big(\texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ \mathit{pred} * \texttt{precond}(r') * p'\big)\ c_1\ \big(\texttt{aPC}\ n * q'\big) \wedge \\
& \quad\ \texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ n * \texttt{cond}(s') * q'\big)\ c_2\ \big(\texttt{aPC}\ \mathit{succ} * q''\big) \wedge (r' \Rightarrow s')))) \\
\bigwedge\ & (\forall n, \mathit{pred} \cdot (\mathit{pred} \in \mathit{nodes} \Rightarrow (n \in \mathit{funcs}) \Rightarrow \mathit{pred} \in \mathit{cfg}_{\mathit{pred}}(n) \Rightarrow n \in \mathit{cfg}_{\mathit{succ}}(\mathit{pred}) \Rightarrow \\
& \exists x, r', p', q', \mathit{nodes}', \mathit{funcs}', \mathit{cfg}'_p, \mathit{cfg}'_s, s', \mathit{postcond}', q'', c_1 \cdot \\
& \quad \big(\texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ \mathit{pred} * \texttt{precond}(r') * p'\big)\ c_1\ \big(\texttt{aPC}\ n * q'\big) \wedge \\
& \quad\ \texttt{FUN\_SAFE}(n, \mathit{nodes}', \mathit{funcs}', \mathit{cfg}'_p, \mathit{cfg}'_s, s', \mathit{postcond}', q', q'') \wedge (r' \Rightarrow s')))) \\
\bigwedge\ & (\forall n, \mathit{pred}, \mathit{succ} \cdot (n \in \mathit{nodes} \Rightarrow (\mathit{pred} \in \mathit{funcs}) \Rightarrow \mathit{pred} \in \mathit{cfg}_{\mathit{pred}}(n) \Rightarrow n \in \mathit{cfg}_{\mathit{pred}}(\mathit{succ}) \Rightarrow \\
& \exists x, \mathit{nodes}', \mathit{funcs}', \mathit{cfg}'_p, \mathit{cfg}'_s, s', \mathit{postcond}', p', q', s'', q'', c_1 \cdot \\
& \quad (\texttt{FUN\_SAFE}(\mathit{pred}, \mathit{nodes}', \mathit{funcs}', \mathit{cfg}'_p, \mathit{cfg}'_s, s', \mathit{postcond}', p', q') \wedge \\
& \quad\ \texttt{MEMCFISAFE}\ x\ \ \big(\texttt{aPC}\ n * \texttt{cond}(s'') * q'\big)\ c_1\ \big(\texttt{aPC}\ \mathit{succ} * q''\big) \wedge (\mathit{postcond}' \Rightarrow s''))))
\end{aligned}
$$

Fig. 2. AUSPICE's `FUNSAFE` rule for function-level and whole-program safety. `aPC` asserts that the program counter contains the asserted value.

that in each of the first two conjuncts of each requirement, the post-state of the predecessor CFG node's `MEMCFISAFE` or `FUN_SAFE` which describes its behavior must match the pre-state of the successor CFG node's `MEMCFISAFE` or `FUN_SAFE`. This is in line with the standard `COMPOSE` rule in Hoare Logic.

Thus, the goal of the `FUN_SAFE` theorem is to state that the machine code of a function (and all its callee functions) possesses the three AUSPICE safety properties at each instruction, which in turn implies that the program has CFI. The soundness and correctness arguments for our proof rules are in [24].

3.4 Proof Automation in AUSPICE

Next, we describe AUSPICE's proof automation algorithm (Fig. 3). At the top level, AUSPICE calls SafeFunctionAnalysis for the entry-function of the program. SafeFunctionAnalysis is a context-sensitive inter-procedural analysis which returns a `FUN_SAFE` theorem for a function proved safe, or terminates with an error message. SafeFunctionAnalysis calls the abstract interpretation in SafetyAssertionAnalysis (Fig. 4). This abstract interpretation is a backwards analysis, whose domain is predicates about processor state. The analysis finds the pre-conditions needed to discharge the safety assertions at each instruction, and its information records undischarged safety assertions. These undischarged safety assertions are added as assumptions to predecessor theorems using the Frame rule in Hoare logic in the AugmentTheorems function. SafetyAssertionAnalysis also checks that undischarged assertions are not propagated in a cycle, otherwise the analysis diverges with new undischarged safety assertions continually recorded. Hence the analysis is terminated and fails.

```
1: function SafeFunctionAnalysis($instr\_thms$)
2:     $(cfg, func) \leftarrow$ ComputeCFGandCallees($instr\_thms$)
3:     $safe\_thms \leftarrow$ AddSafetyAssertions($instr\_thms$)        ▷ Add the $ms$,$cfi_1$,$cfi_2$
   safety assertions to each instruction's Hoare triple.
4:     $bb\_safe \leftarrow$ SafeCompose($safe\_thms$)        ▷ Use MEM_CFI_SAFE_COMPOSE rule.
5:     $func\_safe \leftarrow \forall callee \in func\cdot$ SafeFunctionAnalysis($callee$)
6:     $assertion\_info \leftarrow$ SafetyAssertionAnalysis($bb\_safe, func\_safe, cfg$)
7:     $(bb\_safe', func\_safe') \leftarrow$ AugmentTheorems($bb\_safe, func\_safe, assertion\_info$)
8:     return FUN_SAFE_RULE($bb\_safe', func\_safe'$)
9: end function
10: $instr\_thms \leftarrow \forall instr\cdot$ CambridgeARM_GetInstructionModel($instr$)
11: SafeFunctionAnalysis($instr\_thms$)
```

Fig. 3. Safe Function analysis in AUSPICE [24]. Uses single-instruction theorems from the Cambridge ARM model, and returns `FUN_SAFE` theorem for function. *bb_safe* and *func_safe* contain basic block and callee function safety theorems respectively.

4 Safety Proofs for Machine Code with System Calls

There are two main steps to support safety proofs of machine code with syscalls.

First, we model the supervisor call instruction (`svc`), whose effects occur in both user-mode, and in supervisor-mode where the OS services the syscall. As we focus on the safety of user-mode programs, we do not wish to fully model the actions of the OS. Instead, we assume that the processor correctly handles the mode-switch from user to supervisor mode, and that the OS correctly services the syscall (§2). We focus on only the user-mode-observed effects after the syscall has been serviced by the OS. We model syscalls in user-mode in an *axiomatic* manner: we represent the user-mode-observed effects of syscalls as "axiomatized" (rather than proven) Hoare triples, that we introduce as hypotheses in our model.

Second, we need to augment our syscall models to support safety proof automation. AUSPICE's proof automation needs concrete safety assertions for each instruction. For typical instructions in user-mode programs, the proven Hoare triple for each instruction contains enough information for computing concrete

```
 1: function SafetyAssertionAnalysis($bb\_safe\_thms$, $func\_safe\_thms$, $cfg$)
 2:     $info \leftarrow \emptyset$                    ▷ Analysis information keyed by CFG node
 3:     procedure AssertionAnalysisStep($info$, $last\_info$, $cfg$)
 4:         for all $node \in cfg$, $pred \in$ FindPreds($cfg$, $node$) do
 5:             $pred\_preconds \leftarrow$ GetThmPreconds($pred$) $\bigcup$ $last\_info[pred]$
 6:             $node\_asserts \leftarrow$ GetThmAsserts($node$) $\bigcup$ $last\_info[node]$
 7:             for all $assert \in node\_asserts$ do
 8:                 if PROVE($pred\_preconds$, $assert$) == False then
 9:                     $info.term[pred] \leftarrow info.term[pred]$ $\bigcup$ $assert$
10:                     $a\_path \leftarrow$ FindAssertPath($last\_info.path[node]$, $assert$)
11:                     $info.path[pred] \leftarrow info.path[pred]$ $\bigcup$ $a\_path$
12:                     AbortIfAssertPathIsCycle($a\_path$)
13:     end procedure
14:     repeat
15:         $last\_info \leftarrow info$; $info \leftarrow$ AssertionAnalysisStep($info$, $last\_info$, $cfg$)
16:     until $last\_info == info$
17:     return $info$
18: end function
```

Fig. 4. Safety Assertion Analysis in AUSPICE [24]. FindPreds gives the CFG predecessors of a node; GetThmPreconds and GetThmAsserts are helper functions that return the pre-conditions and safety assertions in a given Hoare triple; Prove invokes the HOL4 `METIS` prover to try to discharge a safety assertion given a pre-condition; FindAssertPath computes the propagation path of an undischarged safety assertion.

safety assertions (Line 3 in Fig. 3). However, the effects of a syscall cannot be determined from the `svc` instruction alone, and depends on the arguments passed to it. These arguments are set up in the instructions leading to the `svc` instruction, and in the callers of the syscall. In AUSPICE-R, we use a *delayed* approach to analyze syscalls: We express the effects of syscalls symbolically, and we concretize these symbolic variables later in the analysis when information is available from callers of the syscall.

4.1 Modeling of System Calls in User-mode Programs

Rationale Behind Model. First, we focus on the user-mode-visible effects of syscalls that may affect our safety properties. Our safety properties are affected by memory addresses that are written to, and by the value of the program counter. As the processor will restore the program counter to the address of the instruction immediately following the `svc` instruction (B1.8.10 in [5]), we need to focus on only the addresses in the user process's memory that are written to during the servicing of the syscall. All other processor state (user-mode registers, apart from `r0` which stores a return value, and status flag values) remains unchanged, as user-mode registers are distinct from supervisor-mode registers, and the processor restores the values of the original status flags (B1.8.10 in [5]).

Second, we need to know the user-mode visible effects of each syscall in user-mode. We need to: (i) retrieve the number of the syscall invoked (passed in register `r7`, based on the Linux Application Binary Interface (ABI) for ARM [1]), (ii) identify the syscall invoked (e.g., from the Linux kernel's documentation and/or source-code), and (iii) retrieve the arguments passed to the syscall (via user-mode registers or the user-mode stack). This allows us to identify the behavior of each invoked syscall from its specification. We can then instantiate our safety-assertions from the user-mode-observed effects of each syscall invocation.

Axiomatization of System Call Effects. We "axiomatize" the Hoare triples for syscalls by constructing an unproven Hoare triple for each syscall, which we then introduce as an assumption. These unproven Hoare triples are collected as hypotheses of the final safety proof, and they formalize our assumption of each syscall's effects on user-mode state, based on the syscall's specification.

Fig. 5 shows an example axiom for the `write` syscall. 3 kinds of assertion relations are shown: (i) `aR` asserts the value of the specified register; (ii) `aPC` asserts the value of the program counter; (iii) `aMEMORY` asserts the domain (df) and contents of memory (map f from addresses to stored values). The pre-state value of register `r7` is asserted to be the literal 4, which is the syscall number for `write`, while the other pre-state register values are asserted to be symbolic variables ($r0$, $r1$, $r2$, $r14$), as they are unknown when we analyze the `svc` instruction alone. We instantiate these symbolic variables with concrete values later when analyzing the instructions leading up to the syscall invocation (details in §4.2). While some of the asserted resources (e.g., register `r1`) remain unchanged and could be omitted from the Hoare triple, we include this information as it may be required in our analysis for modeling the full behavior of the syscall.

Note that the Hoare triple is repeated on the left-hand-side of the turnstile "$\vdash$", indicating that the Hoare triple is a hypothesis. The post-state of this axiom for `write` is identical to its pre-state (except for the value of register `r0`, given by the `aR 0w` assertion), as `write` does not modify any user-mode-visible processor state. The value of register `r0` in the post-state is given by the symbolic variable rv, which indicates the return value from the syscall, and can represent the return value of both failed and successful syscalls. This axiom is representative of the other syscalls AUSPICE-R supports for which there are no effects that are directly visible in user-mode: `open`, `close`, `mmap`, `munmap`, `nanosleep`.

In contrast, consider our constructed axiom for the `read` syscall in Fig. 6. `read` has user-mode-visible effects: the bytes that it reads are written to and visible in the process's memory at the supplied address. The condition "`cond`($addrs \subseteq df$)" asserts that the set of addresses $addrs$ supplied to the syscall are in the domain of the memory map f. Also, the process's memory is updated from map f to (g f), where g represents the effects of `read` on memory. Note that $addrs$ and g are both symbolic. Note also that the value in register `r0` (asserted by `aR 0w`) in the post-state of the axiom is symbolic, and can represent the return values from both successful and failed invocations of the syscall. While the OS may not have written to all the addresses in the set $addrs$ when `read` fails or reads fewer than the requested number of bytes, $addrs$ conservatively lists the maximum extent of the memory written to by `read`.

$$\begin{array}{rl}
\texttt{SPEC ARM\_MODEL}\ (\texttt{aR 0w}\ r0 * \texttt{aR 1w}\ r1\ * \vdash & \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 0w}\ r0\ * \texttt{aR 1w}\ r1\ * \\
\texttt{aR 2w}\ r2\ * \texttt{aR 7w 4w}\ * \texttt{aPC}\ p\ * & \texttt{aR 2w}\ r2\ * \texttt{aR 7w 4w}\ * \texttt{aPC}\ p\ * \\
\texttt{aR 14w}\ r14 * \texttt{aMEMORY}\ df\ f) & \texttt{aR 14w}\ r14 * \texttt{aMEMORY}\ df\ f) \\
\{(p, \texttt{0xEF000000})\}\ (\texttt{aR 0w}\ rv * \texttt{aR 1w}\ r1\ * & \{(p, \texttt{0xEF000000})\}\ (\texttt{aR 0w}\ rv * \texttt{aR 1w}\ r1\ * \\
\texttt{aR 2w}\ r2 * \texttt{aR 7w 4w} * \texttt{aR 14w}\ r14\ * & \texttt{aR 2w}\ r2 * \texttt{aR 7w 4w} * \texttt{aR 14w}\ r14\ * \\
\texttt{aPC}\ (p + \texttt{4w}) * \texttt{aMEMORY}\ df\ f) & \texttt{aPC}\ (p + \texttt{4w}) * \texttt{aMEMORY}\ df\ f)
\end{array}$$

Fig. 5. Constructed Hoare triple axiom for the `write` syscall. `4w` is a numerical constant 4, where the suffix `w` indicates 4 is a fixed-width word.

$$\begin{array}{rl}
\texttt{SPEC ARM\_MODEL}\ (\texttt{aR 0w}\ r0 * \texttt{aR 1w}\ r1\ * \vdash & \texttt{SPEC ARM\_MODEL}\ (\texttt{aR 0w}\ r0 * \texttt{aR 1w}\ r1\ * \\
\texttt{aR 2w}\ r2 * \texttt{aR 7w 3w} * \texttt{aR 14w}\ r14\ * & \texttt{aR 2w}\ r2 * \texttt{aR 7w 3w} * \texttt{aR 14w}\ r14\ * \\
\texttt{aPC}\ p * \texttt{cond}(addrs \subseteq df) * \texttt{aMEMORY}\ df\ f) & \texttt{aPC}\ p * \texttt{cond}(addrs \subseteq df) * \texttt{aMEMORY}\ df\ f) \\
\{(p, \texttt{0xEF000000})\}\ (\texttt{aR 0w}\ rv * \texttt{aR 1w}\ r1 * & \{(p, \texttt{0xEF000000})\}\ (\texttt{aR 0w}\ rv * \texttt{aR 1w}\ r1\ * \\
\texttt{aR 2w}\ r2 * \texttt{aR 7w 3w}\ * \texttt{aR 14w}\ r14\ * & \texttt{aR 2w}\ r2 * \texttt{aR 7w 3w}\ * \texttt{aR 14w}\ r14\ * \\
\texttt{aPC}\ (p + \texttt{4w}) * \texttt{aMEMORY}\ df\ (g\ f)) & \texttt{aPC}\ (p + \texttt{4w}) * \texttt{aMEMORY}\ df\ (g\ f))
\end{array}$$

Fig. 6. Constructed Hoare triple axiom for the `read` syscall.

Implementation. The construction of unproven Hoare triples for each syscall (Fig. 7) is implemented as a wrapper around the model construction for individual instructions in the Cambridge ARM model, and replaces Line 10 in Fig. 3. When a `svc` instruction (`0xEF000000`) is detected, AUSPICE-R constructs an unproven Hoare triple based on the name of the function that the instruction is in. ConstructSyscallTriple implements the unproven Hoare triple construction process described above. We initially support modeling the following syscalls for simple I/O operations: `read`, `write`, `open`, `close`, `mmap`, `munmap`, `nanosleep`.

4.2 Supporting Safety Proof Automation for System Calls

Next, to support automated safety proofs in AUSPICE-R for syscalls, we need to concretize the initially-symbolic effects in the unproven Hoare triples for each syscall, as the safety assertion discharge in SafetyAssertionAnalysis (Fig. 4) reasons about memory addresses individually. To concretize the symbolic effects of a syscall's unproven triple, AUSPICE-R examines the arguments the syscall is invoked with when running SafeFunctionAnalysis (Fig. 3) on the caller of the syscall. We first illustrate how the arguments to system calls are interpreted,

```
1: function AUSPICE_R_GetInstructionModel($addr$, $addr\_to\_func$, $instr$)
2:     if !($instr$ = 0xEF000000) then
3:         return CambridgeARM_GetInstructionModel($instr$)
4:     else
5:         $func\_containing\_instr \leftarrow addr\_to\_func[addr]$
6:         return ConstructSyscallTriple($instr$, $addr$, $func\_containing\_instr$)
7: end function
```

Fig. 7. Algorithm for unproven Hoare triple construction for syscalls.

using the `read` syscall. Then we discuss how the symbolic effects are concretized, before we describe how these are implemented in AUSPICE-R's analysis.

System Call Arguments. The Linux Programmer's Manual [2] states that the `read` syscall takes 3 arguments: (i) an integer indicating the file descriptor, (ii) a pointer at which to store bytes that have been read, and (iii) the number of bytes to read. Fig. 8 shows a fragment of machine code, where the basic block at address `0x80E4` calls the function `c_read`, which is the assembly-code wrapper that invokes the `read` syscall (at address `0x822C`). Fig. 9 shows the C prototype of the assembly-code wrapper. For each invocation of the `read` syscall, the values of the arguments to the syscall are loaded to the relevant registers (`r0`, `r1`, `r2`) at the call-site to its wrapper (i.e., at the basic block at `0x80E4`). AUSPICE-R extracts these values from the post-state assertions of the Safe Basic Block theorem for the call-site. Concretely, for this example, the values to the arguments are fd = 0, buf = `0x10250`, $count$ = 3. Note that the arguments may still be symbolic at this point (e.g., if reading a variable-length number of bytes). However, for AUSPICE to prove our safety properties for the `read` syscall, the pointer to store read bytes and the number of bytes to read must be concrete. This enables AUSPICE-R to update the symbolic safety assertions in the `FUN_SAFE` theorem of `read`'s syscall wrapper with concrete expressions, thus enabling the safety assertions to be discharged. If the pointer and number of bytes read remain symbolic, SafetyAssertionAnalysis cannot reason about the symbolic safety assertions, and the safety proof will fail.

```
80e4: e3a00000  mov r0, #0
80e8: e59f1098  ldr r1, [pc, #152]
80ec: e3a02003  mov r2, #3
80f0: eb000049  bl 821c <c_read>
...   ...
8188: 00010250
...   ...
0000821c <c_read>:
821c: e92d4880  push {r7, fp, lr}
8220: e28db004  add fp, sp, #4
8224: e24dd000  sub sp, sp, #0
8228: e3a07003  mov r7, #3
822c: ef000000  svc 0x00000000
...   ...
```

Fig. 8. Example ARM machine code invoking the `c_read` wrapper to the `read` syscall.

```
ssize_t read(int fd, void *buf,
             size_t count);
```

Fig. 9. Prototype of C function wrapper to `read` syscall.

Updating of Symbolic Effects. Next, we construct variable substitutions for the initial symbolic effects (written-address set $addrs$ and memory-update function g), which we apply to the unproven Hoare triple for the `read` syscall.

These substitutions concretize the effects of the syscall on user-mode processor state, so that SafetyAssertionAnalysis can reason about the safety of these effects. To complete its automated safety-property proofs, AUSPICE needs to enumerate the memory address of each byte written to. While AUSPICE can reason about byte-addresses containing symbolic variables (e.g., when the address written to is a symbolic variable *r3*), it cannot reason about symbolic ranges of addresses where the number of elements in the set is symbolic (even if the elements of the set are drawn from a finite universe, e.g., fixed-width words). This is due to limitations with HOL4's built-in tactics for reasoning about sets (`pred_setLib`). Hence, AUSPICE-R enumerates the byte-addresses written to by the syscall.

For the example in Fig. 8, 3 bytes are written at the address `0x10250`. Hence, we substitute *addrs* with $\{\texttt{0x10250w}; \texttt{0x10251w}; \texttt{0x10252w}\}$, and the update function g with the expression shown in Fig. 10. `extmem__c_read__0x80E4` is an opaque function that represents the results of external I/O, and it returns the (symbolic) data read given the byte-number read; "`=+`" is the map update operator, where "$a = +b$" indicates the value b is stored at address a.

$$\begin{aligned}\lambda f\ .\ & ((\texttt{0x10250w} = + (\texttt{extmem\_\_c\_read\_\_0x80E4 0w})) \\ & \quad ((\texttt{0x10251w} = + (\texttt{extmem\_\_c\_read\_\_0x80E4 1w})) \\ & \quad\ ((\texttt{0x10252w} = + (\texttt{extmem\_\_c\_read\_\_0x80E4 2w}))\ f)))\end{aligned}$$

Fig. 10. Concretized memory-update expression for the `read` syscall in Figure 8.

After substituting the symbolic effects for concrete values in each syscall's Hoare triple axioms, AUSPICE can automatically discharge the safety assertions for these axioms (if the machine code contains the necessary safety-checks).

Implementation. Fig. 11 describes the updated Safe Function analysis algorithm in AUSPICE-R, incorporating the unproven Hoare triple axiomatization (Line 12), and the concretization of symbolic effects (Line 7). In functions that call syscalls, SafeFunctionAnalysisWithSyscalls is first called on each syscall callee (Line 5). Then, the arguments to the syscall are available in the caller of the syscall, and the `FUN_SAFE` theorems of syscalls are concretized using information from the caller's basic blocks, *bb_safe* (Line 7). This concretization must take place before SafetyAssertionAnalysis (Line 8). AUSPICE-R adds 1300 lines of ML proof scripts to AUSPICE's code-base of 11.8 KLOC of ML.

5 Optimizing Safety Proof Automation

AUSPICE-R optimizes SafeFunctionAnalysis (Fig. 3) and SafetyAssertionAnalysis (Fig. 4) to speed up its safety-proof generation, so that larger programs can be verified in less time. AUSPICE-R leverages (i) common patterns in

```
1:  function SafeFunctionAnalysisWithSyscalls($instr\_thms$)
2:      $(cfg, func) \leftarrow$ ComputeCFGandCallees($instr\_thms$)
3:      $safe\_thms \leftarrow$ AddSafetyAssertions($instr\_thms$)
4:      $bb\_safe \leftarrow$ SafeCompose($safe\_thms$)          ▷ Use MEMCFISAFE_COMPOSE rule.
5:      $func\_safe \leftarrow \forall callee \in func \cdot$ SafeFunctionAnalysisWithSyscalls($callee$)
6:      $syscall\_callees \leftarrow \forall callee \in func \mid$ is_syscall($callee$)
7:      $func\_safe' \leftarrow \forall c \in syscall\_callees \cdot$ ConcretizeArgs($func\_safe[c], bb\_safe$)
8:      $assertion\_info \leftarrow$ SafetyAssertionAnalysis($bb\_safe, func\_safe', cfg$)
9:      $(bb\_safe', func\_safe'') \leftarrow$ AugmentTheorems($bb\_safe, func\_safe', assertion\_info$)
10:     return FUN_SAFE_RULE($bb\_safe', func\_safe''$)
11: end function
12: $instr\_thms \leftarrow \forall instr \cdot$ AUSPICE_R_GetInstructionModel($instr$)
13: SafeFuncAnalysisWithSyscalls($instr\_thms$)
```

Fig. 11. Updated Safe Function analysis in AUSPICE-R with support for safety proofs for machine code with syscalls. Added or changed steps are highlighted in blue.

gcc-compiled machine code for local-variable-writes to speed up SafetyAssertionAnalysis, and (ii) the behavior of safety assertions in callee functions in its inter-procedural analysis to speed up SafeFunctionAnalysis.

Common Compiler Conventions. SafetyAssertionAnalysis performs two tasks: (i) it finds pairs of pre-conditions $p \in P$ and safety assertions $a \in A$, such that $p \Rightarrow a$, and (ii) for assertions $a \in A$ for which no p is found, it propagates a to predecessor nodes, and checks if a's propagation path has a cycle. However, computing the propagation path of assertion a is expensive, as it requires symbolic execution along the propagation path.

We leverage two observations in gcc-compiled code: (i) there are two classes of memory-writes: to local variables (i.e., a constant offset from the frame pointer r11 or stack pointer r13), and to arbitrarily-computed addresses (typically stored in registers); (ii) r11 and r13 are generally updated only at the start and end of each function. Thus, safety assertions for writes to local variables will not change during the analysis of function bodies. To speed up SafetyAssertionAnalysis for writes to local variables, AUSPICE-R: (i) reduces the number of assertion terms analyzed, and (ii) skips the propagation-cycle check.

Fig. 12 describes the optimized version of the inner analysis step in SafetyAssertionAnalysis, which replaces AssertionAnalysisStep in Fig. 4. First, we represent the safety assertions for local-variable writes using range predicates: e.g., for a safety assertion "$\{r13 - 21\mathtt{w};\ r13 - 22\mathtt{w};\ r13 - 23\mathtt{w};\ r13 - 24\mathtt{w}\} \subseteq \{addr \mid addr < r11\}$", the addresses that are offset from r13 are where a local variable is stored on the stack; we replace this safety assertion with the range predicate "$24\mathtt{w} \leq r13 < r11 + 24\mathtt{w}$", which implies the original safety assertion. Thus, for writes to N different local variables in a function, only 2 rather than $2N$ predicates are propagated: one each for Safety Properties 1 and 2 (§3.1). We also define a narrowing operator for the meet of two range predicates which returns the more restrictive of two predicates to merge terms from multiple CFG paths. Second,

```
1: procedure AssertionAnalysisStepOpt($info$, $last\_info$, $cfg$)
2:   for all $node \in cfg$, $pred \in$ FindPreds($cfg, node$) do
3:     $pred\_preconds \leftarrow$ GetThmPreconds($pred$) $\bigcup$ $last\_info[pred]$
4:     $node\_asserts \leftarrow$ GetThmAsserts($node$) $\bigcup$ $last\_info[node]$
5:     $(range\_pds, other\_pds) \leftarrow$ partition is_range $pred\_preconds$
6:     $(localvar\_asserts, other\_asserts) \leftarrow$ partition is_localvar $node\_asserts$
7:     for all $assert \in localvar\_asserts$ do
8:       $curr\_range \leftarrow$ narrow(compute_range_predicate($assert$), $range\_pds$)
9:       $(prev\_range, other\_terms) \leftarrow$ partition is_range $(info.term[pred])$
10:      $info.term[pred] \leftarrow other\_terms$ $\bigcup$ narrow($curr\_range, prev\_range$)
11:    for all $assert \in other\_asserts$ do
12:      if PROVE($other\_pds, assert$) == False then
13:        $info.term[pred] \leftarrow info.term[pred]$ $\bigcup$ $assert$
14:        $a\_path \leftarrow$ FindAssertPath($last\_info.path[node], assert$)
15:        $info.path[pred] \leftarrow info.path[pred]$ $\bigcup$ $a\_path$
16:        AbortIfAssertPathIsCycle($a\_path$)
17: end procedure
```

Fig. 12. Optimized analysis step for SafetyAssertionAnalysis in AUSPICE-R. Updated steps are highlighted in blue. is_range and is_localvar return true for predicates that are ranges and that are about local-variable writes respectively.

since writes to local variables are to fixed offsets from the frame pointer (`r11`) or stack pointer (`r13`), which do not change in the function's body, we do not need to compute nor check for cycles in propagation paths.

Context-Sensitivity of Analysis. SafeFunctionAnalysis (Fig. 3) is an inter-procedural analysis which constructs a distinct Safe Function (`FUN_SAFE`) theorem for every call to each callee function. We use the program in Fig. 13 to illustrate AUSPICE-R's approach. First, consider the behavior of SafeFunctionAnalysis in AUSPICE: in `foo()`, `bar()` is called twice, thus one `FUN_SAFE` theorem is constructed for each of its two call-sites. We call this analysis "call-site context-sensitive", or *CSCS*. CSCS provides the highest level of precision. We would like to reduce the precision of our analysis to reduce the number of iterations of SafeFunctionAnalysis (Fig. 3) needed to successfully generate a safety proof.

```
bar() { ... }
baz() { bar();
        ... }
foo() { bar();
        baz();
        bar();
        ... }
```

Fig. 13. Example program for inter-procedural analysis.

Context-insensitive inter-procedural analysis provides the lowest level of precision: we analyze each function once for the whole program and generate one `FUN_SAFE` theorem for it. However, in our example, having only one `FUN_SAFE` theorem for each function results in imprecise analysis by forcing safety assertions from instructions at different call-tree depths (e.g., `foo()` vs. `baz()`) to be framed onto the same theorem (`bar()`). (We refer to the function-level CFG as a call-tree, whose depth is the number of nested function calls.) This is logically equivalent to different instances of the function's stack overlapping in memory at the same time, although during execution, only one instance of the function's stack exists in memory at any point in time, resulting in imprecise analysis. Hence, the proof generation fails when there are safety assertions from a smaller call-tree depth (e.g., `foo()`) than the call-tree depth of the currently-analyzed function (e.g., `baz()`). Having one `FUN_SAFE` theorem per-function per-call-tree-depth is also insufficient, as two caller functions at the same call-tree depth could have different stack sizes, resulting in the same contradiction as above.

On the other hand, in each function, we need to analyze each callee function only once, regardless of how many times that callee function is called. We call this analysis "single-function context-sensitive" (*SFCS*). When analyzing a function F, we need only one `FUN_SAFE` theorem for each callee function C, regardless of how many times C is called in F. Then, we can frame the safety assertions from all the return-sites of C in the function F to the single theorem for C, as there would not be any contradiction in the analysis. In our example, we can merge all the safety assertions required at all the return-sites from `bar()` in `foo()`, and add them to the `FUN_SAFE` theorem for `bar()`. While there is some loss of precision (e.g., the `FUN_SAFE` theorem for the second call to `bar()` does not need to consider the safety assertions that need to be discharged when calling `baz()`), we now need to run SafeFunctionAnalysis fewer times.

Limitations. AUSPICE-R continues to analyze syscall wrapper functions using CSCS analysis, as AUSPICE-R needs to generate a unique `FUN_SAFE` theorem to correctly consider the arguments passed to a syscall at each distinct call-site.

6 Evaluation

First, we evaluate the ability of AUSPICE-R (with §5 optimizations) to automatically prove safety properties in ARM machine code with syscalls. We picked 2 classes of programs: (i) simple versions of file I/O utilities that we implemented for Linux on the ARM platform; (ii) programs with hardware inputs/outputs on the Raspberry Pi single-board-computer. All our test programs are Linux user-mode programs on ARM, and compiled with an unmodified `gcc` toolchain for ARMv6 with `-O0` optimization. All proofs were generated using the HOL4 proof assistant [22] on an Intel Core i7 2.6 GHz with 16 GB RAM. Our test programs are available at `http://users.ece.cmu.edu/~jiaqit/aplas16data` .

6.1 File-based I/O

We implemented simple versions of three common file I/O utilities in C for Linux on the ARM platform. These programs contained the `read`, `write`, `open`, and `close` syscalls. Table 1 summarizes our test programs, their sizes and functionality, and the times taken to automatically prove the safety of each program. Our results show that AUSPICE-R can prove safety automatically in realistic programs with useful I/O functionality, and AUSPICE-R took less than 2 hours to automatically prove the safety of each program.

Program	Lines of C	Instructions	Proof Time	Description
`cat`	411	207	26.9 mins	Outputs contents of a file.
`wc`	427	641	95.1 mins	Counts number of words in a file.
`grep`	428	621	40.2 mins	Prints lines containing given string.

Table 1. Descriptions and proof times for file-based I/O utilities.

Program	Lines of C	Instruc-tions	Call-Tree Depth	Proof Time	Description
`blink`	418	619	3	58.7 mins	Turn LED repeatedly on/off.
`light`	429	854	4	81.7 mins	Use light-sensor to light LED when dark.
`lcd`	559	2229	6	22.2 hours	Print string to 16x2 monochrome LCD.
`fall-det`	923	3331	6	47.9 hours	Detect human falls with accelerometer using algorithm in [13].

Table 2. Proof times for hardware I/O programs.

6.2 Embedded Software

We implemented 4 programs containing hardware inputs and outputs on the Raspberry Pi. These programs contained the `mmap`, `munmap`, `open`, `close`, and `nanosleep` syscalls. Table 2 describes and summarizes our programs, their sizes, and the times taken to prove safety automatically. The proof times for the `blink` and `light` test-programs are under 2 hours, and comparable to the proof times for our file I/O examples above. The proof times for the `lcd` and `fall-det` test-programs are significantly longer, as they are significantly larger, and have much deeper call-trees: the run-time of AUSPICE-R's inter-procedural analysis is exponential in the depth of the call-tree. `lcd` and `fall-det`, with 2229 and 3331 instructions respectively are, to the best of our knowledge, the largest programs for which safety properties have been automatically proved using an approach that considers the full semantics of instructions (vs. 1104 instructions using ARMor [27], which also uses the Cambridge ARM model [18]).

6.3 Proof Optimization

We report the proof times for programs that have been evaluated on prior techniques, to evaluate AUSPICE-R's optimizations. Table 3 summarizes our results for our 3 test programs (without syscalls): `memcpy`, which copies an array of integers; `sort`, which implements Insertion Sort; and `string-search`, from the MiBench benchmark suite [12], which implements the Boyer-Moore string-search algorithm. We compared AUSPICE-R's proof times to our prior work, AUSPICE [24], for all 3 programs: AUSPICE-R's safety proofs were between 252% to 1297% faster. Also, AUSPICE-R's proof time for `string-search` was 1067% faster than ARMor [27] (which used an Intel Core i7 2.7 GHz). AUSPICE-R's proof optimizations significantly improved the times taken for automated safety proofs.

Program	Instructions	AUSPICE-R Proof Time	vs. AUSPICE [24]		vs. ARMor [27]	
			Proof Time	AUSPICE-R X% faster	Proof Time	AUSPICE-R X% faster
`memcpy`	116	6.5 mins	16.4 mins	252%	-	-
`sort`	337	9.4 mins	122 mins	1297%	-	-
`string-search`	530	0.76 hours	6.05 hours	796%	8 hours	1067%

Table 3. Comparing AUSPICE-R's proof times with AUSPICE [24] and ARMor [27].

		Iterations of Analysis		
Program	Instructions	CSCS (AUSPICE)	SFCS (AUSPICE-R)	Optimization
`lcd`	2229	2799	751	73%
`fall-det`	3331	5069	845	83%

Table 4. Comparison of number of iterations of analysis of Call-site Context-Sensitivity (CSCS) vs. Single-Function Context-Sensitivity (SFCS) (§5).

To show the optimization gains from AUSPICE-R's SFCS inter-procedural analysis (§5), we compared the number of iterations of SafeFunctionAnalysis (Fig. 3) in SFCS to that in CSCS analysis. The optimization gains are greatest in programs with repeated calls to non-syscall-wrapper functions. We simulated the number of iterations the inter-procedural analysis needs to run for `lcd` and `fall-det` by analyzing their function-level call-trees. Table 4 summarizes the results of using SFCS over CSCS. The number of iterations of the inter-procedural analysis for constructing a safety proof reduced by 73% for `lcd`, and by 83% for `fall-det`, showing that SFCS made our analysis feasible for large test-programs.

7 Discussion

First, our axioms of syscall behavior provide a formal, succinct expression of our expectations of the behavior of syscalls, as observed in user-mode. We envision that these axioms can be used to empirically validate the behavior of syscalls in future, e.g., through dynamic testing.

Second, we found our requirement of the `read()` syscall to accept only concrete lengths and buffer addresses to not be a significant limitation. To support `read()`s to buffers of variable lengths and symbolic addresses, we implemented a wrapper function that reads to a fixed length/address buffer, and copies its contents to the final destination buffer.

8 Related Work

ARMor [27], and our prior work AUSPICE [24], automatically prove safety properties for ARM machine code using HOL4 [22], and are closest to AUSPICE-R. ARMor supports only "bare-metal" programs running without an OS, while AUSPICE supports user-mode machine code but not syscalls. Goel at al. [21] reason about x86 machine code with syscalls in the ACL2 logic for "heavy-weight", manual proofs of functional-correctness, while AUSPICE-R automates proofs of "light-weight" safety properties. Goel's formalization of x86 syscalls tracks OS-state that may not be user-mode-visible that is needed for functional correctness proofs, while AUSPICE-R focuses only on user-mode-visible state that impacts our safety properties.

Sequoll [9] performs model-checking on ARM machine code using the Cambridge ARM model [18]. Bedrock [10] "mostly-automates" functional correctness proofs for an idealized machine-language, whereas, AUSPICE-R proves safety properties for machine code emitted by standard compilers. CompCert [16] is a compiler that has been formally verified to preserve the semantics of well-behaved C programs during compilation. VST [8] is a program logic for reasoning about programs in Cminor (a CompCert intermediate language), whose claims hold in its compiled machine code due to its use of CompCert. Verasco [14] is a formally verified static-analyzer for Cminor whose guarantees carry over to its compiled code due to CompCert. It checks for the absence of run-time errors that can cause safety violations such as the ones AUSPICE-R proves the absence of, but does not produce proofs for individual programs. SeaHorn [7] and Dafny [15] allow users to specify source-code assertions for checking arbitrary properties, while AUSPICE-R focuses on specific safety properties for CFI [6], without needing user specifications.

9 Conclusion and Future Work

We have presented AUSPICE-R, an extension to AUSPICE [24] that: (i) automates safety-property proofs in ARM machine code containing system calls by axiomatizing their user-mode-visible effects, and (ii) optimizes automated safety-property proofs by leveraging common conventions in compiled code and by providing a more efficient inter-procedural analysis. We have demonstrated AUSPICE-R on simple file I/O utilities implemented for Linux on ARM, and on programs containing hardware I/O on the Raspberry Pi single-board-computer. We showed that AUSPICE-R is up to 12x faster and supports programs up to 3x larger than prior work.

In future, we plan to tackle the challenges associated with more complex syscalls, e.g., for network communications, that may have more complex user-mode effects. We also plan to investigate how our axioms of syscall behavior can be used to aid dynamic testing of syscall-servicing behavior by OS kernels.

References

1. Application Binary Interface for the ARM Architecture, `http://bit.ly/22OaMai`
2. Linux Programmer's Manual: Syscalls, `http://bit.ly/1VChJMY`
3. The ARM-THUMB Procedure Call Standard (2000), `http://bit.ly/1NbOQhT`
4. As Gadgets Shrink, ARM Still Reigns As Processor King (Sep 2013), `http://onforb.es/19LIzgd`
5. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2014)
6. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow Integrity. In: ACM CCS (2005)
7. A.Gurfinkel, T.Kahsai, A. Komuravelli, J.A.Navas: The SeaHorn Verification Framework. In: CAV (2015)
8. Appel, A.: Verified Software Toolchain. In: ESOP (2011)
9. Blackham, B., Heiser, G.: Sequel: A Framework for Model Checking Binaries. In: IEEE RTAS (2013)
10. Chlipala, A.: Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In: PLDI (2011)
11. G. Klein et al.: seL4: Formal verification of an OS kernel. In: SOSP (Oct 2009)
12. Guthaus, M. et al.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: IEEE WWC Workshop (2001)
13. Jia, N.: Detecting Human Falls with a 3-Axis Digital Accelerometer (2009), `http://bit.ly/23fXhFE`
14. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL (2015)
15. K. Rustan M. Leino: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR (2010)
16. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7) (2009)
17. Miller, C., Valasek, C.: Remote Exploitation of an Unaltered Passenger Vehicle, `http://bit.ly/1Xk71rn`
18. Myreen, M., Fox, A., Gordon, M.: Hoare Logic for ARM Machine Code. In: FSEN (2007)
19. Myreen, M., Gordon, M.: Hoare Logic for Realistically Modeled Machine Code. In: TACAS (2007)
20. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE LICS (2002)
21. S. Goel et al.: Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls. In: FMCAD (2014)
22. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: TPHOLs (2008)
23. Tan, J., Tay, H., Drolia, U., Gandhi, R., Narasimhan, P.: PCFIRE: Towards Provable Preventative Control-Flow Integrity Enforcement for Realistic Embedded Software. In: EMSOFT (2016)
24. Tan, J., Tay, H., Gandhi, R., Narasimhan, P.: AUSPICE: Automatic Safety Property Verification for Unmodified Executables. In: VSTTE (2015)
25. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient Software-Based Fault Isolation. In: SOSP (1993)
26. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and Understanding Bugs in C Compilers. In: PLDI (2011)
27. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: ARMor: Fully Verified Software Fault Isolation. In: EMSOFT (2011)