PCFIRE: Towards Provable Preventative Control-Flow Integrity Enforcement for Realistic Embedded Software

Jiaqi Tan
Carnegie Mellon University
Dept. of ECE,
Pittsburgh, PA, USA
tanjiaqi@cmu.edu

Hui Jun Tay
Carnegie Mellon University
Dept. of ECE,
Pittsburgh, PA, USA
htay@andrew.cmu.edu

Utsav Drolia
Carnegie Mellon University
Dept. of ECE,
Pittsburgh, PA, USA
udrolia@andrew.cmu.edu

Rajeev Gandhi
Carnegie Mellon University
Dept. of ECE,
Pittsburgh, PA, USA
rgandhi@ece.cmu.edu

Priya Narasimhan
Carnegie Mellon University
Dept. of ECE,
Pittsburgh, PA, USA
priya@cs.cmu.edu

ABSTRACT

Control-Flow Integrity (CFI) is an important safety property of software, particularly in embedded and safety-critical systems, where CFI violations have led to patient deaths and can render cars remotely controllable by attackers. Previous techniques for CFI may reduce the robustness of embedded and safety-critical systems, as they handle CFI violations by stopping programs. In this work, we present PCFIRE, a preventative approach to CFI that prevents the root-causes of CFI violations to allow recovery, and enables programmers to specify robust recovery actions by providing CFI via source-code safety-checks. PCFIRE's CFI can be formally proved automatically, and supports realistic features of embedded software such as hardware and I/O access. We showcase PCFIRE by providing, and automatically proving, CFI for: benchmark programs, text utilities containing I/O, and embedded programs with sensor inputs and hardware outputs on the Raspberry Pi single-board computer.

Keywords

Control-Flow Integrity; Program Logic; Proof Assistant; Interactive Theorem Proving; ARM Executables; HOL

1. INTRODUCTION

Control-Flow Integrity (CFI) [5] is an important safety property of software. Software whose CFI has been violated can behave in unexpected ways and allow users to hijack its execution via malicious inputs. CFI has even greater importance in embedded systems, whose software may be safety-critical (e.g., in cyber-physical systems), where failures may "result in loss of life, significant property damage, or damage to the environment" [16]. Informally speaking, the CFI of a program is preserved when its executed machine-code (i.e., its control-flow) behaves as specified by its source-code. CFI violations occur when a program's executed machine-code deviates from the behavior specified in its source-code (e.g., when a buffer is overflowed to overwrite indirect jump targets in memory, changing the program's execution). CFI violations in embedded software have resulted in fatal failures of medical devices, e.g., drug infusion-pumps [34], resulting in large-scale safety recalls [32], and have enabled attackers to control safety-critical software in cars [27].

There are a number of important features when ensuring the CFI of embedded, potentially safety-critical software. First, we need to ensure that the software is able to recover from CFI violations to continue operation. Current CFI techniques halt the execution of software when CFI violations are detected [5, 33, 38, 19, 6] to prevent attackers from hijacking compromised programs. This can be a problem for safety-critical software: e.g., a buffer-overflow in a Baxter Colleague 3 drug infusion-pump was detected and the pump was stopped, directly leading to a patient's death [34]. Second, to enable software to recover from CFI violations and continue operating, we need to ensure that the root-causes of CFI violations (e.g., overwriting of jump targets or program instructions in memory) are prevented in the first place. If CFI violations are detected only after their root-causes have occurred, the root-causes cannot be undone, and recovery is not possible. Third, CFI techniques for embedded software need to support features in realistic embedded software, such as system-calls in user-mode programs running in an OS. Previous techniques for embedded software [38] targeted "bare-metal" programs running directly on hardware without an OS. As embedded systems become more sophisticated, they often have an OS, and applications run as user-mode programs [20]. Fourth, we want to enable automated formal proofs of CFI, so that programmers can easily obtain high-assurance of the CFI of their programs without knowledge of formal methods.

In this paper, we present PCFIRE, a **preventative** approach to providing Control-Flow Integrity (CFI) for software **in a way that prevents the root-causes of CFI**, that: (i) enables application-specific recovery actions, (ii) supports important features in realistic embedded applica-

EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4485-2/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2968478.2968492

tions, and (iii) is amenable to automated formal CFI proofs, for embedded applications. First, we develop the PCFIRE-C tool, which prescribes source-code safety-checks for CFI for C programs, which programmers then apply to their source-code. While source-code safety-checks are not themselves novel, the novelty in our prescribed source-code safety-checks is that they are specially crafted such that the safety-checks compile to machine-code that can be proved to have CFI fully automatically, without any user inputs (e.g., code annotations). Second, we build on our existing logic framework, AUSPICE [31], for our CFI proofs, and we develop a novel extension, AUSPICE+, to support CFI proofs in machine-code with system-calls. Finally, we show how PCFIRE supports I/O in our case-studies. While our case-studies do not feature actual embedded software running in consumer devices, we demonstrate support for realistic I/O behavior, including hardware I/O, in small utilities. We develop and automatically prove the CFI of: (i) text-utilities, and (ii) embedded software with sensor inputs and hardware outputs on the Raspberry Pi single-board computer.

Our contributions are: (i) a novel set of heuristics for writing safety-checks using *check-and-branch* statements in C programs, such that CFI violations can be prevented, and the compiled ARM machine-code of the programs can be automatically proved to have CFI, (ii) a tool (PCFIRE-C) which realizes our heuristics by prescribing source-code CFI safety-checks for a C program, (iii) the AUSPICE+ extension of the AUSPICE logic framework, and (iv) case-studies showing how we can write embedded applications in C containing system-calls and hardware inputs/outputs (e.g., sensors and LCD), which can be automatically proved to have CFI using AUSPICE+. To the best of our knowledge, PCFIRE is the first framework for CFI that prevents CFI violations to enable programmers to customize recovery actions, and supports programs containing system-calls, while still enabling CFI to be formally proved automatically.

1.1 Control-Flow Integrity (CFI)

Control-Flow Integrity (CFI) is a safety property which states that the execution of software follows a path of a Control-Flow Graph (CFG) that is "determined ahead of time" [5]. While programmers reason about software behaviors in the programming language they use (e.g., C), software execution manifests as the actual (machine-code) instructions that are executed by the processor. Hence, there are two views of a program's execution: one at the source-code level (which captures the programmer's intentions), and one at the machine-code level (which represents the actual instructions executed). Then, the execution of software with CFI follows the CFG which captures the source-code-specified behavior of the software. CFI violations occur when the sequence of machine-code instructions executed is not present in the CFG of the software's source-code. While CFI is a general safety property, the CFI of a given program is specific to the target architecture of its machine-code. Thus, CFI techniques are also architecture-specific, although they can be adapted to different architectures.

Low-level programming languages, such as C, give programmers direct access to memory. This can give rise to CFI violations in the actual instructions executed. We present a simple example, which we will also use to illustrate our approach later. Consider this piece of C code:

```
void arraycopy (int *src, int *dst, int n) {
   int i;
   for (i = 0; i < n; i++) { dst[i] = src[i]; }
}
```

At first glance, `arraycopy` is designed to copy the array `src` to the array `dst`. From its C source, `arraycopy` superficially appears to not have any buffer-overflows (which can give rise to CFI violations), as the caller supplies `n`, limiting the number of array elements copied. However, the compiled machine-code of `arraycopy` exposes the low-level behavior of the function, where we can see there are potential CFI violations. Consider this fragment of ARM machine-code for the statements `dst[i] = src[i]`, `i++`, and `i < n`.

```
0x8094: e92d0810  push {fp, lr}
0x8098: e28db004  add fp, sp, #4
0x809c: e24dd018  sub sp, sp, #24
...
0x8150: e51b3008  ldr r3, [fp, #-8]
0x8154: e1a03103  lsl r3, r3, #2
0x8158: e51b2014  ldr r2, [fp, #-20]
0x815c: e0823003  add r3, r2, r3
0x8160: e51b2008  ldr r2, [fp, #-8]
0x8164: e1a02102  lsl r2, r2, #2
0x8168: e51b1010  ldr r1, [fp, #-16]
0x816c: e0812002  add r2, r1, r2
0x8170: e5922000  ldr r2, [r2]
0x8174: e5832000  str r2, [r3]
0x8178: e51b3008  ldr r3, [fp, #-8]
0x817c: e2833001  add r3, r3, #1
0x8180: e50b3008  str r3, [fp, #-8]
0x8184: e51b2018  ldr r2, [fp, #-24]
0x8188: e51b3008  ldr r3, [fp, #-8]
0x818c: e1520003  cmp r2, r3
...
```

CFI violations can occur when function return addresses saved to the stack are overwritten. At the machine-code level, an instruction must write to memory for this to occur. In this fragment of machine-code, there are two `str` instructions which write to memory, at addresses `0x8174` (`str r2, [r3]`), and `0x8180` (`str r3, [fp, #-8]`). Then, from the function prologue (addresses `0x8094`, `0x8098`), we can see that the link register, `lr`, which stores the return address of the function calling `arraycopy`, is saved to the stack, and the frame pointer, `fp`, is advanced past the address where the link register is saved to the stack. Hence, any writes to memory addresses smaller than `fp`, will not overwrite the saved `lr` value on `arraycopy`'s stack, for a descending stack. Thus, we know that the second memory write, `str r3, [fp, #-8]`, will not overwrite the saved `lr` value, and cause a CFI violation, since its target address $(\texttt{fp} - 8) < \texttt{fp}$ (for $\texttt{fp} > 8$). However, the first memory write, `str r2, [r3]`, is to a dynamically computed address. This instruction potentially overwrites the saved `lr` value, since it is hard to determine statically from the machine-code alone what the address `[r3]` is. In fact, `[r3]` is computed from arguments `dst` and `n`, and a buffer-overflow can occur, if (i) `dst` does not point to an array of integers, or if (ii) `n` is larger than the size of the array at `dst`.

2. PROBLEM STATEMENT

Goals. The main objective of PCFIRE is to enable CFI for software to be provided in a **preventative** way that is amenable to realistic embedded applications. Our goals are: (i) to enable CFI violations to be prevented, (ii) to do so using purely source-code mechanisms, so that developers can add application-specific recovery actions, (iii) to provide formal safety proofs of the CFI of machine-code, (iv) to provide these safety proofs automatically without user inputs (e.g., code annotations, loop invariants), (v) to work with unmodified, standard compilers, so that our approach is transparent to software development methodologies, and (vi) to automate safety proofs of CFI in programs with system-calls, so as to support realistic embedded applications.

Scope. PCFIRE targets C programs for providing source-code safety-checks, as C is a popular programming language for handling low-level I/O behavior on embedded platforms. PCFIRE's formal CFI safety proofs target machine-code programs. We target ARM, as it is the dominant processor architecture for mobile and Internet-of-Things embedded devices [4]. PCFIRE's CFI safety proofs are at the machine-code level, because CFI is a safety property about the machine-code of programs, and because machine-code safety proofs reduce the Trusted Computing Base (TCB) of our approach by letting us exclude the compiler from the TCB. While PCFIRE currently targets C source-code and ARM machine-code, we believe that our approach can be generalized to other architectures by adapting it to the Application Binary Interface (ABI) of other architectures. We target user-mode programs which run in an OS, and we consider Linux user-mode programs. This is realistic, as increasingly, embedded devices run full-featured OSes, and applications run as user-mode programs in an OS (e.g., as described in [20]).

Threat Model. PCFIRE's threat model consists of an attacker who is able to supply arbitrary (and potentially malicious) inputs to the target program. We assume that the physical security of the device running our target software is not compromised (i.e., no hardware nor OS-/firmware-replacement attacks). We further assume that the OS isolates user processes, and that the OS is neither malicious nor compromised (i.e., no direct changes to a process's memory).

Assumptions. In PCFIRE, we build on the AUSPICE logic framework [31], which builds on a trustworthy and detailed formalization of the ARM Instruction Set Architecture (ISA) from Cambridge University [21, 8] (the Cambridge ARM model). Hence, the CFI proofs in PCFIRE inherit the assumptions and limitations of the above frameworks. We assume PCFIRE's target programs:

1. Are unaffected by hardware exceptions, interrupts, and page table operations (not modeled),
2. Do not contain floating-point instructions (not modeled),
3. Do not contain recursive function calls (AUSPICE does not support them),
4. Do not contain `goto` or `longjmp` statements,
5. Do not contain arbitrary function pointers (PCFIRE's safety-checks can be extended to function pointer targets, but we leave this to future work),
6. Are single-threaded (only sequential behavior modeled),
7. Are statically compiled and linked, so that every instruction that can be executed is available to be verified,
8. Are compiled with commodity compilers, e.g., `gcc`, which obey the ARM-THUMB Procedure Call Standard (ATPCS) [3], with `gcc -O0` (or equivalent) optimization, and with debug information,
9. Can be disassembled with well-defined function boundaries using standard tools such as `GNU objdump`,
10. Have well-defined function prologues and epilogues.

We also assume that for programs which contain system-calls (syscalls), the underlying OS services the system-call correctly. Recent work has verified OS microkernels [10], making it possible for syscall servicing to be verified.

Non-goals. While PCFIRE enables developers to specify their own recovery actions from detected CFI violations, we do not prescribe recovery actions, as they are application-specific. PCFIRE focuses specifically on CFI, and is not concerned with other safety/security properties. PCFIRE's CFI proofs focus only on our source-code-based CFI, and we do not claim that PCFIRE (using AUSPICE) can prove the CFI of any safe program. We consider cooperative developers who wish to ensure the CFI of their programs, and we do not consider maliciously-written malware. Our current focus is on enabling preventative CFI which is automatically provable, and we plan to explore lowering the run-time overheads of our safety-checks in future.

3. APPROACH

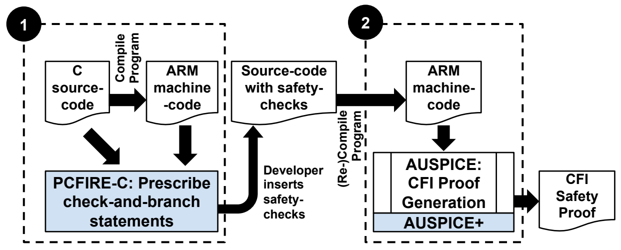

1
C source-code
Compile Program
ARM machine-code
PCFIRE-C: Prescribe check-and-branch statements
Source-code with safety-checks
Developer inserts safety-checks
(Re-)Compile Program
2
ARM machine-code
AUSPICE: CFI Proof Generation
AUSPICE+
CFI Safety Proof

Figure 1: Overview of PCFIRE's approach. Blue boxes indicate our contributions.

PCFIRE provides *preventative* enforcement of CFI, which (i) prevents the root-causes of CFI violations, (ii) enables developers to insert application-specific recovery behavior, and (iii) is automatically provable at the machine-code level. Figure 1 summarizes the two-step process for obtaining a program with provable preventative CFI. First, given the C source-code and (initially) compiled machine-code of a program, the PCFIRE-C tool prescribes safety-checks in C for the program. These safety-checks are based on a set of heuristics about program locations that require CFI safety-checks (§3.2). We **intentionally** leave it to developers to insert these safety-checks into their source-code, so they can specify application-specific recovery where necessary. Second, developers recompile their source-code (with the inserted C safety-checks), and pass the machine-code to AUSPICE, which automatically generates a CFI safety proof for the program (§3.3). If the CFI proof fails, AUSPICE returns the machine-code addresses where the CFI proof failed [31] (we will handle proof failures in future work). Third, we develop the AUSPICE+ extension to AUSPICE to automate CFI proof generation for programs with syscalls (§3.4).

3.1 Ensuring Control-Flow Integrity for ARM

3.1.1 Necessary and Sufficient Properties for CFI

We begin by describing the CFI safety property provided by PCFIRE, which is based on the safety theorem proved by AUSPICE [31]. For programs which AUSPICE proves are safe, the following safety properties hold:

Prop.1 The instructions in the program's text section loaded to memory cannot be modified,

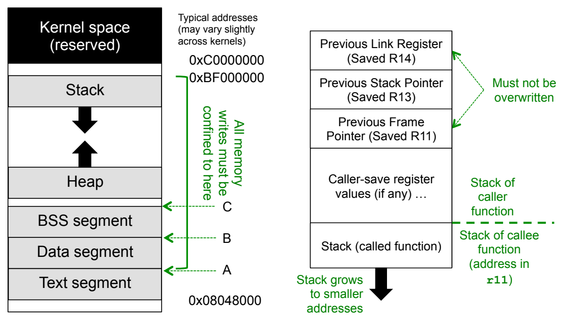
Prop.2 Function-return addresses saved to the stack cannot be modified, and

Prop.3 Only initially-loaded instructions are executed.

Together, these properties eliminate the root-causes of CFI violations. We explain how these properties are necessary and sufficient to ensure CFI holds for a machine-code program. The execution of a machine-code program possesses CFI when the control-flow of its machine-code obeys the CFG captured by its source-code (§1.1). Supposing the program has not been modified since compilation (which implies that the loaded instructions obey the source-code CFG of the program), then the program's CFI will be violated when its CFG is changed at run-time. The CFG of the program comprises vertices representing instructions, and edges representing control transfers between instructions. The three properties of AUSPICE's safety theorem prevent CFI violations by implying that the program's CFG cannot be changed at run-time (in the absence of unstructured jumps such as `goto` and `longjmp` statements, as stated in §2), as summarized in Table 1 (edges cannot be added to a CFG as this corresponds to adding a jump target, which requires an instruction, i.e., a vertex, to be changed).

Change to CFG	Effect on CFG	Protected by
Modify loaded instructions	Change CFG vertices	(1) prevents changing loaded instructions
Change function return address	Modify CFG edges	(2) prevents changing callee-saved registers
Inject and run instructions	Add CFG vertices	(3) prevents executing injected instructions

Table 1: CFI via AUSPICE's Safety Theorem


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

(a) Process Memory Layout for Linux (b) Activation Record for Function Calls

Figure 2: PCFIRE's CFI safety (diagram from [31]).

3.1.2 Allowed Program Behaviors for CFI

We concretely describe the behaviors that a Linux user-mode program executing on the ARM architecture must have for it to meet AUSPICE's safety properties. First, consider the memory layout for a user-mode process in Linux for a 32-bit architecture (Figure 2(a)). To prevent overwriting of loaded program instructions (Property 1), all memory writes must be restricted to addresses from `0xBF000000` (largest user-mode address) to the location `A` in Figure 2(a) (largest address where a program instruction has been loaded).

Second, consider the memory layout of the stack activation record for function calls as specified by the ATPCS [3] (Figure 2(b)). In a function call, the prologue of a function saves the values of callee-saved registers to the stack. Based on the ATPCS, these callee-saved registers include the link register (`r14`), stack pointer (`r13`) and frame pointer (`r11`). Also, the stack and frame pointers, which point to the end and start of the stack respectively, must not be overwritten, as they indicate where the link register is saved in memory. Hence, for Property 2 to hold, all saved link register, stack, and frame pointer values from all function calls prior to the current function must not be overwritten in memory. Hence, all memory writes must be to addresses smaller than the current value of the frame pointer.

Third, Property 3 states that only loaded program instructions can be run. This means that the value of the program counter must always be in the range of addresses where program instructions have been loaded.

3.2 Source-code Enforcement of CFI

Next, we construct safety-checks in source-code to ensure our 3 CFI safety properties (§3.1) hold. Safety-checks are needed before potentially dangerous C statements, which we will call *suspect statements*. Suspect statements may cause CFI violations, and are surrounded with safety-checks. Each safety-check is a *check-and-branch* statement in C: it *checks* if the suspect statement will cause a CFI violation when run: if so, it *branches* to an alternative statement (which can be a recovery action); if not, it allows the suspect statement to run. By running safety-checks before suspect statements, we can recover before a CFI violation occurs.

First, we consider the kinds of C statements where the safety properties may be violated. Properties 1 and 2 can be violated only when an instruction writes to memory. In a program without unstructured control-flow jumps (i.e., no `goto`, `longjmp`, explicit function pointers, or direct writing to the program counter), Property 3 can be violated only at function returns, as all other jump targets are statically fixed (as long as the loaded program text and program counter are not overwritten, and no injected instructions are executed). Then, since function return addresses are saved to the stack, ensuring Property 2 will ensure Property 3. Thus, we only need to ensure that C statements that write to memory obey Properties 1 and 2, for all three properties to hold.

Second, we consider the check-and-branch C statements needed to ensure Properties 1 and 2, which require the memory addresses being written to, to be within safe ranges. The check-and-branch statements need to extract the addresses written to by the C statement and ensure that the addresses are safe. The safe address range for Property 1 can be statically obtained from the machine-code (e.g., extract the address and size of the `text` section using GNU `readelf`), while the safe address range for Property 2 requires the frame pointer value (in register `r11`) to be extracted at run-time.

Third, we need to identify C statements that are potentially dangerous, which need to be surrounded by check-and-branch statements. While C statements that write to memory may violate Properties 1 and 2, AUSPICE can automatically prove that (the compiled machine-code of) certain classes of memory-write statements will not violate CFI, and we need to add check-and-branch statements only around the remaining statements.

3.3 Formal Proofs of CFI: AUSPICE

AUSPICE is a Hoare-Logic-based framework for automatically proving CFI safety in ARM machine-code [31].

We briefly describe AUSPICE; detailed proof rules can be found in [31]. AUSPICE is built on the Cambridge ARM model [8, 21], in the HOL4 interactive theorem prover [29]. PCFIRE uses AUSPICE to generate CFI safety proofs for ARM machine-code. AUSPICE can automatically prove CFI safety in programs whose CFI safety-checks comprise long instruction sequences, e.g., when safety-checks are provided in source-code, as compared to the simpler machine-code safety-checks in prior verified CFI techniques [38, 33]. AUSPICE uses Hoare Logic to reason about single instructions and basic blocks of instructions, and automatically inserts safety assertions at each instruction asserting the CFI safety properties described in §3.1.

AUSPICE begins by obtaining Hoare Logic theorems (Hoare triples) for each machine-code instruction from the Cambridge ARM model, and composes them to form Hoare triples about basic blocks (linear sequences) of instructions. Hoare Logic [13] describes programs as triples, ⊢ {p} c {q}, where p and q are predicates about machine resources, i.e., register values, flag values, and memory contents. c is a program (one or more ARM machine-code words), and p and q describe the state of the processor before and after running c respectively. Hence, Hoare Logic theorems describe the before-and-after effects of executing one or more instructions on processor state. Figure 3(a) shows an example Hoare triple for the instruction 0xE5832000 (“`str r2 [r3]`”).

$\vdash$ SPEC ARM_MODEL *Hoare triple with ARM instruction semantics*

Pre-state, p

$$(\texttt{aR 3w } r3 * \texttt{aR 2w } r2 * \texttt{aPC } p * \texttt{aMEMORY } df\ f * \texttt{cond}((r3\ \&\&\ 3w = 0w) \wedge (r3 \in df)))$$

$$\{(p, 0xE5832000w)\}$$ *Code, c*

$$(\texttt{aR 3w } r3 * \texttt{aR 2w } r2 * \texttt{aPC } (p + 4w) * \texttt{aMEMORY } df\ ((r3 =+ r2)\ f))$$ *Post-state, q*

(a) Example Hoare triple. p: instruction address, 0xE5832000: instruction described.

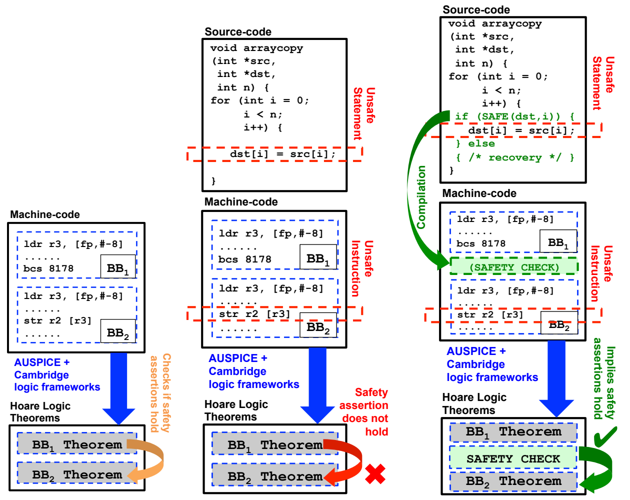
aR 3w $r3$	Register number 3w stores variable $r3$ (w indicates “3” is a fixed-width word.)
aPC p	Program counter stores variable p
aMEMORY df f	Memory has domain df and is represented by variable f, mapping addresses to stored words
$((r3 =+ r2)\ f))$	Memory f stores variable $r2$ at address $r3$

(b) Key concepts of the example Hoare triple.

Figure 3: Hoare triple for instruction “`str r2 [r3]`”.

Figure 3(b) explains key parts of the Hoare triple theorem in Figure 3(a). “$*$” is a Separating Conjunct from Separation Logic [26]. “$\texttt{cond}((r3\ \&\&\ 3w = 0w) \wedge (r3 \in df))$” states the memory alignment requirement for writes to the address $r3$, and that $r3$ is a valid address for memory f.

These Hoare triples can be augmented with *pre-conditions*, which are predicates that hold before an instruction executes (e.g., for statements in the body of “`if (i == j) {...}`”, the pre-condition $(i == j)$ holds), and with *assertions*, which are predicates we assume to be true. Then, AUSPICE instantiates its CFI safety properties (§3.1) at each instruction's Hoare triple as assertions to be proved. AUSPICE automatically discharges the safety assertions at each instruction where possible (e.g., when addresses written to are a constant offset from the frame pointer). Then, it carries out a proof search at the intra- and inter-procedural level. For each basic block, AUSPICE checks if its CFI safety assertions can be discharged by the pre-conditions of all its predecessor blocks (Figure 4(a)). For C statements (and their corresponding compiled instructions) with potentially unsafe operations, the Hoare triples for these instructions will have safety assertions that cannot be automatically discharged by the predecessor blocks, leading to a safety proof failure (Figure 4(b)). In §4.1, we describe how PCFIRE-C prescribes source-code safety-checks whose compiled machine-code have Hoare triples that have pre-conditions which imply that the safety assertions of the Hoare triples of unsafe instructions hold, enabling the AUSPICE CFI safety proof to succeed (Figure 4(c)).



```
void arraycopy
(int *src,
 int *dst,
 int n) {
for (int i = 0;
     i < n;
     i++) {
  if (SAFE(dst,i)) {
    dst[i] = src[i];
  } else
  { /* recovery */ }
}
```

```
ldr r3, [fp,#-8]
......
bcs 8178
(SAFETY CHECK)
ldr r3, [fp,#-8]
......
str r2 [r3]
......
```

(a) Proofs of CFI (b) Proof failure (unsafe program) (c) Provable CFI using C checks

Figure 4: How PCFIRE's source-code safety-checks yield machine-code programs with provable CFI. BB_1 and BB_2 represent basic block addresses.

3.4 CFI Proofs for Realistic Code: AUSPICE+

The Cambridge ARM model [21] does not model the user-mode-visible outcomes of syscall-servicing by an OS. Thus, AUSPICE is unable to prove the CFI of programs with syscalls. We extend AUSPICE with AUSPICE+ to support programs that invoke syscalls using the ARM `svc` instruction. AUSPICE+ is currently specific to Linux on the ARM platform, and we assume that the OS services syscalls as specified by the POSIX standard (for Linux).

We model the effects of syscall servicing that are visible to user-mode programs, to capture the effects on the CFI of the program. We formally encode our assumption that the OS services each syscall “correctly” (according to the POSIX standard) by constructing Hoare triples to represent these assumptions, which we add to the hypotheses of an AUSPICE+ CFI proof. Thus, for each machine-code `svc` instruction, which invokes a syscall, instead of obtaining a Hoare triple theorem from the Cambridge ARM model, we construct a Hoare triple *assumption* that encodes the POSIX specification of the syscall's behavior. We analyze the Hoare triples of the instructions leading up to the `svc` instruction to obtain the machine state (i.e., register values) prior to the `svc` instruction, which tell us the parameters passed to the OS on invoking the `svc` instruction. For instance, consider this instruction leading up to the `svc` instruction: `mov r7, #4 / svc 0x00000000`. For the Linux kernel on ARM, the syscall number is passed in register `r7` by convention. Hence, we know that syscall 4 is invoked (i.e., `write()`), and we construct the Hoare triple shown in Figure 5.

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

Figure 5: Constructed Hoare triple for `write` syscall.

The post-state of the Hoare triple for the `write()` syscall does not change the processor's state, except for register `r0`, which stores the return value from the syscall, as captured by the symbolic variable *rv*. For the `write` syscall, the OS outputs the contents of the buffer to the given file descriptor, without changing any of the user-mode visible state (register and flag values, and memory) of the program (apart from register `r0`). Note that the Hoare triple is repeated on the left of the turnstile "⊢", indicating that the Hoare triple is a hypothesis. Note, also, that the values for registers `r0`, `r1`, `r2`, `r14`, which provide the OS with the file descriptor, buffer address, buffer length, and return address respectively, contain symbolic variables. These variables are concretely filled in only at each call-site to the syscall. We also modified the composition of basic block theorems in AUSPICE to fill in concrete values for syscall arguments at each call-site.

For syscalls that modify user-mode processor state (e.g., `read()`, which copies its results to a specified location in the memory of the calling process), we construct a memory expression in the Hoare triple to represent this modified user-process memory. Then, the CFI safety assertion for the Hoare triple of the syscall will reflect the process's memory addresses that are written to by the syscall. Note that our approach does not capture aspects of system state that are not explicitly visible in user-mode (e.g., file descriptor mappings in `open`, and user-space memory mappings in `mmap`). Our current goal is to focus on proving CFI safety of user-mode programs, which is (directly) affected only by user-mode-visible state. We have implemented automatic Hoare triple construction for the following syscalls as an initial proof-of-concept, to provide basic I/O functionality: `read`, `write`, `open`, `close`, `exit`, `mmap`, `munmap`, `nanosleep`. AUSPICE+ added 1.2 KLOC of ML proof scripts for HOL4 to the existing 11.8 KLOC code-base of AUSPICE [31].

4. DESIGN AND IMPLEMENTATION

4.1 Source-code Safety-checks: PCFIRE-C

Next, we describe the implementation of the PCFIRE-C tool for prescribing safety-checks in C programs. PCFIRE-C takes as input: (i) C source files, and (ii) a single, statically linked and compiled binary file (with debug symbols), and prescribes safety-checks to ensure CFI. These safety-checks are check-and-branch C statements, which programmers insert in their programs. First, we describe our prescribed check-and-branch statements (§4.1.1). Second, we describe how to identify suspect statements needing check-and-branch statements (§4.1.2).

4.1.1 Check-and-Branch for C Programs

The key idea of PCFIRE's safety-checks is to provide a guard that ensures that suspect C statements with memory writes (which AUSPICE cannot automatically prove to be safe) will execute only if the target address being written to is safe with respect to PCFIRE's safety policy (§3.1). We need to ensure that the safety-checks, which are C statements, will be compiled to machine-code whose Hoare triples will provide the pre-conditions needed to discharge the safety assertions for the dangerous memory write instruction from the suspect statement (§3.3 and Fig. 4(c)). PCFIRE-C uses the Clang [2] compiler front-end for C programs to analyze the Abstract Syntax Tree (AST) of each suspect statement to construct the prescribed safety-checks.

The safety-checks, or guards, which PCFIRE-C constructs, are (memory) address comparison statements with two parts: (i) the address being written to by the suspect statement on the left-hand-side (LHS), and (ii) the upper and lower bounds allowed for the memory address being written to on the right-hand-side (RHS). First, PCFIRE-C walks the Clang AST of the suspect C statement to find the sub-expression for the memory address being written to. PCFIRE-C then converts the sub-expression into a form that enables the exact memory address being written to, to be checked in the guard statement. Table 2 shows how PCFIRE-C converts LHS sub-expressions into a comparable memory address for guard statements. For complex LHS expressions, PCFIRE-C will extract the type of the LHS of the expression, and suggest to programmers to extract the address being written to (e.g. for "`<lhs expr> = <expr>;`", use "`tmp_ptr = &(<lhs expr>); (*tmp_ptr) = <expr>;`").

Statement type	Original expression	Expression for safety-check
Prefix operation	`*++s = <expr>;`	`(s+1)`
Postfix operation	`*s++ = <expr>;`	`s`
Memory write	`*s = <expr>;`	`s`
Memory write	`*(s <binop> t) = <expr>;`	`(s <binop> t)`
Array access	`s[i] = <expr>;`	`(s + i)`

Table 2: LHS expressions for memory safety guards.

Second, PCFIRE-C adds the upper- and lower-bounds for allowed memory-write addresses to the RHS of the address comparison statements, as shown in Table 3. PCFIRE-C creates a conjunction of the conditions listed in Table 3, using the memory address expression from the last step as the LHS for each conjunct. PCFIRE-C first places a guess for the value of `text_hi`, as newly inserted safety-checks will increase the size of the program. PCFIRE-C provides a utility for programmers to query the recompiled binary for the new value of `text_hi` to insert in their source-code.

Additional code is required to dynamically extract the current value of the frame pointer from register `r11` before each safety guard. The value of the frame pointer must be stored in a register rather than a local variable stored on the stack to ensure that AUSPICE is able to use the value in its comparison. Figure 6 shows the macro used to extract the current value of the frame pointer to a temporary register

Name	Bound	Value	Origin
`text_hi`	Lower-bound	Highest-address of `text` section	Program text
`stack_lo`	Upper-bound	Highest-allowed address of program stack	Constant (0xBF000000)
`fp`	Upper-bound	Current frame pointer	Dynamically obtained (`r11`)

Table 3: RHS values for memory safety guards.

for checking against in the safety guard, and the C variable declaration required for a local variable stored in a register to store the frame pointer value.

```
#define GET_FRAME_POINTER(dest_var) \
  asm ( "mov r4,r11"                \
: "=r" (dest_var)                   \
: /* no inputs */                   \
: /* no clobber */)
register unsigned int r11_val asm ("r4");
```

Figure 6: Macro to extract frame pointer value, and local register variable to hold frame pointer value.

```
GET_FRAME_POINTER(r11_val);
if (((unsigned int)(s+i) <= STACK_LO)
   && ((unsigned int)(s+i) >= TEXT_HI)
   && ((unsigned int)(s+i) < (r11_val - (3*WORD_SIZE)))
   && (r11_val >= (3*WORD_SIZE))) {
      s[i] = <expr>;
} else { /* recovery here */ }
```

Figure 7: Full safety-check for statement, "`s[i] = <expr>;`" in a function with 3 callee-saved registers.

Figure 7 puts together the full PCFIRE-C safety guard for a suspect statement, which writes to memory, `s[i] = <expr>;`. The safety guard also allows programmers to specify their own recovery actions in the `else` branch of the safety guard, although we do not prescribe recovery actions in this work. In addition, `WORD_SIZE` stores the number of bytes used to represent a machine-address (e.g. 4 in a 32-bit architecture), and there is an additional guard conjunct, `(r11_val >= (N * WORD_SIZE))`. This guard is used to ensure that there is no arithmetic underflow of the computed address `(s+i)`, and the lower bound `(N * WORD_SIZE)` is needed if there are N callee-saved register values on the current function's stack. This ensures that no memory writes can overwrite the current function's callee-saved register values. PCFIRE-C extracts the number of callee-saved register values written to the stack, N, by parsing the `push` instruction in the prologue of the suspect statement's function.

4.1.2 Identifying Suspect Statements

PCFIRE-C identifies suspect statements by analyzing the (disassembled text from "`objdump -d`" of) compiled machine-code of target programs. PCFIRE-C first identifies the addresses of instructions that are suspects. Then, PCFIRE-C identifies the source-file and line number of suspect statements using the binary's debug information. This analysis is implemented using Python. We describe the analysis to identify instructions that would cause the CFI proof in AUSPICE to fail. The main goal of this analysis is to identify unsafe memory-write instructions for which additional branch pre-conditions, as provided by source-code safety-checks, are needed. Note that the analysis identifies potentially unsafe instructions, but is unable to check if the necessary branch safety pre-conditions are present.

As described in §1.1, memory-write instructions writing to a constant offset from the current frame pointer `fp` (e.g., "`str r3, [fp, #-8]`") can generally be automatically proved to be safe, but not instructions writing to computed addresses (e.g., "`str r2 [r3]`"). The analysis first narrows down its search to instructions that write to memory (specifically ARM's `str`, `strb` and `strh` instructions). The analysis ignores writes to a constant offset from the frame pointer, and reports the address of any other write instruction. The analysis ignores the `push` instruction, as we observed that `gcc`-emitted code only uses `push` in function prologues. The analysis extracts the number of callee-saved registers for each function from `push` instructions in function prologues to construct check-and-branch statements (§4.1.1).

4.1.3 Provability of Source-code CFI

To recap, we describe how PCFIRE-C's check-and-branch C statements result in machine-code that can be automatically proved to have CFI. First, the values of the bounds in the memory-address checks (§4.1.1) match the bounds in Properties 1 and 2 of the AUSPICE safety theorem (§3.1). Second, the logic expressions for the checked and written addresses in (i) the safety-check, and (ii) the suspect statement, are the same in the machine-code Hoare triples, as our construction uses the same C expression in (i) the LHS of each safety-check, and in (ii) the LHS of the memory address written to in each suspect statement. This enables AUSPICE to reason about the suspect statement using the safety-check, resulting in automatically-provable CFI.

The requirement that the logic expressions representing the checked and written addresses be equal implies that our safety-checks need to be adjacent to their suspect statements. Any statements (and hence machine-code) between a safety-check and its suspect statement are likely to cause AUSPICE's proof search to fail. This precludes us from optimizations such as moving safety-checks out of loops.

4.2 Supporting I/O Behavior

Next, we describe the practical steps for supporting C programs with I/O behavior through syscalls, whose machine-code can be automatically verified using AUSPICE+. Typically, developers perform I/O using an implementation of the C standard library, `libc`. However, `libc` implementations typically include additional support code, e.g., from user-space thread libraries, which (i) cannot be verified by AUSPICE due to unsupported concurrent behavior, and (ii) increase the size of the machine-code (when statically compiled), posing scalability challenges for AUSPICE.

We aim to support I/O behavior in programs, while ensuring the programs remain amenable to automatic CFI proofs. First, we use a custom GNU `ld` linker script which avoids running the `libc` initializer (`__libc_init()`) ahead of the application's `main()`. Instead, we construct a minimal initializer which only prepares command-line arguments (i.e., `argc`, `argv`) for the program. This minimizes the code that is included in the target program. Second, we implement thin wrappers around syscalls using hand-written assembly code, whose CFI safety AUSPICE+ can prove, and provide

Test Case	Source Lines (before)	Source Lines (after)	Instructions (before)	Instructions (after)	Size Increase	Run-time Slowdown	AUSPICE Proof Time
`arrcpy`	17	33	44	75	70%	113.3%	8.6 mins
`sort`	25	46	84	158	88%	72.3%	15.3 mins
`mibench stringsearch`	68	100	423	489	16%	213.2%	82.3 mins
`mibench crc32`	91	119	130	183	41%	43.1%	30.7 mins
`matmult`	30	47	134	153	14%	0.78%	6.14 hours
`bionic memcpy`	98	143	239	355	49%	800.8%	38.1 mins

Table 4: PCFIRE overheads: C lines, machine-code percentage size increase, and run-time slowdown.

C function prototypes for these wrappers. Developers can invoke the relevant syscall using these C function prototypes.

5. EVALUATION

5.1 Run-time Overheads and Safety Proof Times

Next, we evaluate the program size and run-time overheads of PCFIRE's safety-checks. We also report the time taken to prove the CFI of each program. Table 4 summarizes our results. We evaluate PCFIRE on six test programs:

1. `arrcpy`: Array-copy example in §1.1.
2. `sort`: Implementation of Selection Sort.
3. `stringsearch`: Boyer-Moore string search, part of MiBench [12] benchmark suite for embedded applications.
4. `crc32`: Checksum algorithm, part of MiBench [12] suite.
5. `matmult`: Matrix multiplication (has a triply-nested loop).
6. `bionic memcpy`: Efficient implementation of `memcpy` from the Bionic [1] C library for Android systems.

We compared the run-times of programs with and without safety-checks. All tests ran on the Raspberry Pi 1 Model B+ with a 700 MHz ARMv6 processor and 512 MB RAM with Linux 3.18. We report the average run-times over 1000 iterations of each program. The run-time slowdown depended on the proportion of each program's workload that invoked suspect statements (with safety-checks), and ranged from 0.78% for `matmult`, to 800.8% for `bionic memcpy`. `matmult` had the smallest increase in run-time, as its workload was mainly computation. `bionic memcpy` had the highest run-time overhead. This is a worst-case scenario for PCFIRE as it is made up entirely of memory writes requiring PC-FIRE's safety-checks. For programs with mixed workloads (e.g., `stringsearch`, `crc32`, `sort`), the run-time slowdown ranged from 43% to 213%.

Proof Times. We evaluated the time taken to prove each program's CFI on an Intel Core i7 2.6 GHz. The proof times were less than 90 minutes in all but one case. The proof time for `matmult` was higher at 6.14 hours, due to the large number of memory operations in the matrix multiplication loops. We contrast the proof time of 82.3 minutes for `stringsearch` with ARMor [38], which took 8 hours for its safety proof for `stringsearch` on an Intel Core i7 2.7 GHz.

5.2 Case Study 1: File-based I/O

Next, we show that PCFIRE-provided syscall wrappers can provide I/O functionality in realistic programs, and that AUSPICE+ can prove CFI in these programs. We implemented simple versions of three common text utilities: (i) `cat`, which outputs the contents of a file, (ii) `wc`, which counts the number of words in a file, and (ii) `grep`, which outputs lines from a file containing a given string. Table 5 summarizes the source- and machine-code sizes of each utility, the time taken to prove CFI, and the run-time and slowdowns for each program. We ran each program on a 10 MB input file over 5 iterations and we report the average run-times. We report the run-time slowdown of each "safe" utility, as compared to an "unsafe" version without the PCFIRE-prescribed safety-checks. The "safe" version of each utility slowed down between 0.39% to 2.71% as compared to the "unsafe" version without our safety-checks. This suggests that the slowdown due to PCFIRE's safety-checks in more realistic programs with a mixed workload is likely to be much less than shown in §5.1.

Since our goal was to show that we could implement programs with I/O, we did not aim to optimize our implementations. Nonetheless, we compared the run-times of our utilities with the system-provided versions for completeness. Both the safe and unsafe versions of each utility were significantly slower than their system-provided versions. Slowdowns ranged from 4.7x for `wc`, to 25.5x for `cat`, as the system-provided utilities used large input buffers to amortize the overheads of syscalls, whereas our utilities invoke the `read` syscall for each character to minimize proof times.

Next, we measured the effects of increasing input buffer sizes on our implementation of `cat`. Table 6 summarizes our results. As the buffer size is increased from 1 to 30 bytes, the run-time slowdown improves significantly from 2549% to 56.5%, while the safety proof time increases by 4x to 165.6 minutes. The proof time increases with larger input buffers

Prog.	C Lines	Instructions	Proof Time
`cat`	411	207	44.2 mins
`wc`	427	641	2.7 hours
`grep`	428	621	1.1 hours

	Run-times			Slowdown	
Prog.	Safe	Unsafe	Orig.	vs. Unsafe	vs. Orig.
`cat`	151s	149s	5.7s	0.96%	2549%
`wc`	25.2s	24.5s	4.4s	2.7%	479%
`grep`	37.3s	37.2s	4.9s	0.40%	655%

Table 5: CFI-proved versions of simple text utilities, their run-times, and run-time slowdowns.

Buffer Size	Proof Time	Run-time Slowdown vs. Orig
1 byte	44.2 mins	2549%
10 bytes	48.58 mins	148.9%
20 bytes	82.2 mins	61.5%
30 bytes	165.6 mins	56.5%

Table 6: Improved run-time slowdown but slower proof times with larger buffer sizes for `cat`.

as AUSPICE+ needs to check that each byte in the input buffer is safe. Hence, there is a trade-off between run-time performance and safety proof times.

5.3 Case Study 2: Raspberry Pi GPIO

Program	Source Lines	Instructions	Proof Time
`blink`	418	619	81.7 mins
`light`	429	854	112.5 mins
`lcd`	559	2286	21.8 hours
`fall-det`	923	3169	44.9 hours

Table 7: Raspberry Pi GPIO test programs.

Next, we show that PCFIRE-provided syscall wrappers can support I/O from external hardware, e.g., sensors, LEDs, and an LCD display. We inserted PCFIRE-prescribed safety-checks into all our test programs, and proved their CFI using AUSPICE+. We used the Raspberry Pi 1 Model B+ for our experiments, and we based our hardware access on the WiringPi C library [9], which provides access to Raspberry Pi's General Purpose I/O (GPIO) hardware interface. We ported a small number of functions in WiringPi for our programs by changing C library calls (e.g., `mmap()`) to calls to PCFIRE's syscall wrappers. We wrote three test programs using our ported version of WiringPi with PCFIRE-prescribed safety-checks: (i) `blink` periodically turns an LED on and off; (ii) `light` and turns on an LED when the ambient light falls below a threshold, and (iii) `lcd` outputs a string to a 16×2 monochrome LCD. We also implemented a fall detector (`fall-det`) based on Jia's algorithm [14] using the ADXL345 accelerometer with the Raspberry Pi. Table 7 summarizes our results.

6. DISCUSSION

Tradeoffs for Preventative CFI. Our evaluation shows that our prescribed safety-checks introduce run-time slowdowns of up to 800%, and between 43% to 213% for most cases. Past CFI techniques that detect violations after-the-fact, e.g., Abadi et al. [5], XFI [33], and CCFIR [6], incurred average overheads of 16%, 11%, and 3.6% respectively. ARMor [38], which like PCFIRE uses preventative checks (but does not allow recovery), incurred overheads of 240% on the `stringsearch` benchmark, as compared to our overhead of 213%. This suggests that preventative CFI fundamentally incurs higher overheads, as other CFI techniques that only stop on CFI violations need much fewer checks than preventative CFI, which requires checks at all suspect memory-writes. We believe this is an acceptable trade-off for applications (e.g., safety-critical systems) needing robust recovery.
Strategies for CFI recovery actions. While it is not our goal to suggest recovery actions from potential CFI violations that have been prevented, different strategies can be used by programmers in writing recovery code in the `else` branch of our inserted safety-checks. We posit that programmers need a comprehensive approach for recovery actions. For instance, library functions (e.g., `memcpy`) may not have enough application context to robustly handle CFI violations, and programmers should signal to the callers of library functions to handle the potential violation.
Allowed program behaviors. PCFIRE's (and AUSPICE's) requirement that memory-writes must be to addresses smaller than the current function's frame pointer prevents functions from changing memory (and hence any local variable) on their callers' stacks. As PCFIRE does not currently support heap memory management due to our exclusion of `libc`, any memory to be changed by callee functions must be globally declared (so that it is in the `bss` or `data` sections). This reduced programmer convenience is necessary for AUSPICE's proof automation [31], otherwise, run-time safety-checks will need complex stack analyses, which may not be feasible.
Library functions. Currently, we use syscall wrapper functions to directly invoke syscalls. However, programmers are not limited to using our syscall wrappers for I/O. Safe versions of most C standard library functions can be developed with PCFIRE-C-suggested safety-checks. We have begun writing and using safe versions of some C library functions in our example I/O programs in §5.2 and §5.3.
Source-code vs. Machine-code proofs. Proofs about source-code require compilers to correctly compile a program for the proven properties to hold in the compiled machine-code. Yang et al. [36] found many bugs in mainstream C compilers (e.g., GCC), such as incorrect unsigned integer behavior, which can affect CFI safety. Our machine-code-level proofs are not affected by such compiler bugs.
Other Limitations. While we currently do not support compiler optimizations, we intend to explore supporting them in future. Some safety-critical software standards discourage or recommend additional tests for compiler optimizations (e.g., §4.4.2, §6.4.4.2 in [25]), hence we do not see our (current) lack of support for compiler optimizations to be significant. Also, while we do not support applications in an Real-Time OS (RTOS), we believe our approach can be adapted for preventative CFI for RTOS applications (e.g., RTOS tasks) by: (i) analyzing each task as a sequential program, and (ii) modeling the behavior of the RTOS as observed by each RTOS task as hypotheses, similar to how AUSPICE+ models syscall behavior. While the prescribed safety-checks will increase the run-time of RTOS tasks, we believe this increased run-time can be accounted for using techniques such as WCET (Worst-case Execution Time) analysis [24], as our safety-checks consist of only C `if` statements.

7. RELATED WORK

Many techniques have been proposed for CFI, and the closely related property of Software Fault Isolation (SFI) [35]. Most techniques detect changed control-flow, and stop the program's execution [5, 33, 6, 22]. They do not detect the root-causes of CFI (due to high run-time costs), and thus do not allow recovery from potential CFI violations. Most techniques use machine-code safety-checks inserted automatically in binaries [5, 33, 6, 22, 38, 19, 37]. These techniques provide programmer convenience, but do not allow programmers to specify application-specific recovery code, unlike PCFIRE's source-code safety-checks. CFI and SFI have been verified formally [33, 38, 19], but their verification requires automatically-inserted machine-code safety-checks, as compared to AUSPICE [31], which PCFIRE uses, which supports source-code safety-checks. Goel et al. [28] enabled proofs about x86 programs with syscalls in the ACL2 prover for functional-correctness, whereas AUSPICE+ automates simpler CFI proofs. CFI can also be achieved in hardware, e.g., in SOFIA [7], although this requires processor modifications, and incurs run-time slowdowns as well.

CFI for programs can also be achieved using safe dialects of C. Cyclone's [30] run-time checks do not allow application-specific recovery as they are inserted post-compilation. Control-C [17] restricts C behaviors, e.g., no pointer arithmetic, while PCFIRE does not, as our CFI proofs are at the machine-code level. CompCert [18] is a verified compiler whose correctness proof implies memory safety for unambiguous programs, but CompCert does not generate safety proofs for such programs, whereas PCFIRE generates proofs of CFI safety. CCured [11] automatically inserts C safety-checks for memory-safety at large-scales, and does not enable programmers to implement their own recovery actions. Verification efforts for safety-critical software in medical devices have focused on functional correctness [23, 15], which is orthogonal to CFI, which is an implementation-level safety property.

8. CONCLUSION AND FUTURE WORK

We have presented PCFIRE, a novel approach for provable and preventative Control-Flow Integrity (CFI) for C programs compiled to the ARM architecture. PCFIRE is a step towards CFI for realistic embedded applications by enabling programmers to specify application-specific recovery actions (through source-code safety-checks), which are important in safety-critical embedded software, and by supporting automated CFI proofs in programs with syscalls using AUSPICE+. We have demonstrated PCFIRE's flexibility through a range of case-studies, from real-world benchmarks [12], to programs containing I/O, for which we could prove CFI automatically. In future, we intend to improve PCFIRE's safety-checks using AUSPICE's proof failures, improve the run-time efficiency of PCFIRE's safety-checks, and improve AUSPICE's proof times for larger programs.

9. REFERENCES

[1] Bionic. http://bit.ly/1V0cJl3.

[2] clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[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] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *ACM CCS*, 2005.

[6] C. Zhang et al. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Security & Privacy*, 2013.

[7] R. de Clercq et al. SOFIA: Software and Control Flow Integrity Architecture. In *DATE*, 2016.

[8] A. Fox. Formal specification and verification of ARM6. In *TPHOLs*, 2003.

[9] G. Henderson. WiringPi. http://wiringpi.com/.

[10] G. Klein et al. seL4: Formal verification of an OS kernel. In *SOSP*, Oct 2009.

[11] G. Necula et al. CCured: Type-Safe Retrofitting of Legacy Code. In *POPL*, 2002.

[12] Guthaus, M. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE WWC Workshop*, 2001.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), Oct. 1969.

[14] N. Jia. Detecting Human Falls with a 3-Axis Digital Accelerometer, 2009. http://bit.ly/23fXhFE.

[15] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In *TACAS*, 2012.

[16] J. Knight. Safety Critical Systems: Challenges and Directions. In *ICSE*, 2002.

[17] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems. In *CASES*, 2002.

[18] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[19] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security*, 2006.

[20] C. Miller and C. Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. http://bit.ly/1Xk71rn.

[21] M. Myreen, A. Fox, and M. Gordon. Hoare Logic for ARM Machine Code. In *FSEN*, 2007.

[22] B. Niu and G. Tan. Modular Control-Flow Integrity. In *PLDI*, 2014.

[23] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. Safety-critical Medical Device Development Using the UPP2SF Model Translation Tool. *ACM TECS*, 13(4s), Apr. 2014.

[24] R. Wilhelm et al. The worst-case execution-time problem–Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[25] Radio Technical Commission for Aeronautics (RTCA). DO-178C: Software Considerations in Airborne Systems and Equipment Certification, 2012.

[26] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, 2002.

[27] S. Checkoway et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security*, 2011.

[28] S. Goel et al. Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls. In *FMCAD*, 2014.

[29] K. Slind and M. Norrish. A Brief Overview of HOL4. In *TPHOLs*, 2008.

[30] T. Jim et al. Cyclone: A Safe Dialect of C. In *USENIX ATC*, 2002.

[31] J. Tan, H. Tay, R. Gandhi, and P. Narasimhan. AUSPICE: Automatic Safety Property Verification for Unmodified Executables. In *VSTTE*, 2015.

[32] N. Y. Times. F.D.A. Deal Leads to Recall of Infusion Pumps, May 2010. http://nyti.ms/1TEGK8a.

[33] U. Erlingsson et al. XFI: Software Guards for System Address Spaces. In *OSDI*, 2006.

[34] U.S. F.D.A. MAUDE Adverse Event Report., Aug 2007. http://1.usa.gov/25NWCKC.

[35] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *SOSP*, 1993.

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.

[37] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security*, 2013.

[38] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: Fully Verified Software Fault Isolation. In *EMSOFT*, 2011.