Prescriptive Safety-Checks through Automated Proofs for Control-Flow Integrity

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Jiaqi Tan

B.S., Computer Science, Carnegie Mellon University
M.S., Computer Science, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

November, 2016

To my parents, Tan Yew Koon and Leong Sau Wan, who have always encouraged me, and to my wife, Neo Wei Ling, for her love, kindness, support, and for always believing in me.

Acknowledgments

This dissertation is the culmination of what has been an enjoyable and meaningful journey for me, which would not have been possible without the support, encouragement, and guidance of many people, and I would like to express my gratitude to those who have helped shape this research.

First, I would like to thank my parents, Tan Yew Koon and Leong Sau Wan, who have always encouraged me to chase my dreams since I was young, and who have always gone the extra mile to support me. I would also like to thank my sister and brother-in-law, for their love and concern, and for holding the fort at home. I would also like to thank my parents- and siblings-in-law, for their support, and their visits to Pittsburgh over the years, making home feel a little less distant during the years we have been away from home. Most of all, I would like to thank my wife, Neo Wei Ling, for her love, constant encouragement, and unwavering support, who has always kept me grounded, and who never stopped believing in me, without whom this journey would not have been possible.

Second, I would like to thank my dissertation advisor, Prof. Priya Narasimhan, for her support, guidance, encouragement, and counsel during my graduate studies. Her keen technical insights, deep understanding of trends in technology, and acumen in understanding and communicating research results, have been instrumental in developing this dissertation. Priya's encouragement to focus on contemporary systems have been an important part of this dissertation. Most of all, I would like to thank Priya for the privilege of working with her, during which time I have learned how to identify problems, develop practical solutions, and effectively present and communicate results.

Next, I would like to thank my thesis committee members, Prof. Anupam Datta, Prof. André Platzer, and Prof. Magnus Myreen, who have provided me with invaluable feedback on my dissertation. Their suggestions, comments, and thoughtful reading have helped greatly to improve the overall quality of my thesis. Anupam suggested sharpening our initial work on generic safety-properties to focus on Control-Flow Integrity; André helped sharpen the formalisms in my dissertation; Magnus provided many helpful code suggestions for my logic approach. I would also like to thank Prof.

Lujo Bauer for making himself available to attend my thesis defense, and for providing invaluable feedback at my thesis defense and on my dissertation.

I would also like to thank Dr. Arie Gurfinkel and Dr. William Klieber at the Software Engineering Institute at Carnegie Mellon for feedback and comments on my dissertation research. I first learned about software security in Prof. Anupam Datta's and Prof. Lujo Bauer's class. Prof. Edmund Clarke's class introduced me to formal methods and interactive theorem proving. Prof. Jonathan Aldrich introduced me to program analysis, from which I have borrowed many principles in my dissertation. I would also like to thank Prof. Greg Ganger, whose guidance and feedback over my years at the Parallel Data Lab have helped shape my research, and for the opportunity to showcase my research at numerous Parallel Data Lab Retreats. I would also like to thank Prof. Garth Gibson, from whom I learnt to think about operating systems and distributed systems at a deeper level. I would also like to thank Dr. Lily Mummert and Dr. Steve Schlosser, with whom I was fortunate to have had the opportunity to work with during an internship at Intel Labs Pittsburgh, and Prof. David O'Hallaron, to whom I had the opportunity to present my work during my internship.

I would also like to thank Dr. Lu Zhao, formerly from HP Fortify, who provided me with the code associated with, and answered many of my questions about his research, to which my dissertation is closely related. I would also like to thank Jeremy Richards from SAINT Corporation, who shared with me some of his findings in his medical device vulnerability research.

I have also had the pleasure of working with other brilliant researchers during my graduate studies. Hui Jun Tay was instrumental in the empirical evaluation in my dissertation, and was key in developing the embedded systems examples. Utsav Drolia helped sharpen the communication of some of the key ideas in my dissertation. Nathan Mickulicz, Dr. Rolando Martins, Dr. Michael Kasick, Dr. Wesley Jin and Prof. Rajeev Gandhi also provided me with invaluable feedback on my research, and thanks also go to Dr. Soila Kavulya for providing me with help and pointers along the way. I would also like to thank Prof. Tudor Dumitraș for giving me the opportunity to work with him on his Ph.D. research as an undergraduate, and for the guidance that he had given me.

I would also like to thank my colleagues at DSO National Laboratories in Singapore, who also provided me with feedback on my research along the way during my graduate studies. In particular, I would like to thank Dr. Tan Yang Meng, for encouraging me to embark on research in formal methods. I would also like to thank Zhuang Xinyu, Koh Ming Yang, and Koh Chuen Hoa, for their helpful discussions, comments and feedback on my dissertation topic, and on paper drafts.

I would also like to thank our friends in Pittsburgh who have made home feel a little less far

away, for their friendship and support: Somchaya Liemhetcharat, Junyun Tay, Carol Chan, Francisco Garrido, Danny Koh, Jason Ng, Rodney and Suzanne Fu, Yanchuan Sim, Lionel Wong, Chong Wen Haw, Lee Sin Yee, Andre Ng, Kee Wei Lam, Joshua Tan, Diana Seah.

I would also like to thank Karen Lindenfelser, Joan Digney, Jennifer Zeni, Toni Fox, Nathan Snizaski, and Samantha Goldstein for their administrative support, which has helped to ensure a smooth journey through my graduate studies at Carnegie Mellon. I would also like to thank Prof. Mark Stehlik and Prof. Scott McElfresh for their advice as my undergraduate academic advisors during my time at Carnegie Mellon.

During my dissertation research, I received financial support from DSO National Laboratories, Singapore, through the DSO Postgraduate Scholarship. I also received the support of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and Carnegie Mellon's CyLab and Parallel Data Lab.

Abstract

Embedded software today is pervasive: they can be found everywhere, from coffee makers and medical devices, to cars and aircraft. Embedded software today is also open and connected to the Internet, exposing them to external attacks that can cause its Control-Flow Integrity (CFI) to be violated. Control-Flow Integrity is an important safety property of software, which ensures that the behavior of the software is not inadvertently changed. The violation of CFI in software can cause unintended behaviors, and can even lead to catastrophic incidents in safety-critical systems.

This dissertation develops a two-part approach for CFI: (i) prescribing source-code safety-checks, that prevent the root-causes of CFI, that programmers can insert themselves, and (ii) formally proving CFI for the machine-code of programs with source-code safety-checks. First, our prescribed safety-checks, when applied, prevent the root-causes of CFI, thereby enabling software to recover from CFI violations in a customizable way. In addition, our prescribed safety-checks are visible to programmers, empowering them to ensure that the behavior of their software is not inadvertently changed by the prescribed safety-checks. However, programmer-inserted safety-checks may be incomplete. Thus, current techniques for proving CFI, which assume that safety-checks are complete, may not work. Second, this dissertation develops a logic approach that automates formal proofs of CFI for the machine-code of software containing both source-code CFI safety-checks and system calls. We extend an existing trustworthy Hoare logic with new proof rules, proof tactics, and a novel proof-search algorithm, which exploit the principle of local reasoning for safety properties to automatically generate CFI proofs for the machine-code of programs compiled with our prescribed source-code safety-checks.

To the best of our knowledge, our approach to CFI is the first to combine programmer-visible source-code enforcement mechanisms for CFI–enabling programmers to customize them and observe that their software is not inadvertently changed–with machine-code proofs of CFI that can be automated, and that does not require a trusted or verified compiler to ensure its proven properties hold in machine-code.

We evaluate our CFI approach on realistic embedded software. We evaluate our approach on the MiBench and WCET benchmarks, implementations of common file utilities, and programs interfacing with hardware inputs and outputs on the Raspberry Pi single-board-computer. The variety of our target programs, and our ability to support useful features such as file and hardware inputs and outputs, demonstrate the wide applicability of our approach.

Contents

Appendices

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Software is all around us today. With the rise of the Internet-of-Things (IoT) phenomenon [1], everyday objects from household appliances to automobiles are equipped with microprocessors. With this rise in the adoption of IoT devices, there is a growing amount of embedded software, whose primary role is to interact with physical objects [2], in these IoT devices. While embedded software is not new, the embedded software in today's IoT devices is different in a number of ways. First, embedded software today is more open and connected than before: not only do IoT devices have networking capabilities, they are increasingly connected to the Internet. Gartner estimates that there will be 6.4 billion "connected" IoT devices in 2016, representing an increase of 30% over 2015 [3]. Second, embedded software is becoming pervasive in our everyday lives: they are no longer limited to specialized applications such as industrial control systems, and are now in application domains such as medical devices, automotive systems, and household appliances [4]. Third, with the rapidly increasing computational power and falling cost of microprocessors, the hardware that embedded software runs on today is more sophisticated, enabling embedded software to run as user-mode applications on full-fledged operating systems (OSes) [5].

The open, connected, and pervasive nature of embedded software today poses new challenges for ensuring the security of embedded software. First, the open and connected nature of embedded software in IoT devices increases the attack surface of these devices, as attackers can now remotely compromise the embedded software in IoT devices. Security researchers have found and identified numerous IoT devices that contain vulnerable software that can be remotely compromised [6, 7, 8]. This increases the need for embedded software to be secured against external inputs, which can now be received over the Internet from unknown and potentially malicious sources. Second, the pervasive nature of embedded software today has resulted in the introduction of embedded

software into new application domains, some of which are safety-critical, such as medical devices and automotive systems. Thus, there is a need for programmers to use high-assurance techniques to ensure the security of their embedded software for such safety-critical domains, where failures can "result in loss of life, significant property damage, or damage to the environment" [9]. In addition, the pervasive nature of embedded software has resulted in large amounts of embedded software being written. This large amount of embedded software makes it helpful for high-assurance techniques for security to be automated, and to not require a large amount of manual effort or expert knowledge from programmers, so that programmers can cope with the large amount of code that needs to be secured. Third, some of the high-assurance techniques for security are currently targeted at embedded software that runs "bare-metal" directly on a processor without an OS. However, such techniques will not apply to embedded software that runs as user-mode applications in a full-fledged OS, and new techniques are required.

Many techniques have been developed to address the security challenges of embedded software today. These range from ensuring that sensor inputs are trustworthy, to developing secure communications protocols, and to developing cryptographic algorithms that are sufficiently lightweight for executing on processors with limited computational capabilities [10, 11]. In addition, in high-assurance embedded software development, particularly for safety-critical devices such as medical devices, techniques have been developed to ensure the functional correctness of their software [12, 13].

However, the security guarantees established by these techniques can be undermined by implementation-level bugs that give rise to software vulnerabilities. The open and connected nature of today's embedded software, particularly in IoT devices, exacerbates this risk of software vulnerabilities being exploited by remote attackers. Attackers can then cause the software to crash, or even hijack and modify the execution of the software, causing the software to behave in ways other than it was intended to. **Control-Flow Integrity (CFI)** is an important security property for software which can protect the software from vulnerabilities that can be exploited due to external inputs. CFI ensures that the execution of software cannot be circumvented by external inputs, and prevents the software's behavior from being unexpectedly changed at run-time.

This dissertation has developed an approach to Control-Flow Integrity (CFI) to address the security challenges faced by today's open and connected embedded software, that is also amenable to the pervasiveness of today's embedded software. We develop an approach for programmers to attain the security property of CFI in their embedded software, such that: (i) the enforcement mechanisms for

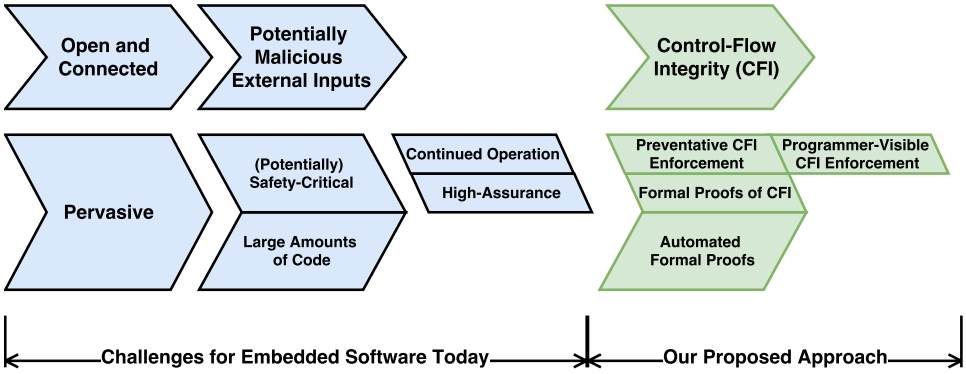


Figure 1.1. Challenges for security in embedded software, and how our approach addresses each challenge.

achieving CFI are prescribed to programmers, making these mechanisms visible to programmers, thus empowering programmers to take care to ensure that these mechanisms do not inadvertently change the behavior of their programs, (ii) CFI is achieved in a preventative way that prevents CFI violations from occurring, and (iii) the CFI achieved can be formally proved in an automatic manner without manual or expert inputs. Our approach to CFI addresses the security challenges faced by embedded software today in the following ways, as illustrated in Figure 1.1. First, CFI protects the execution and behavior of embedded software from being circumvented and modified by inputs that may be malicious due to the open and connected nature of embedded software today. Second, by prescribing mechanisms for enforcing CFI, and by ensuring that these mechanisms are programmer-visible, we enable programmers to ensure that our prescribed CFI enforcement mechanisms do not interfere with the safety-critical functionality of their software, since programmers are given the discretion to apply these prescriptions to their programs. Third, by preventing CFI violations from occurring before-the-fact, our approach to CFI enables programmers to recover their software from potential (but prevented) CFI violations, which is important for the continued operation of safety-critical software that must not fail. Fourth, by providing formal proofs of CFI, our approach provides CFI in a high-assurance manner, which is important for embedded software in safety-critical applications. In addition, by automatically generating proofs of CFI without manual inputs, programmers can easily utilize our approach to achieve CFI in a high-assurance manner for their software without requiring onerously large amounts of effort (e.g., to annotate code) or knowledge of formal methods.

Our approach to CFI for embedded software consists of two parts: (i) an **enforcement approach**, which prescribes mechanisms to enforce CFI in programs, and (ii) a **logic approach**, which uses formal logic to prove that CFI holds in programs that use our enforcement approach.

Our approach to CFI prescribes preventative enforcement for CFI at the source-code level, and provides automated formal proofs of CFI at the machine-code level. I argue that CFI violations are best discerned at the machine-code level, where they manifest directly as unexpected changes to the machine-code CFG (Control-Flow Graph, a directed graph in which the nodes represent basic blocks (linear sequences of instructions) and the edges represent control- flow paths between basic blocks [14]) of a program. While remedies to prevent CFI violations can be introduced at the machine-code level, I argue that remedying CFI violations is ideally performed at the source-code level, which is visible to programmers. This enables programmers to ensure that the CFI mechanisms do not break the programmer-expected functionality of the program. Informally speaking, there are two perspectives of the program and two places where we could seek to address the problem of CFI violations: the machine-code level and the source-code level. We advocate for an approach where we detect the problem at the place where it manifests (machine-code level), and where we fix the problem through prescriptions at the place that affords programmers the opportunity to take their program's functionality into account (the source-code level). This allows programmers to ensure the continued operation of any safety-critical software that must not be stopped, even in the face of a CFI violation. Specifically, our approach translates into three aspects: (i) providing CFI enforcement at a programmer-visible level through prescriptions of CFI mechanisms in the program's source-code, while leaving it to programmers to apply these prescriptions to their source-code (so that programmers can take care to ensure that the functionality of their programs is not inadvertently changed), (ii) proving CFI at the machine-code level of a program, and (iii) enabling these proofs to be generated automatically, without programmer intervention or manual inputs.

Previous techniques for providing CFI mechanisms and verifying CFI have done so either entirely at the machine-code level, or entirely at the source-code level. Abadi et al. [15], XFI [16], PittSFIeld [17] and ARMor [18] all provide CFI mechanisms at the machine-code level by using binary-rewriters to insert machine-code CFI mechanisms into the binaries of programs. Hence, their CFI enforcement mechanisms are not visible to programmers, and may change the functionality of these rewritten programs. These techniques are able to automatically verify the CFI they provide at the machine-code level as their inserted CFI mechanisms are designed to relieve their verification from requiring manual inputs, e.g., loop invariants. While these techniques can automatically verify the CFI they provide at the machine-code level, the use of machine-code-inserted CFI enforcement mechanisms may introduce behaviors that are unexpected to programmers, which is undesirable for software in safety-critical systems.

On the other hand, VCC [19] and VeriFast [20] are two software verifiers that can verify security properties such as CFI in the source-code of C programs, and they leave it up to programmers to ensure that the source-code of their programs is safe. While this allows programmers to insert CFI enforcement mechanisms into their source-code and verify that CFI holds, this verification cannot be done automatically. This is because these software verifiers (VCC, VeriFast) require specialized programmer inputs such as loop invariants and pre- and post-conditions for functions. In addition, the CompCert verified compiler [21] has been proved to correctly compile well-formed C programs. While CompCert guarantees the memory safety of the machine-code compiled from well-formed C programs with no undefined behavior, which implies Control-Flow Integrity, CompCert is unable to identify when an input source program contains undefined behavior, and CompCert is unable to generate a certificate proof of memory safety of their compiled programs (even for safe programs). In contrast, our approach automatically generates CFI proofs for machine-code programs (containing safety-checks prescribed by our enforcement approach).

By facilitating CFI at a programmer-visible level through prescribed safety-checks in a program's source-code, and proving that CFI holds at the machine-code level of a program, my dissertation enables CFI mechanisms to be implemented by programmers in a way that takes into account the functionality of their programs, while still enabling proofs of CFI to be generated automatically.

1.1 Control-Flow Integrity

Control-Flow Integrity (CFI) [15] is an important property to ensure in embedded software. CFI can be defined as follows. Typical software execution, for any type of system, can be captured through a Control-Flow Graph (CFG), which is a directed graph in which the nodes represent basic blocks (linear sequences of instructions) and the edges represent control-flow paths between basic blocks [14]. CFI implies that the dynamic execution of the system's software follows only a predetermined Control-Flow Graph; *any deviation from this predetermined CFG is a violation of CFI* [15]. Therefore, CFI violations can lead to unpredictable behavior in software, which can have highly undesirable consequences in embedded and safety-critical systems.

As an example, the Baxter Colleague 3 drug infusion-pump, a safety-critical medical device, was recalled because the infusion-pump unexpectedly ceased operation for actual patients. In one incident in 2007, a patient's death was directly attributed to the stoppage of drug delivery due to a malfunctioning Baxter Colleague 3 infusion-pump [22]. The malfunction in the infusion-pump's

software was traced to a buffer overflow, which caused the software to incorrectly and unexpectedly stop operating, effectively resulting in a CFI violation.

CFI violations in safety-critical systems have also been found to result in potentially-exploitable security vulnerabilities that can lead to fatal accidents. Checkoway et al. found that the media-player in a consumer automobile had vulnerabilities that could be exploited to compromise the CFI of the media-player's firmware [23]. While the media-player in an automobile is not typically considered to be a safety-critical (sub)system, it is possible for non-safety-critical subsystems with vulnerabilities to be exploited to gain access to other networked safety-critical subsystems. Thus, CFI is an important safety and security property of software, and is needed to ensure that the execution of the software is not subverted or modified in unexpected ways.

When ensuring the CFI of embedded software, it is also important to ensure that the software is able to continue operating even when faced with potential CFI violations. This means that CFI must be ensured in a **preventative** way that prevents the root-causes of CFI violations from occurring. If CFI violations are detected only after they have occurred, the violation cannot be undone, and the software must be stopped to prevent further unpredictable behavior. However, stopping embedded software that is safety-critical can lead to catastrophic accidents. For example, the software in a Baxter Colleague 3 drug-infusion pump, a safety-critical medical device, unexpectedly stopped operating after a buffer-overflow was detected, resulting in the death of the patient [22]. Thus, techniques for ensuring the CFI of embedded software must enable continued operation of the software by preventing the root-causes of CFI from occurring.

In addition to preventing the root-causes of CFI violations, techniques for ensuring the CFI of embedded software must also enable programmers to customize the recovery actions taken in response to a potential CFI violation. This is because embedded software is used in many different applications and domains, and the appropriate response to a potential CFI violation to ensure continued operation of the software is likely to be different for different application domains, and even for different functions in the same application. Thus, the CFI enforcement mechanisms must be **customizable** by programmers. This means that CFI enforcement mechanisms must be introduced at a level of abstraction that is visible by programmers, so that they can easily customize the recovery actions at the level of abstraction they work with.

Finally, to provide high-assurance of the CFI of embedded software (i.e., that its CFI will not be violated), it is highly desirable to obtain a formal proof stating that the CFI of a piece of embedded software holds. With the rapid growth in the number of embedded devices today [24], there will be

a large amount of embedded software written. In addition, most programmers who write embedded software are not likely to have in-depth knowledge of formal methods. Thus, it is not feasible to manually construct proofs of CFI for software, due to the large amount of embedded software in today's rapidly growing number of IoT devices, and due to the difficulty of training programmers to use formal methods tools to construct proofs of CFI. Hence, it is desirable for CFI to be achieved in a way that CFI proofs can be obtained in a fully automated manner, without any manual effort or specialized inputs from programmers.

1.2 Challenges for CFI for Embedded Software

At a high level, Control-Flow Integrity (CFI) violations occur in software when the software executes in an unexpected way. The expected behavior of a program can be considered to be captured by its source-code, which represents the original, statically-specified (in source-code) behavior of the program. Hence, the Control-Flow Graph (CFG) of the source-code of the program represents the original statically-specified behavior of the program, while the actual behavior of a program is the sequence of machine-code instructions that are executed by the processor that the program runs on. Then, CFI violations occur when the sequence of machine-code instructions executed by the processor is not present in the Control-Flow Graph (CFG) of the source-code of the program.

To provide mechanisms in software to ensure that its CFI holds, the main challenge is to address the gaps between the source-code and machine-code levels of abstraction, that allow CFI violations to occur. If the machine-code of a program that has been loaded to memory cannot be modified, then the execution of the machine-code can differ from the behavior specified in its source-code only when the instruction being executed is an indirect jump, whose jump target (i.e., address of its next instruction) is dynamically specified in memory. Thus, the main abstraction gap between machine-code and source-code that needs to be addressed is the presence of dynamically-specified jump targets that can affect the program's execution.

The main approach for CFI in current techniques is to insert mechanisms such as safety-checks in the target software to be protected. We explore some of the challenges that current proposed mechanisms for CFI would pose for today's open, connected, and pervasive embedded software.

1. **Preventing CFI violations after-the-fact:** Many of the current CFI techniques detect the end-results of CFI violations by checking dynamic-jump targets for corruption [15, 16, 18, 25, 26]. Such techniques detect CFI violations after-the-fact: when dynamic-jump targets

have been corrupted, the CFI violation has already occurred. Unfortunately, such after-the-fact detection of CFI violations makes it impossible for the software to recover from the CFI violation to continue operation, since the original dynamic-jump targets have already been corrupted. In this dissertation, we would like to enable CFI violations to be detected in a preventative way. This would make the recovery and continued operation of the target software possible, which is highly desirable for embedded software.

2. **Customizable recovery actions:** Current CFI techniques [15, 16, 18, 25, 26] insert safety-checks automatically in the machine-code of target programs by rewriting program binaries after compilation. First, safety-checks are inserted at the machine-code level after compilation. This makes it difficult for programmers to modify these safety-checks to specify appropriate recovery actions when CFI violations are detected. This is because programmers typically work with the source-code of programs before compiling the program, and programmers typically do not directly modify the machine-code of compiled programs. Second, the automatic insertion of safety-checks also results in the same action being used for all parts of the program in response to detected CFI violations. Current techniques all stop the target program's execution upon detection of a CFI violation, to prevent further unexpected behavior. In this dissertation, we would like to enable programmers to customize the recovery actions in their target programs that are taken when CFI violations are detected. This requires CFI safety-checks (and hence recovery actions) to be inserted at a level of abstraction that is visible to programmers, and it requires programmers to be allowed to insert (prescribed) CFI safety-checks at their own discretion.

3. **Transparent to software development:** Some current CFI techniques are not transparent to the software development process, and require modified tools in the software compilation process. These can range from additional tools such as binary-rewriters that manipulate post-compilation program binaries [15, 16, 25, 18], to modified compilers that compile programs with different behaviors such as shadow stacks [18]. The use of modified tools in the software compilation process can present problems in certain application domains, such as in safety-critical systems, which can have stringent software standards. For instance, the DO-178C standard for software in flight-control systems requires that non-standard compilers and other software tools undergo tool qualification to ensure that the generated software does not have unexpected behavior [27]. In this dissertation, we would like to develop an approach that

works together with existing software development tool-chains such as compilers, without requiring any modification to existing tool-chains.

4. **Support for automated proofs:** Some CFI techniques currently perform some level of verification of the CFI of their target programs [16, 18, 17]. While some techniques are able to automatically verify the CFI of their target programs [16, 18], their automated verification relies on automatically-inserted safety-checks being in machine-code. However, as we have argued, automatically inserting safety-checks presents challenges for customizing safety-checks for embedded systems. Hence, a different verification approach that does not assume the presence of automatically-inserted safety-checks is required for CFI achieved using safety-checks that are customizable (e.g., at the source-code level).

5. **Support for realistic features in embedded applications:** Most current CFI techniques are aimed at desktop applications [15, 17] and device drivers [16]. On the other hand, CFI techniques for embedded software mainly support programs running bare-metal on processors without an operating system (OS) [18]. However, increasingly, as embedded processors become cheaper and more powerful, embedded systems are running full-fledged operating systems, with applications implemented as user-mode programs running in an OS (as described in [5]). Hence, it is desirable for CFI to be achieved in a way that supports user-mode embedded programs, which is also amenable to realistic programs features, such as the presence of system-calls.

1.3 Thesis Statement

This dissertation explores the following hypothesis:

> *We can provide CFI in a provable, preventative, and programmer-visible manner through a combination of automated prescriptions to enable programmers to remedy potential CFI violations at the source-code level, along with automatic proofs of CFI at the machine-code level.*

Specifically, this dissertation presents a two-part approach to provide enforcement for Control-Flow Integrity that is amenable to software in embedded systems. The first part of the approach, the **enforcement approach**, involves prescribing CFI enforcement mechanisms that are amenable

to embedded software, and allowing programmers to insert these enforcement mechanisms in their programs at their own discretion. These CFI enforcement mechanisms: (i) prevent the root-causes CFI violations from occurring, and (ii) enable programmers to customize the actions to take in the event of a detected (and prevented) CFI violation. The second part of our approach, the **logic approach**, involves enabling fully-automated proofs of CFI in a trustworthy formal logic that does not require any manual or specialized inputs (e.g., code annotations, loop invariants) from programmers. The second part of our approach supports automated proofs of CFI in programs whose CFI enforcement is provided using the first part of our approach. By separating our enforcement approach for providing CFI enforcement mechanisms, and our logic approach for proving CFI, we provide programmers with the opportunity to inspect and modify our prescribed enforcement mechanisms to provide customized recovery actions from prevented CFI violations, while still providing automatically generated formal proofs of CFI for such programmer-customizable enforcement mechanisms.

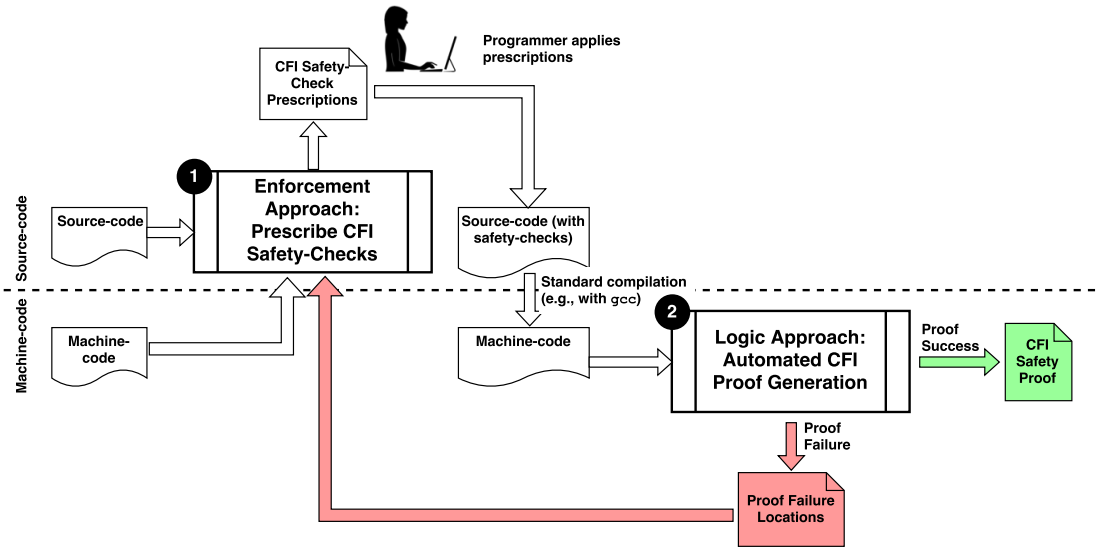


Figure 1.2. Overview of approach to CFI in this dissertation. Numbered boxes represent the parts of our approach, and we show the inputs and outputs of each step in our approach.

Figure 1.2 presents an overview of the approach in this dissertation, and we highlight the inputs and outputs of our two-part approach to CFI enforcement.

The first part of our approach, the **enforcement approach**, helps programmers add CFI enforcements to the source-code of their programs to ensure its CFI, while at the same time enabling the CFI of the resulting program to be automatically provable using the second part of our approach, the logic approach.

First, programmers compile their source-code to obtain the machine-code of their program, and

they supply both the source-code and machine-code of their program as inputs. We then use the machine-code of the program to locate sources of potential CFI violations. Next, we construct prescriptions of safety-checks for programmers. Each prescription consists of source-code statements to use as a safety-check, and the location of a potential CFI violation in the source-code of the program. Programmers apply the prescription to their program by inserting the prescribed safety-check at the prescribed source-code location in their program.

Our enforcement approach uses source-code statements for CFI safety-checks, so that programmers can customize the safety-checks to add recovery actions that are appropriate for their application. We prescribe, rather than automatically insert, our source-code safety-checks, so as to leave it to programmers to insert these safety-checks. This is to allow programmers to ensure that the inserted safety-checks do not change the functionality of their programs, and to allow them to customize the recovery actions in the safety-checks.

The second part of our approach, the **logic approach**, automatically generates formal proofs of CFI for the machine-code of programs that contain source-code CFI safety-checks as prescribed in our enforcement approach. The logic approach expects as its input the machine-code of the target program for which programmers want a formal proof stating that the program's CFI cannot be violated. The logic approach does not need any further inputs (e.g., code annotations, loop invariants) for its proof generation to work. In addition, the logic approach can work with machine-code that has been compiled using standard, unmodified, off-the-shelf compilers, such as `gcc`. Then, where possible, the logic approach automatically generates a formal proof stating that the target program is indeed safe with respect to CFI. However, the logic approach may fail to automatically generate a safety proof when safety-checks in the target program are insufficient.

Then, if the safety-checks in the target program are insufficient (e.g., inadequate or incorrect safety-checks inserted by the programmer), the logic approach is able to terminate the proof attempt (as opposed to the automated proof process running without termination). Then, the output from the logic approach is supplied to the enforcement approach, and the enforcement approach provides feedback to the programmer, in the form of prescriptions of CFI safety-checks, to help him/her rectify the CFI safety-checks in his/her program.

Goals. The primary goals of this dissertation are:

- The prescription of source-code safety-checks for ensuring CFI in programs, that: (i) prevent the root-causes of CFI violations, and (ii) are amenable to customization by programmers.

- The automated generation of formal proofs of CFI for the machine-code of programs that: (i) have been compiled using a standard, unmodified, off-the-shelf compiler (e.g., `gcc`), and (ii) may contain system call invocations.

Scope. This dissertation carries out the above goals with respect to the following scope:

- Prescribed safety-checks: The enforcement approach supports prescribing safety-checks for C programs.
- Formal proofs of CFI: The logic approach supports formal proofs of CFI in machine-code targeted at the ARM architecture. Specifically, the logic approach supports ARM instructions for the ARMv6 architecture in this dissertation.
- Program type: The logic approach supports user-mode programs running in a full-fledged operating system, specifically Linux, in this dissertation.

Assumptions. This dissertation makes the following assumptions about our target programs:

- Target programs are statically compiled and linked.
- Target programs obey the ARM-Thumb Procedure Call Standard (ATPCS) [28] for function call and return behavior.
- Target programs are compiled with well-known function prologues and epilogues, and the function boundaries in machine-code are available.
- The Control-Flow Graph (CFG) of the machine-code of our target programs are correctly computed.
- The underlying OS (Linux) "correctly" services system call invocations by correctly restoring the context (program counter, register, and memory contents) of the user-process at the end of every system call invocation, and providing its documented and specified (e.g., in Section 2 of the Linux Programmer's Manual [29]) functionality.
- System calls are invoked via an assembly wrapper with a C function prototype that sets up the arguments for the system call, and whose name identifies the invoked system call.

Non-goals. This dissertation does not address the following:

- The automatic insertion of safety-checks into the source-code of target programs.

- The prescription of recovery actions for potential CFI violations for target programs.
- The automatic formal proofs of CFI for target programs containing safety-checks not prescribed by our approach.
- The automatic generation of formal proofs for safety properties other than CFI.
- The application-level security and privacy of target programs.
- The functional correctness of target programs.
- The verification of the correctness of the underlying operating system in servicing system calls.
- The indirect effects of system call invocations that are not visible to user-mode programs (e.g., file descriptor mapping, user-space memory mappings in the `mmap` system call).

Threat model. The goal of this dissertation is to protect user-mode programs from external inputs received from external sources, such as network inputs, file inputs, and user inputs. As such, we assume that the underlying hardware, firmware (e.g., BIOS), virtual machine monitor (if there is one), and operating system kernel are in our Trusted Computing Base (TCB), and are not compromised. As such, we assume the following for our target programs:

- The physical security of the host running the target program is not compromised.
- Target programs run in an operating system which provides inter-process memory isolation, i.e., no other process is allowed to modify the contents of the target program's memory at run-time.

We model our attacker as an external user who is able to supply arbitrary inputs to our target programs. We call this an "external input attacker". Our attacker does not have physical access to the hardware, and the attacker **does not** have the ability to do the following:

- Modify the OS kernel or run code in kernel space.
- Execute a process in parallel with the target program.
- Execute a thread in the process space of the target program.

Note that our threat model differs from the typical CFI threat model [15] of allowing the attacker to execute a thread in the process space of the target program, while being allowed to write only to writable memory pages. While our "external input attacker" is weaker than the attacker in classical CFI [15], we believe our attacker model is realistic for a "defense-in-depth" system that has deployed multiple complementary layers of security measures. We discuss some possible complementary measures in such a "defense-in-depth" approach to software security in §8.2.

Residual threats. The approach in this dissertation does not consider non-control-data attacks [30], such as the overwriting of data items in the stack or heap of the target program. This may allow attackers to overwrite data items in memory that may then have direct effects on the subsequent control-flow of the program (e.g., logic attacks), and may even allow attackers to overwrite arguments passed to system calls. We discuss methods to mitigate such residual threats in §8.

1.4 Thesis Map

This dissertation explores the provision of CFI enforcement mechanisms in a preventative, customizable, and provable manner, that is suitable for software in embedded systems that may be safety-critical. We apply our approach for providing CFI to three classes of target programs: (i) benchmark programs, to demonstrate that our approach works for a range of program behaviors; (ii) system utilities, to demonstrate that our approach works for general file-based input/output behaviors; and (iii) programs with hardware inputs and outputs, to demonstrate that our approach works for embedded systems that may interact with hardware.

We begin by surveying existing approaches to providing CFI for software to understand the aspects where each approach may be unsuitable for use with software in embedded systems. Chapter 2 discusses related work in techniques for CFI, as well as related techniques in the verification of safety and security properties for software, the verification of other properties such as functional correctness for software, and more general techniques for developing programs that are safe.

Chapter 3 describes our enforcement approach for providing CFI enforcement mechanisms that are preventative and customizable. We describe our approach to prescribing safety-checks for enforcing CFI for the source-code of our target programs by using heuristics to identify machine-code locations where CFI may be violated. We also describe how to construct these CFI safety-checks, and how the safety-checks can be customized.

Chapter 4 presents our logic approach, and its novel program logic for automatically proving CFI safety in the machine-code of programs whose CFI is enforced using the safety-checks prescribed by our logic approach. We begin by describing the existing Hoare logic framework used to reason about ARM machine-code [31, 32] in this dissertation. Then, we describe how we extend this existing trustworthy Hoare logic for to enable automated reasoning and proof-construction for CFI. Specifically, we develop new proof rules to enable "local reasoning" about safety-assertions which ensure CFI safety. These proof rules specify CFI safety at the instruction-level, and extend this to the basic block and function level, to enable automated reasoning about CFI safety at the whole-program level.

Chapter 5 describes our proof-automation algorithms for constructing the proof of CFI of a machine-code program in our logic approach. We describe our proof tactics which enable proof-automation for machine-code programs containing source-code safety-checks, and our abstract interpretation-based algorithm for automatically discovering and discharging CFI safety-proof obligations. We also describe optimizations which improve the scalability of the proof-automation process.

Chapter 6 describes extensions to our proof-construction and proof-automation algorithms in our logic approach to support realistic features in embedded programs. Specifically, we describe how we reason about system call invocations in Hoare logic for safety proofs, and we describe how we automate safety-property proofs for machine-code containing system call invocations. We also present optimizations to our proof-automation algorithms to support proof generation at scale for larger and more complex programs, and we also evaluate the extent to which our proof-automation algorithms support compiler-optimized machine-code programs.

Chapter 7 describes our experimental evaluation of our approach to CFI, and presents experimental results. We describe the process of applying our prescribed source-code CFI safety-checks to our target programs, and we evaluate our target programs for (i) the run-time overheads of our prescribed CFI safety-checks, and (ii) the times taken by our approach to automatically generate CFI safety-proofs.

Chapter 8 discusses some of the limitations of our approach in more detail. Finally, Chapter 9 discusses future directions for our work, and concludes this dissertation.

1.5 Contributions

This dissertation presents two tools for providing Control-Flow Integrity enforcement for embedded software in a way that is preventative, customizable, and provable. The enforcement approach is realized in the PCFIRE-C tool that we have built, which provides prescriptions of CFI safety-checks for C programs, and the logic approach is realized in the AUSPICE framework that we have built, which automatically generates proofs of CFI for ARM machine-code programs in a formal logic. The contributions of this dissertation are:

Control-Flow Integrity can be enforced in a preventative way by using safety-checks around potentially dangerous memory-write source-code statements. The root-causes of CFI violations are writes to memory that may overwrite dynamic-jump targets in memory, such as saved function-return addresses. Thus, to prevent the root-causes of CFI violations, safety-checks are needed around memory-write statements. Some source-code statements which write to memory can be statically determined to not violate CFI, such as writes to local variables. The remaining source-code statements, whose memory-write targets are computed at run-time, will need to be surrounded by safety-checks. In this dissertation, we describe how to identify such dangerous source-code statements via a superficial inspection of the machine-code corresponding to the source-code. We then describe how to construct prescriptions of safety-checks for such statements.

Control-Flow Integrity can be enforced in a customizable way when preventative safety-checks are provided using source-code statements. Recovery from potential CFI violations is possible only when the CFI violation has been prevented. In this dissertation, we show that when CFI violations have been prevented, recovery actions can be made available to programmers to customize by providing preventative CFI safety-checks using source-code statements. Our prescribed CFI safety-checks are visible to programmers, as programmers are required to explicitly insert our prescribed safety-checks into the source-code of their programs. This empowers programmers by enabling them to ensure that the behavior of their software is not inadvertently changed by the inserted safety-checks. In addition, our prescribed CFI safety-checks are amenable to customized recovery actions, and we show how programmers can modify our prescribed safety-checks to introduce custom recovery actions for their applications. Like our work, ARMor [18] proposed SFI safety-checks that, like our CFI safety-checks, are preventative in nature. ARMor inserts safety-checks at memory-writes instead of at indirect jumps. However, as ARMor's safety-checks are inserted automatically at the machine-code level, they are not easily customizable by programmers

since they are not visible to programmers at the level of abstraction that they work with (i.e., at the source-code level).

Proofs of Control-Flow Integrity at the machine-code level can be automated using proof rules for "local reasoning" to encode proof obligations to discharge for CFI to hold, and using an Abstract Interpretation based analysis to discharge these proof obligations automatically. In this dissertation, we develop proof rules that encode CFI safety in a way that is amenable to proof automation. These proof rules use the "local reasoning" principle to build a bottom-up specification of CFI safety. The proof rules begin by specifying CFI safety at the instruction level, and then expand this specification to the basic block and whole-program levels. These proof rules specify the CFI safety-assertions which must be proved for each instruction, and specify how these safety assertions can be discharged. Then, we develop an abstract interpretation-based algorithm that automatically discovers program properties that can discharge these safety assertions, or identifies safety assertions that cannot be automatically discharged.

While prior work has automated safety proof generation for CFI and other similar safety properties, they have assumed the presence of sufficient safety-checks in their target programs, and their analysis may not terminate for unsafe programs [18, 16]. Our analysis improves upon prior work [18] by allowing for and detecting safety assertions that cannot be discharged, and terminating the analysis. Our key advancements over prior work are in: (i) novel proof rules that build directly on a trustworthy Hoare logic for ARM machine-code, and (ii) a novel Abstract Interpretation algorithm which terminates on potentially-failing proofs, and which is more efficient, improving the scalability of our proof-automation over prior work [18].

Proofs of Control-Flow Integrity at the machine-code level for programs containing system calls can be automated through a combination of axiomatizing system call results from the operating system, and instantiating system call arguments at invocation sites. In this dissertation, we develop an approach to automating safety-property proofs for machine-code containing system call invocations. We enable reasoning about system call invocations in user-mode programs by constructing axioms of the operating system's behavior in servicing a system call, which we explicitly instantiate in our logic approach as undischarged hypotheses of our safety theorems. Then, our proof automation accommodates the analysis of system calls by concretizing the arguments at each system call invocation site by instantiating axioms with the arguments to the system call.

Demonstration of the effectiveness of our approach. We evaluate the effectiveness of our approach for providing preventative, customizable, and provable CFI for embedded software, using

three corpora of software. First, we used benchmark programs from the MiBench [33] and WCET Benchmarks [34] to demonstrate the applicability of our approach on generic software programs with different types of control structures and programming language features. Second, we used simple versions of common systems utilities that we implemented to demonstrate the applicability of our approach to programs with file inputs/outputs. Third, we used programs written for the Raspberry Pi single-board-computer that contain hardware inputs, such as light sensors and accelerometers, and hardware outputs, such as LEDs and LCDs. We demonstrate the ability of our approach to prescribe safety-checks for our target programs. We also evaluated our approach for: (i) the time taken to generate a CFI safety-proof for each of our target programs, and (ii) the run-time and space overheads introduced by the source-code safety-checks prescribed by our approach.

1.6 Novelty of Our Approach

To the best of our knowledge, our approach to Control-Flow Integrity (CFI) is the first that separates the levels of abstraction at which we: (i) provide mechanisms for enforcing CFI, in the source-code of our target programs, and (ii) prove that CFI holds, in the machine-code of our target programs. As a result, our approach to CFI is the first that provides programmer-visible mechanisms for enforcing CFI, that programmers can ensure do not inadvertently change the behavior of their programs, while still providing automatically-generated CFI proofs that do not require manual user inputs, that do not require a trusted compiler, and that are not affected by mis-compilation bugs, such as those described by Yang et al. [35].

Most CFI techniques provide enforcement mechanisms that are automatically-inserted into the machine-code of their target programs [15, 16, 18, 25, 26]. Such mechanisms are not programmer-visible, and may inadvertently change the behavior of target programs. CFI techniques that provide automatic proofs also make use of enforcement mechanisms inserted in machine-code [18, 16, 17]. Techniques that provide safe software behavior in the source-code of programs include Ivory [36], which provides a domain-specific language from which safe C programs can be generated, and CCured [37], which retrofits existing C/C++ programs to insert safety-checks to ensure memory safety. Both Ivovy and CCured require either a trusted or a verified compiler to not be affected by miscompilation bugs. In addition, Ivory requires programmers to learn a new programming language and does not handle system calls, unlike our approach, while CCured inserts safety-checks at scale, and may inadvertently change the behavior of their target programs. CompCert [21] is a

verified compiler which has been proven to correctly compile well-formed C programs faithfully to machine-code that behaves as programmers would expect from the behavior of the program's source-code. However, CompCert is unable to identify ill-defined programs, and programmers will not be alerted to when an ill-defined C program is being compiled.

Thus, to the best of our knowledge, our approach to CFI is the first that combines the programmer-visibility of source-code enforcement mechanisms for CFI, allowing programmers to customize them and observe that the behavior of the software is not inadvertently changed, while still providing machine-code proofs of CFI that can be automated, and that does not require a trusted or verified compiler to ensure its proven properties hold in machine-code.

1.7 Limitations

Our approach to preventative, customizable, and provable CFI enforcement is aimed at addressing the challenges of providing CFI enforcement for embedded software. As a result, we make a number of design trade-offs in our CFI approach to address challenges specific to embedded software.

First, in our enforcement approach, we focus on the preventative and customizable nature of CFI safety-checks, and we do not specifically focus on the run-time overhead of our prescribed safety-checks. In particular, in Chapter 7, we show that preventative safety-checks intrinsically incur higher overheads than current CFI techniques that detect CFI violations after-the-fact, and the higher run-time overheads of preventative CFI safety-checks relative to current CFI techniques is fundamental.

In addition, the proofs of CFI in our logic approach is built on a trustworthy formalization of the ARM instruction set architecture (ISA) [31, 32] in the logic of the HOL4 proof assistant [38]. As a result, our proofs are limited by the features of the ARM ISA that are modeled, and our approach does not model the following features of program behavior in target programs due to limitations in the underlying logic we use to formalize ARM instructions:

- Hardware interrupts, hardware exceptions, and page table operations,
- Multi-threaded, concurrent behavior,
- Floating point instructions.

Our logic approach also currently does not address the following features in our target programs:

- Conditionally-executed non-branching instructions,

- Unstructured control-flow jumps, i.e., unstructured `gotos` (e.g., Duff's Device), nor `longjmp` statements,
- Recursive function calls,
- Explicit function pointers,
- Compiler optimizations (i.e., other than `-O0` optimization level for `gcc`).

We currently do not address conditionally-executed non-branching instructions as they induce multi-graphs in the Control-Flow Graphs of target programs, although this can be handled simply by "unconditionalizing" programs, or by constructing a non-address-based CFG representation of the program. We do not handle the unstructured use of `gotos` (i.e., when they result in unstructured jumps) nor the use of `longjmp` statements in our target programs, although we do not believe this is a significant limitation on the applicability of our approach to most embedded programs, as such program features are discouraged for safety-critical software. We do not support recursive function calls, as this would require the manual specification of inductive invariants by programmers, which violates our goal of fully automating CFI proofs, or they would require the automatic inference of inductive invariants, which is a challenging problem in itself. We currently also do not fully support compiler-optimized programs, and we believe this is also not a significant limitation, especially for safety-critical embedded software development, for which a number of safety standards (e.g., the DO-178C [27] for avionics software) require additional tool qualification or validation activities for using compiler optimizations.

Finally, we have focused on programs compiled using `gcc` 4.6.3 on the ARM platform in this dissertation. We have not explored if our approach is able to work with programs compiled using other compilers such as Clang, although we believe that our approach can be adapted to programs compiled using other compilers with minor modifications, as long as the information required for our safety property proofs (e.g., availability of the frame pointer register used to indicate the base address of a function's stack frame) is available in the compiled programs.

Chapter 2

Related Work

This dissertation develops an approach to Control-Flow Integrity (CFI) enforcement for software that is preventative, customizable, and provable, especially for embedded software that may be safety-critical. In this chapter, we will discuss the mechanisms used by other techniques to achieve CFI and other similar safety and security properties, and how current techniques verify CFI and other similar safety and security properties in software.

We begin by discussing early techniques for addressing security vulnerabilities such as buffer overflows, which are one of the main ways that CFI can be violated in software (§2.1). Next, we describe the context behind CFI, and a closely related safety property, Software Fault Isolation (SFI) (§2.2), and we discuss prior techniques for ensuring CFI in software (§2.3). Then, we discuss alternative approaches to achieving safety properties similar to CFI using safe languages, such as safe dialects of existing languages such as C and type-safe assembly languages (§2.4). We compare these approaches to the approach in this dissertation for the differences in their enforcement and verification (if any) of their achieved properties, the security guarantees achieved, and the tradeoffs made in achieving their properties. Next, we discuss techniques for verifying CFI in programs using Interactive Theorem Proving, which are closest to the verification approach in this dissertation (§2.5.1), before we discuss applications of Interactive Theorem Proving to the verification of other properties (§2.5.2). We then discuss the use of Software Model Checking, an orthogonal approach to verification, for verifying properties in the machine-code of programs (§2.5.4). We also review another class of approaches to verification, of using verified analyses that have been built on top of verified compilers (§2.5.5). Finally, we review recent work on attacks on existing techniques for CFI, and we qualitatively discuss their applicability to our approach (§2.6).

2.1 Buffer Overflow Attacks and Protection

Buffer overflow attacks [39] refer to the broad class of security vulnerabilities where attackers overwrite buffers in a program's memory to gain control of the execution of the program. Attackers may inject attack code, which they then execute, or they may simply change the sequence of instructions that are executed in the program (as compared to the original, programmer-intended sequence of instructions). Return-Oriented Programming (ROP) [40] is a more advanced variant of the basic buffer overflow attack, in which an attacker changes a program's behavior without injecting any attack code. Buffer overflows are closely related to Control-Flow Integrity, because when a buffer overflow attack succeeds, a CFI violation has occurred. As such, we examine some of the techniques that have been introduced to protect against buffer overflow attacks, and we discuss the suitability of such protections for embedded software.

Cowan et al. [39] identified two key ingredients needed for a successful buffer overflow attack: (i) the attacker needs to introduce the attack code that he/she wishes to run in the context of the target program, and (ii) the attacker needs to cause the program's execution to jump to the introduced attack code. Without the ability to directly overwrite code memory (i.e., the program's code memory is protected from being overwritten, e.g., through code page read-only flags), attackers can only change the program's execution by changing *code pointers*, which are present in: (i) function activation records (i.e., the return address of each function, as saved to the function's stack), (ii) function pointers in C (e.g., "`void (* f) ()`" declares a function pointer variable f which points to a function that takes no arguments and has a return type of `void`), and (iii) `longjmp` buffers in C. Then, the basic workflow of a buffer overflow attack will involve: (i) injecting attack code to memory, and (ii) overwriting the chosen code pointer to cause execution to jump to the injected attack code. This requires: (i) the buffer being overflowed to be sufficiently close to the target code pointer in memory, and (ii) the program to have weak or non-existent bounds checks on the chosen buffer being overflowed, so that the buffer overflows onto the code pointer, allowing the attacker to write arbitrary data into the code pointer.

The basic form of the buffer overflow attack is the stack smashing attack [41], in which function return addresses saved to the stack are overwritten. More advanced attacks include pointer subterfuge attacks (overwriting of function pointers, data pointers that are aliases of function pointers, exception handler addresses, and C++ virtual function table entries) and heap smashing attacks which corrupt the metadata in memory allocators to enable buffer overflows [42].

Techniques to prevent or minimize the damage from buffer overflow attacks have addressed one or both of the key ingredients for a buffer overflow attack, of preventing the execution of code injected into data memory, and detecting overwritten code pointers.

2.1.1 Data Execution Prevention (DEP)

One way to prevent attackers from running code that has been injected, is to prevent the execution of any instructions that are found in the data segment of a program's address space. Modern processors enable memory pages to be marked "no-execute", such as using the XD (eXecute Disable) bit on Intel processors, or the XN (eXecute Never) bit on ARM processors [43]. This then prevents the processor's execution from jumping to addresses in memory pages containing data (e.g., the program's stack or heap), where attackers may have introduced attack code. Operating systems can then make use of such XN or XD bits to mark the memory pages holding a program's heap and stack as non-executable. However, this does not prevent ROP attacks where attackers change the order of instructions executed in the program without injecting any attack code. DEP also does not prevent code pointers from being overwritten and corrupted in the first place, and can still cause programs to crash, e.g., when a code pointer is overwritten with an invalid address.

2.1.2 Canary-based Methods

Another way to mitigate buffer overflow attacks is to detect when code pointers have been overwritten by attackers. Canary-based methods, such as StackGuard [44], use instructions inserted into a program at compile-time to place a secret "canary" value in memory adjacent to sensitive locations, such as those containing code pointers. They assume that attackers write to memory in sequence, so that they can overwrite code pointers only by overwriting the memory location immediately before the location where the code pointer is located. Then, prior to the value in the code pointer being used to perform a jump, the secret canary value is checked. If the canary value has been corrupted, the code pointer is deemed to have also been overwritten, and the program is halted. However, canary-based methods detect buffer overflows only after-the-fact, and do not allow for recovery after an attack has occurred. In addition, canary-based methods can be overcome by directly overwriting code pointers [45].

2.2 Control-Flow Integrity and Software Fault Isolation

Control-Flow Integrity (CFI) was first introduced by Abadi et al. [15], and CFI is related to the weaker safety property of Software Fault Isolation (SFI), which was proposed by Wahbe et al. [46]. While CFI and SFI have different goals, similar techniques have been used to enforce as well as verify both CFI and SFI in software.

The main goal of SFI is to enable untrusted code to run in the same address space as trusted code so as to improve the performance of software with untrusted modules, while still ensuring the security of the trusted code. SFI implies that untrusted code, when executed in the same address space as trusted code, cannot overwrite the data or modify the execution of trusted code. One example where SFI is useful is when a trusted application wishes to host and run untrusted third-party plugins, such as kernel modules in an operating system and third-party plugins and extensions for web browsers.

On the other hand, the main goal of CFI is to protect a program's execution from being modified and hijacked by potentially malicious external inputs. CFI implies that the execution of a program does not deviate from the programmer's intentions, as captured in a predetermined CFG of the program. CFI, when enforced, can rule out control-flow hijacking attacks, such as those caused by buffer overflows.

While SFI and CFI are closely related, SFI's threat model protects trusted code from untrusted code that executes in the same address space, whereas CFI's threat model protects all code of a program from external inputs (e.g., user, network, or file inputs). CFI can be used to strengthen SFI implementations by preventing SFI enforcement mechanisms from being circumvented [15].

2.3 Mechanisms for Control-Flow Integrity

Next, we discuss the different techniques that have been proposed for enforcing CFI in software, and compare them to our approach. We begin by discussing mechanisms for CFI that are introduced statically in software by modifying or rewriting programs before they are executed (§2.3.1). Then, we discuss dynamic mechanisms for CFI that operate at runtime (§2.3.2), before we discuss other techniques that have been proposed to specifically mitigate Return-Oriented Programming (ROP) attacks against CFI defenses, and hardware mechanisms for enforcing CFI (§2.3.3).

2.3.1 Statically-introduced Mechanisms for CFI

The main source of potential CFI violations is from indirect jumps, which arise in (i) function calls using explicit function pointers, (ii) function returns, and (iii) the use of the `longjmp` statement in C. Most contemporary static CFI techniques that adopt the threat model of Abadi et al. [15] focus on the first two sources of CFI violations.

In this dissertation, as our focus is on preventative CFI that is also provable in a fully-automatic manner, we have chosen to focus on CFI for function returns from directly-called functions, while we leave CFI for explicit function pointers to future work. We discuss extensions to our approach to support explicit function pointers in §9.1.1. Nonetheless, we discuss contemporary techniques for enforcing CFI, most of which focus on enforcing CFI for indirect jumps made using explicit function pointers, and contrast them to our approach.

Classic CFI

Next, we discuss the classic CFI enforcement developed by Abadi et al. in their seminal work [15]. First, they use static-analysis to compute a CFG for the program, which identifies allowed jump targets for each instruction. Then, CFI requires that only jump targets present in this computed CFG are allowed. For direct jumps (e.g., conditional branches, loops, function calls), a static verifier is able to check that each jump is allowed. For indirect jumps, the program is instrumented with runtime checks to ensure that only jumps present in the computed CFG are allowed. The program is aborted on any other jump whose target is not present in the computed CFG.

The instrumentation for CFI enforcement is as follows. For indirect jumps with explicit function pointers, their jump targets are partitioned into equivalence classes: two target addresses are equivalent if the CFG allows jumps from one indirect branch to the two addresses. Then, each equivalence class is assigned a unique identifier, which the program is instrumented with at each jump target. Next, each indirect jump is instrumented with a safety-check that allows the jump only if its jump target contains the identifier associated with it. For indirect jumps that are function returns, the check for validity of the return target is provided by using a shadow call stack that is protected (using memory page permissions) from being overwritten.

The instrumentation in classic CFI [15] detects CFI violations when function pointers and function return addresses do not contain the unique identifiers that the program was instrumented with. This precludes the possibility of preventing the CFI violation in the first place to enable recovery

actions, as CFI violations are detected only after function pointers and function return addresses have been modified, whereas our approach enables recovery by preventing the root-causes of CFI violations.

Other CFI Techniques

Next, we review the other CFI techniques that have been proposed since classic CFI [15, 16, 17, 48, 49, 18, 50, 26, 51, 52, 53, 54, 25, 55, 56, 57, 58, 59, 60], and we summarize them in Table 2.1, and contrast their approach to CFI to ours.

We briefly summarize the reviewed techniques. First, we include a number of related techniques that provide CFI as a side-effect. SFI techniques [16, 17, 18] provide enforcement for SFI as well as CFI, and they use CFI to protect the execution integrity of their introduced SFI mechanisms. In earlier, more general work, Inlined Reference Monitors (IRM) [63] were proposed as a means to enforce security policies using binary-rewriting to introduce instrumentation to enforce security policies expressed as finite-state machines. WIT [48] provides memory safety by preventing dangerous memory writes, while CPI [57] prevents dangerous writes to code pointers, both of which are similar to our approach of preventing dangerous memory writes, although neither technique allows for customizable recovery nor generation of safety proofs. NaCl [49] provides coarse-grained sandboxing for untrusted browser plugins, and Zeng et al. [26] uses CFI to achieve a more efficient data sandbox.

Second, many proposed CFI techniques tackle challenges such as improving performance, and supporting separate compilation (independent compilation of different modules) and modularity (e.g., use of dynamically loaded shared libraries). Control-Flow Locking [50] provides a lightweight CFI enforcement scheme to improve performance. CCFIR [25] and bin-CFI [54] both provide practical CFI, with CCFIR providing a lightweight 3-ID scheme, with 3 equivalence classes for indirect jump targets, and bin-CFI providing CFI for COTS binaries. Niu et al. provide various techniques for facilitating CFI with separate compilation [53] and modularity [55]. Forward-edge CFI [58] provides practical CFI that is implemented for LLVM and GCC. More recent CFI techniques have focused on providing finer-grained CFI to thwart gadget construction in Return-Oriented Programming (ROP) attacks [40]. PiCFI [61] provides per-input CFI, to constrain the allowed CFG in the program to those made possible by inputs, to prevent attackers from using disused parts of code as ROP gadgets. TypeArmor [62] has a similar goal, and prevents Counterfeit Object-Oriented Pro-

gramming (COOP) attacks [64] which chain calls to existing functions through legitimate call-sites, and they achieve this by matching call-site signatures to return-sites.

Third, specialized CFI techniques have been developed that use different enforcement schemes, or support different platforms, such as smartphones. KCoFI provides CFI for OS kernels [56]. Cryptographic CFI [59] use message authentication codes to help detect illicitly modified indirect jump targets. Opaque CFI [60] provides a randomized CFI enforcement scheme to defeat Return-Oriented Programming (ROP) attacks. MoCFI [51] and Pewny et al. [52] provide CFI for smartphone apps on the Android and iOS platforms respectively and handle the limitations of working on such platforms.

Comparison with other CFI techniques

We classify these CFI techniques by: (i) the location of their instrumentation, (ii) their enforcement behavior when a CFI violation is detected, (iii) the granularity of their enforcement, (iv) how they protect against function return hijacking, and (v) how (if any) their technique was verified. We first focus on CFI techniques that use enforcement mechanisms that are entirely in the target program, without the need for hardware or operating system assistance.

First, we found that all reviewed CFI techniques insert enforcement mechanisms in either the machine-code, assembly code, or compiler intermediate representation (IR) of the program [15, 16, 17, 48, 49, 18, 50, 26, 51, 52, 53, 54, 25, 55, 56, 57, 58, 59, 60, 61, 62]. These levels of abstractions are not accessible to programmers, making it very challenging for programmers to modify the instrumentation to provide customized recovery actions appropriate for their applications. Second, the reviewed CFI techniques either abort when CFI violations are detected [15, 16, 48, 18, 26, 51, 52, 53, 54, 25, 55, 56, 57, 58, 59, 60, 61, 62], or they clobber illegal jump targets to force them to a legal address [17, 49, 50], both of which will give rise to undefined behavior, and are undesirable for embedded software that may be safety-critical.

In contrast, our approach to CFI is preventative: in preventing root-causes of CFI violations, our approach is not forced to abort when CFI is violated; in addition, our approach prescribes enforcement mechanisms at the source-code level, which then allows programmers to add their own recovery actions in our prescribed source-code mechanisms to provide recovery from prevented CFI violations. This makes our approach amenable to embedded software that may be safety-critical, for which aborting is unsafe.

Third, the reviewed CFI techniques varied in their level of CFI granularity, which refers to the number of equivalence classes of indirect jump targets supported for jumps using explicit function pointers. As our approach does not support explicit function pointers, this metric is not applicable to our approach.

The highest level of granularity is *Fully-Precise Static CFI* [30], which allows an indirect control-flow transfer along an CFG edge only if "there is a non-malicious trace that follows that edge" [30]. At the lowest level of precision, PittSFIeld [17] and NaCl [49] only require that indirect jump targets have memory-aligned addresses, so as to prevent jumps into the middle of variable-width x86 instructions. Abadi et al. [15] proposed that a coarse, "1-ID" CFI implementation can be obtained by using a single identifier for all indirectly-called functions (and their return-sites). This prevents attackers from hijacking execution to jump to the middle of a function, but still allows attackers to change the order in which functions are executed. Other levels of precision include a 2-ID CFI implementation which uses one unique identifier for call-sites, and one unique identifier for return-sites. CCFIR [25] uses a 3-ID CFI implementation to separate returns to sensitive and non-sensitive functions. Then, Fully-Precise Static CFI is effectively an N-ID CFI implementation for N distinct indirectly-called functions. However, precision is still lost when an indirectly-called function is called from multiple call-sites, as a return to any of its return-sites (for each call-site) will be valid behavior. Our approach to CFI is fully precise, mainly because we currently do not allow explicit function pointers, which is where precision issues arise.

Fourth, most reviewed CFI techniques do not protect function returns, as they point to earlier techniques such as StackGuard [44] that can protect against buffer-overflow-based attacks against function-return pointers, and also due to the high performance overheads of maintaining a protected shadow call stack to check that function return pointers have not been overwritten. The other techniques that protect function returns using a protected shadow call stack [15, 16, 18, 55] require modified compilers to introduce the shadow call stack, which can be problematic for safety-critical software development which would require tool qualification for modified compilers (§1.2). In contrast, our approach focuses on protecting function returns, and does not require a shadow call stack, as we prevent function return addresses from being overwritten in the first place, although this limits allowed program behaviors (§8.1).

Finally, a small number of the reviewed CFI techniques provide verifiers to ensure that their instrumented programs have sufficient CFI enforcement instrumentation [16, 49, 17, 26], and some have machine-checked proofs of correctness of their verifiers [17]. The classic CFI technique [15]

was formally proved to be safe [47], and the KCoFI and CPI techniques were partially formalized and proved to be correct [56, 57]. ARMor [18] is the only technique that can generate a machine-checkable proof in a theorem prover that states that its target program is safe. Our approach is similar to ARMor, in that we are able to generate a machine-checkable proof in a theorem prover stating that our target program is safe. The main difference is that ARMor's proof generation assumes the presence of automatically-inserted safety-checks, and may not terminate when safety-checks are missing. In contrast, our proof generation can terminate if there are insufficient safety-checks, which is necessary because our CFI approach requires programmers to insert prescribed safety-checks, which may be subject to human error.

We note that while most CFI techniques choose to provide CFI enforcement for explicit function pointers, and relegate the protection of function returns to the use of shadow call stacks, or even point readers to earlier protections (e.g., StackGuard [44]), we have chosen to focus on protecting function returns while disallowing the use of explicit function pointers. This was a deliberate design decision to allow us to focus on providing CFI in a preventative way that is also automatically provable formally, to yield CFI that is amenable to embedded, potentially safety-critical software. We believe we will face the same design decisions for achieving fine-grained yet efficient CFI for explicit function pointers in future work, and we discuss this in greater detail in §9.1.1.

2.3.2 Run-time and Other Mechanisms for CFI

A number of techniques have also been proposed to dynamically enforce CFI at runtime. Program Shepherding [65], Vx32 [66] and Scott et al. [67] all dynamically insert safety-checks in the instructions of running programs to enforce safety properties such as memory safety [65], and to sandbox untrusted code, all of which can imply CFI. PathArmor [68] also dynamically instruments programs to track program paths during execution to ensure a CFI safety policy, and to protect against ROP attacks. PathArmor also makes use of hardware-accelerated features in modern processors for branch recording to achieve efficient fine-grained CFI. SOFIA [69] is a hardware-based technique for enforcing CFI in running programs, and to guard against code-reuse attacks such as ROP.

In general, run-time techniques for enforcing CFI incur high runtime penalties, and require additional software to run, which is unsuitable for embedded systems. In addition, the presence of additional run-time components increases the size of the Trusted Computing Base (TCB) of these techniques, and users need to trust that the run-time components work correctly. While hardware-assisted [68] and hardware-based techniques [69] for CFI may reduce the overheads of CFI en-

forcement, it may not be possible to introduce the required hardware mechanisms on embedded platforms.

2.3.3 Specific Defenses against Return-Oriented Programming

A number of techniques have also been proposed recently to specifically mitigate code-reuse attacks (e.g., Return-Oriented Programming (ROP) [40]) that may be possible due to coarse-grained CFI mechanisms, and due to difficulties in performing precise static-analysis for CFG construction. Most of these techniques specifically address specific behavioral signatures of ROP and code-reuse attacks and are orthogonal to contemporary CFI techniques.

Opaque CFI [60] is a CFI technique that randomizes its enforcement bounds to yield a coarse-grained CFI enforcement mechanism that reduces the probability of code-reuse attacks. ROPGuard [70] is a run-time tool for detecting ROP attacks, and it monitors the call-stack of another running process to detect suspicious uses of the x86 `RET` function return instruction, which is a key ingredient of an ROP attack. ROPecker [71] and kBouncer [72] both make use of the Last Branch Recording (LBR) instruction in modern Intel processors to perform a sanity check on the call stack to detect ROP attacks.

2.4 Safe Languages

CFI can also be achieved indirectly by writing programs in safe languages that provide CFI-like guarantees. First, we review safe dialects of C that are designed to eliminate some of the dangerous behaviors in C that give rise to possible CFI violations, while still retaining the low-level features of C that make it suitable for systems programming (§2.4.1). Next, we discuss techniques to provide memory safety for C that can rule out CFI violations (§2.4.2). Then, we discuss efforts to develop a type-safe assembly language that can be statically proven to not have CFI violations, and to develop domain-specific languages (DSLs) that have been designed to enable programmers to write safe software (§2.4.3).

2.4.1 Safe Dialects of C

CFI and other safety properties can be achieved by using dialects, variants or subsets of the C language, such as Control-C [73] and Cyclone [74]. Control-C formally specifies its static-analyses of memory safety [75, 76], but it imposes a number of restrictions on allowed C behaviors, such as

disallowing pointer arithmetic, and imposing restrictions on the types of array indices allowed for array accesses inside loops, which our approach does not impose as we work directly with machine-code. In addition, the static-analyses of C code in Control-C do not produce a concise safety proof, unlike in our approach.

Cyclone statically analyzes a program's source-code, and inserts run-time safety checks into the compiled program. Cyclone's safety checks are not visible to programmers as they modify post-compilation code, unlike our approach, which uses customizable safety-checks that are left to programmers to insert. In addition, C0 [77] is a safe subset of the C programming language, augmented with contracts, developed for teaching an introductory Computer Science class.

2.4.2 Memory Safety for C

Next, we discuss techniques for achieving memory safety for C, which can provide CFI by ruling out illegal memory writes that are necessary for overwriting sensitive memory locations that can affect CFI.

CCured [37] uses type-inference to identify unsafe memory accesses in C programs that are then protected with automatically-inserted C safety-checks. However, CCured is designed to automatically retrofit existing C code at a large scale, and this retrofitting can be considered to be non-customizable although the safety-checks are in a program's source-code, as it does not give programmers the chance to customize the automatically-inserted safety-checks.

SoftBound [78] is a system for providing spatial memory safety for C programs. SoftBound stores metadata about C pointers and arrays to record bounds information, and automatically inserts safety-checks at the compiler intermediate representation level to ensure that writes to locations addressed by pointers do not overflow and affect other sensitive locations. SoftBound's safety-checks prevent illegal overwrites to memory, just like in our approach. However, their safety checks require additional metadata to be maintained in memory. SoftBound is not suitable for embedded software, similarly to most of our reviewed CFI techniques. This is because the automatically inserted instrumentation aborts the target program when an illegal memory write is detected, and SoftBound's safety-checks are not visible to programmers, and hence are not customizable.

Data-Flow Integrity (DFI) [79] extends the CFI idea to memory writes, to prevent attackers from using buffer overflows to write to unintended locations in memory. In DFI, instead of computing a control-flow graph, a data-flow graph is computed, and a reaching definitions analysis is carried out to identify legal write locations for each location. Then, safety checks are inserted to check if

each memory write is safe according to the computed reaching definitions. Like in our approach, as well as SoftBound [78], DFI prevents illegal writes. However, like SoftBound, as well as most of our reviewed CFI techniques, DFI's instrumentation is inserted automatically in the compiler intermediate representation level, which is not accessible to programmers for customizing recovery actions, and its instrumentation aborts the program on illegal memory writes, which is unsuitable for embedded software.

2.4.3 Other Safe Languages

Typed Assembly Language (TAL) [80, 81, 82] is an approach for obtaining provably safe software. TAL is a type-safe assembly language, which is annotated with types, and the soundness proof of TAL's type system implies that a TAL assembly code program that can be type-checked successfully has memory and control-flow safety, thus ensuring CFI of the resulting program. TAL was conceived as an instantiation of Necula and Lee's Proof Carrying Code [83] system, where untrusted programs can be compiled to TAL, then the annotated types in the TAL program would constitute "hints" to the "proof system", and the validation of the program would involve type-checking the program.

TALx86 [82] is a specific instantiation of the TAL vision for the x86 architecture, and TALx86 is an typed assembly language for the Intel IA-32 architecture. Morrisett et al. developed a type-safe C-like language, Popcorn, which can be compiled to TALx86, and subsequently assembled with the Microsoft Macro Assembler (MASM) to produce IA-32 binaries.

As compared to our approach, our approach works with commodity compilers, whereas TAL requires specialized compilers which can preserve type information during compilation to produce type-annotated assembly language programs. In addition, our approach does not impose *a priori* restrictions on the syntax of C programs, whereas Popcorn in TAL [82] leaves out certain dangerous features of C, such as pointer arithmetic, the address-of operator, and pointer casts. As such, our approach is more easily adapted to existing development workflows for embedded software.

Ivory [36] is an embedded Domain-Specific Language (EDSL) that is embedded in the functional programming language, Haskell. Ivory programs can be compiled to C programs, whose memory-safety is guaranteed by the type-safety of the meta-language, Haskell, that Ivory is embedded in. By embedding the language in Haskell, Ivory is able to leverage the type-safety of Haskell to produce safe C programs. The main strength of Ivory is in the economy of its implementation (6 engineer-months and approximately 6000 lines of Haskell code). However one key weakness is that

programmers need to learn a new language to program in, whereas our approach works with regular C programs. Hickey et al. [36] also developed Tower, a separate EDSL that provides communication facilities for Ivory programs. Ivory and Tower have been demonstrated on generating software for an embedded controller for an autopilot system.

2.5 Verification of Safety and Other Program Properties

Next, we discuss various verification techniques that are related to our approach. First, we discuss recent techniques that have used Interactive Theorem Proving to prove the CFI of target programs and to prove the correctness of CFI verifiers (§2.5.1). Then, we discuss applications of Interactive Theorem Proving to other kinds of general program properties (§2.5.2), and applications of Software Model Checking to verify safety properties for machine-code programs (§2.5.4) Finally, we discuss new directions in applying verified compilation to produce verified analyses of various properties about programs (§2.5.5).

2.5.1 Verification for CFI

Theorem Proving Approaches for CFI

ARMor [18, 84] and XFI [16] both verified that their target programs, which contain their added SFI instrumentation, did indeed possess the SFI safety property [46]. XFI has been applied to device drivers and multimedia codecs for x86 Windows binaries, while ARMor has been applied to embedded ARM software running "bare metal" on ARM processors without an operating system. XFI uses its own implementation of a Hoare logic model of the x86 ISA to perform its verification, while ARMor's verification is proof-producing, and like our approach, is able to produce a safety proof for each target program it is applied to.

XFI developed their own abstraction of the x86 ISA, and is not based on foundational and validated ISA abstractions; their verifier consisted of 3000 lines of C++ code. ARMor verified its SFI isolation using the Cambridge ARM Model [32, 31] in the HOL4 theorem prover [38], which was independently developed from the ARMor project and independently validated, thus increasing the trustworthiness of ARMor's verification. Like ARMor, our approach uses the Cambridge ARM Model to verify safety properties for machine-code programs. However, ARMor's verification relies on a prior binary-rewriting step to insert safety checks and does not handle failures in safety

proofs, whereas our approach, the verification does not require binary-rewriting, and can handle proof failures.

In addition, ARMor developed its own Hoare Logic [84] based on the single-instruction semantics provided by the Cambridge ARM Model [32, 31], whereas the logic used in our approach directly builds on the Hoare Logic for single-entry, single-exit basic blocks in the Cambridge ARM Logic. Thus, the logic used for verification in this dissertation builds on a larger fragment on the Cambridge ARM Logic, and adds less new extensions than ARMor to the Cambridge ARM Model.

RockSalt [85] and PittSFIeld [17] both developed checkers that verify that a given target program does indeed possess their SFI isolation properties after they have been instrumented. This reduces the Trusted Computing Base of their checker, and removes the need to trust the part of their toolchain that adds the instrumentation for isolation. PittSFIeld [17] developed their own formalization of the x86 ISA in the ACL2 theorem prover [86], while RockSalt developed their own formalization of the x86 ISA in the Coq theorem prover [87], which they validated using simulation. Both approaches subsequently proved that their checkers were correct. Unlike our approach, RockSalt and PittSFIeld rely on the safety proof of their checkers to enable trust in their approach, whereas our approach produces a safety proof about each target program.

2.5.2 Theorem Proving Approaches for Other Properties

Bedrock [88] provides "mostly-automated" verification for general program properties such as functional correctness in an idealized machine language. Bedrock does not provide full automation of proofs, but allows users to prove arbitrary properties about programs by allowing/requiring users to provide some manual inputs to the proof process such as code annotations, whereas we provide full proof automation for one class of properties (CFI) without requiring any user input. The Cambridge ARM Model [89] was also originally designed to hide away low-level details about machine-code programs while still allowing users to directly reason (albeit manually and interactively) about the correctness of machine-code programs. The Foundational Proof Carrying Code project [90] uses a richer Hoare Logic than in this dissertation, and hence can reason about unstructured control-flow in machine-code programs. However, the FPCC project also requires the use of a special compiler to emit type information in the compiled assembly programs, whereas our approach proves safety properties for programs compiled using an unmodified commodity compiler.

Recent work has also extended the approach pioneered by Myreen et al. [89, 31, 32] for partially-automating functional correctness proofs for machine-code programs to support system

calls. Goel et al. [91] formalized the x86 Instruction Set Architecture in the ACL2 proof assistant [86], and built an executable model of the x86 ISA. Their approach first models user-mode machine-code behavior in their ACL2 model of the x86 ISA. They then provide two modes for reasoning about the user-mode-observed end-result of system call processing. First, they support simulation of the system call being serviced by the OS by supplying the underlying OS kernel with concrete arguments to the system call, allowing the kernel to execute the system call, and collecting the concrete results for reasoning about at the logical level. Second, they support logical reasoning about the end-results of the system call being serviced by the OS kernel. They demonstrated their approach using a manually constructed proof of correctness for a simple word count application. As compared to our approach to reasoning about system calls, Goel et al.'s approach is more detailed, and features an opaque external environment that captures details such as file descriptor mappings and the underlying filesystem, which our approach does not model. However, our approach fully automates proofs for a specific safety property (CFI) whereas Goel et al.'s construction requires manual interaction with the proof assistant, but is more general.

In addition, CakeML [92] is a compiler for the ML programming language that has been mechanically verified using the Cambridge model [89, 93], and targets multiple architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V.

Certified Assembly Programming (CAP) [94, 95] used Hoare Logic and Separation Logic to build certified libraries manually proven to be functionally correct. In addition, the CAP approach has been extended to build CertiKOS (Certified Kit Operating System) [96, 97], a certified operating system kernel. CertiKOS uses a Proof Carrying Code [83] approach to construct an OS kernel from a clean-slate design, such that the kernel can be proved to have important security properties such as Information-Flow Security [98], and such that important hardware-influenced behaviors such as interrupt-handling [99, 100] can be incorporated into their proofs of functional correctness of the OS kernel. CertiKOS has manually verified various security and functional correctness properties which are important for an OS kernel, albeit generating machine-checkable proofs that can be utilized in a Proof Carrying Code (PCC) framework [83]. Both CertiKOS and our approach make use of proof assistants (Coq in CertiKOS vs. HOL4 in our work). In contrast, our approach focuses on a specific safety property (Control-Flow Integrity) for user-mode code, and aims to automatically generate a proof of CFI for a variety of target programs.

In addition, theorem proving approaches, in particular building on the Cambridge ARM Model [89], have been applied to verify information-flow security for an ARM-based separation kernel

[101], to prove instruction-level isolation properties for the ARMv7 Instruction Set Architecture (ISA) [102], and to formally verify security properties for Direct Memory Access (DMA) at the instruction set and processor implementation levels [103]. Dam et al. [101] formally verified the information-flow security of their PROSPER separation kernel by proving that the observable execution traces for each partition separated by the kernel are the same. Their effort consists of a combination of verifying processor-level behavior, which they performed using HOL4, and kernel-level behavior, which they performed using the BAP binary analysis platform [104] to verify that specific contracts were met by the kernel code. Khakpour et al. [102] proved five information-flow safety properties at the processor level, such as the non-infiltration of neighboring kernel-mode and user-mode processes executing at the same time on a processor, and the non-exfiltration of machine resources, in the ARMv7 ISA. They performed a machine-assisted proof using the HOL4 theorem prover, and extended the Cambridge ARM Model with a formalization of the Memory Management Unit (MMU) in the processor. Schwarz and Dam [103] proved various noninterference-oriented isolation properties by extending the Cambridge ARM Model to support a general device framework to enable reasoning about DMA operations with hardware devices.

A theorem proving approach, which is a form of deductive verification, has also been applied to verify safety properties about physical phenomena for hybrid systems, such as in the KeYmaera theorem prover for hybrid systems [105]. KeYmaera provides a foundation for deductive verification, and supports verification using differential dynamic logic, which enables the specification of and reasoning about physical properties for hybrid systems in a real-valued first-order dynamic logic [106, 107]. KeYmaera is intended for reasoning about and verifying design-level properties about hybrid systems, whose behavior is represented using hybrid automata. In contrast, the logic approach in this dissertation focuses on implementation-level safety properties of software. Nonetheless, the use of deductive verification in KeYmaera to reason about the potentially unbounded state-space of hybrid systems validates our choice of using deductive verification in our approach to verifying and automatically generating proofs of CFI safety properties. KeYmaeraX has also been used in ModelPlex [108] to verify the correct run-time validation of models of cyber-physical systems, where run-time enforcement mechanisms are used to enforce safety-properties, and these enforcement mechanisms are verified in the dynamic logic deductively using KeYmaerax. The enforcement mechanisms in our approach are effectively run-time enforcement mechanisms, and the safety-properties verified in our logic approach can also be similarly achieved using an approach such as in ModelPlex.

2.5.3 Verification of Critical Systems Software

In addition to the CertiKOS [96, 97] and PROSPER [101, 102, 103] efforts, the XHMF project has also developed a verified hypervisor. Vasudevan et al. [109] developed a modular hypervisor, with the goal of enabling security properties to be verified for the hypervisor. They verify the memory-integrity of their hypervisor using the CBMC model-checker for C code [110] to automatically check that manually-inserted assertions for various safety properties are not violated. In more recent work, Vasudevan et al. [111] developed the *ü*berSpark architecture for verifying extensible hypervisors written in both C and assembly code. The verification of their hypervisors is achieved using the Frama-C [112] static-analysis tool for C99 programs, which they use to verify assume-guarantee behavior specifications for their C programs. Vasudevan et al. then implemented a C99 hardware model for the x86 hardware-virtualized platform to enable Frama-C to also verify the assembly code embedded in their C programs. Finally, the *ü*berSpark project makes use of the CompCert [21] certified compiler to ensure the properties they verify at the source-code level carry over to their compiled machine-code.

The seL4 [113] project was the first to formally and mechanically verify that a realistic separation kernel, seL4, was functionally correct, and that its implementation met its high-level specifications. seL4 was first prototyped in Haskell, and then automatically translated to its Isabelle/HOL [114] specification, and this specification was proved to refine its abstract specification, and its C implementation was proved to refine its Isabelle/HOL specification.

2.5.4 Software Model Checking for Machine-code Programs

Apart from Theorem Proving, other techniques have been used to check that safety properties hold in machine-code programs. McVETO [115] uses software model-checking [116] to check if a stripped machine-code program satisfies a safety property. McVETO resolves many soundness issues faced by software model-checking when applied to machine-code programs, such as sound disassembly and self-modifying code, which this dissertation does not address. Xu et al. [117] developed a type-state-based static-analysis tool and defined a calculus for their analysis for checking if safety predicates hold on machine-code programs, and they used the Induction Iteration Method to generate invariants for loops. In general, Theorem Proving approaches, such as in this dissertation, can generate more concise theorems describing safety properties such as CFI, than with software model-checking and static-analysis of machine-code.

2.5.5 Verified Compilation and Analyses

Verified compilers, such as CompCert [21], have been proven to produce machine-code programs that have memory safety, which implies CFI, when the source program has well-defined semantics. However, CompCert is unable to produce a proof stating that the machine-code program has memory safety nor CFI, whereas our approach is able to produce proofs for our target programs. In addition, CompCert is currently not able to identify if a given C program that it is compiling does indeed have well-defined semantics. However, recent work on tracking properties about memory contents in the memory model used in CompCert can provide static analyses about memory safety [118].

The Verified Software Toolchain (VST) [119], which is built on top of CompCert, is an approach that enables proofs about programs to be carried out at the source-level, such that these proofs will be preserved during compilation via the use of a verified compiler such as CompCert. However, VST is designed for proofs to be carried out manually at the source-level, albeit for general program properties, whereas our approach generates proofs automatically, but only for the single property of CFI.

Verasco [120] is a static-analyzer for C programs that has been formally verified to be correct, and whose analysis results carry over to compiled code due to its use of CompCert. Verasco can check for the absence of run-time errors that can cause safety violations such as CFI violations that we prove the absence of in our approach, but does not produce proofs for individual programs.

2.6 Attacks on Control-Flow Integrity

In recent work, attacks against CFI have been described that take advantage of the coarse granularity of practical CFI techniques, as well as residual threats due to data attacks. We summarize these attacks as well as whether our approach is vulnerable to each attack in Table 2.2.

The main attacks that have been described have been on the feasibility of gadget construction to enable ROP attacks even for programs protected by CFI [122, 123]. However, such ROP attacks require the use of explicit function pointers, which we disallow in our approach. Hence, such attacks are not relevant to our approach. Carlini et al. [121] then described attacks that specifically target weaknesses in the techniques designed to detect ROP attacks [71, 72].

Data attacks that can affect the control-flow of vulnerable programs have also been presented. Carlini et al. [30] described Control-Flow Bending (CFB), a class of attacks that overwrite non-

control-data that can have effects on control-flow, such as in the `printf` function when using the "`%n`" format specifier. Our approach is not vulnerable to such attacks, as we require all jump targets to be statically available for our safety proof, hence behavior such as in the "`%n`" format specifier for the `printf` function would be disallowed by our approach. Stack Defiler [124] described attacks that corrupt callee-saved registers and shadow call stacks to overcome protected shadow stack enforcement of CFI for function returns. Our approach is not vulnerable to Stack Defiler's attack, as we explicitly prevent overwriting of callee-saved registers in order to maintain the integrity of the activation records of all functions, and our protection of function return addresses relies on preventing writes to function return addresses rather than bookkeeping via a protected shadow call stack.

On the other hand, Control Jujutsu [125] described data attacks via overwriting of non-control-data such as system call arguments (e.g., arguments to the `execve` system call), and to overwrite data that affects dispatcher function selection. Our approach is not vulnerable to dispatcher function selection modification, since we disallow explicit function pointers. However, our approach is vulnerable to the overwriting of system call arguments, as we currently do not protect such data items in memory.

Technique	Location	Enforcement	Granularity	Function-Return Protection	Verification
Abadi et al. [15]	Machine-code	Abort on failure	1-ID	Protected shadow call stack	Formalization of technique [47]
XFI [16]	Machine-code	Abort on failure	N-ID (fine-grained)	Protected call stack	Hoare Logic verifier (hint-driven)
PittSFIeld [17]	Assembly code	Clobber illegal addresses	Aligned jump targets	Clobber illegal addresses	Machine-checked proof of verifier
WIT [48]	Compiler IR	Raise exception on failure	N-ID (Precise)	Provided by write-integrity	None
NaCl [49]	Assembly code	Clobber illegal addresses	Aligned jump targets	Clobber illegal addresses	Proved correctness of design
ARMor [18]	Machine-code	Abort on failure	Fine-grained	Protected shadow stack	Generates safety proof
Control-Flow Locking [50]	Assembly code	Clobber addresses + Abort on failure	2-ID (indirectly vs. directly called)	2-ID	No formalization
Zeng et al. [26]	Compiler IR	Abort on failure	1-ID	None	Unvalidated verifier
MoCFI [51]	Machine-code	Abort on failure	Fine-grained	Protected shadow stack	None
Pewny et al. [52]	Machine-code	Abort on failure	Fine-grained	None	None
MIP [53]	Assembly code	Abort on failure	Aligned jump targets	None	None
bin-CFI [54]	Machine-code	Abort on failure	2-ID (intra- vs. inter-module)	None (Can return to any valid return-site)	None
CCFIR [25]	Machine-code	Abort on failure	3-ID	Partial (Order not enforced)	Unvalidated verifier
MCFI [55]	Machine-code	Abort on failure	N-ID	None	Unvalidated verifier
KCoFI [56]	Machine-code	Not described	1-ID	None	Machine-checked proof of correctness of scheme
Code Pointer Integrity [57]	Machine-code	Abort on failure	N-ID (Precise)	Provided by write-integrity	Formally modeled semantics of scheme
Forward-edge CFI [58]	Compiler IR	Abort on failure	Variable (configurable)	None	None
CCFI [59]	Machine-code	Abort on failure	N-ID (fine-grained)	Write-integrity (Encryption of return address)	None
Opaque CFI [60]	Machine-code	Abort on failure	Coarse-grained	None	None
PiCFI [61]	Compiler IR	Abort on failure	Per-input	Partial	None
TypeArmor [62]	Machine-code	Abort on failure	Call/Return type matching	Partial	None
Our Approach	Source-code	Customizable recovery actions	Not Applicable	Protect stack	Generates safety proof

Table 2.1. Comparison of CFI techniques.

Attack	Description	Vulnerable?
History hiding and evasion attacks [121]	Evade ROP detection techniques	Not vulnerable (function pointers not allowed; function return pointers protected from overwriting)
Out of Control [122]	Call-site and Entry-point gadget construction for ROP	Not vulnerable (function pointers not allowed; function return pointers protected from overwriting)
Davi et al. [123]	Construct ROP gadgets	Not vulnerable (function pointers not allowed; function return pointers protected from overwriting)
Control-Flow Bending (CFB) [30]	Attack non-control data in `printf`, and write pointers	Not vulnerable (function pointers not allowed; function return pointers protected from overwriting)
Stack Defiler [124]	Corrupt callee-saved registers and shadow call stacks	Not vulnerable (callee-saved registers, function return addresses protected from overwriting)
Control Jujutsu [125]	Attack system call arguments, affect dispatcher function-pointer selection	Vulnerable to system call argument attacks (data attacks)

Table 2.2. Summary of attacks on CFI and vulnerability of our approach to such attacks.

Chapter 3

Preventative Control-Flow Integrity

In this chapter, we describe our approach for providing enforcement mechanisms for Control-Flow Integrity (CFI). Our approach uses source-code mechanisms to prevent the root-causes of CFI violations. Our approach generates prescriptions of source-code locations where CFI enforcement mechanisms are required, and generates source-code enforcement mechanisms to be used for enforcing CFI. Programmers then apply these prescriptions to the source-code of their programs to insert CFI enforcement mechanisms. This enables programmers to consider the original functionality of their program when adding CFI enforcement mechanisms, and to ensure that the inserted CFI enforcement mechanisms do not change the functionality of their program in unintended ways.

We begin by concisely defining the Control-Flow Integrity (CFI) safety and security property, and we provide a concrete example of CFI for ARM executables. Then, we explore the root-causes of CFI violations (§3.1), and we describe how CFI can be enforced using source-code mechanisms in a preventative manner by preventing the root-causes of CFI violations (§3.2). Next, we describe the design and implementation of our PCFIRE-C tool [126] for prescribing source-code CFI enforcement mechanisms for programmers (§3.3). Finally, we describe the source-code constructs that are supported by PCFIRE-C's safety-check prescriptions (§3.4).

3.1 Control-Flow Integrity

3.1.1 Illustration of CFI

Control-Flow Integrity (CFI) is a safety property that states that the execution of software follows a path in a Control-Flow Graph (CFG) that is "determined ahead of time" [15]. While programmers reason about software behaviors in the programming language they use (e.g., C), software execu-

tion 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 that 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, and this can lead to the introduction of unexpected behavior into the program, which can enable attacks on the software. 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 the piece of C code below in Figure 3.1.

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

Figure 3.1. Example C function with a potential CFI violation.

At first glance, `arraycopy` is designed to copy the array `src` to the array `dst`. A common dangerous class of bugs in functions that write to arrays is buffer overflows, which SANS ranks as the third most dangerous bug [127], as they can give rise to CFI violations. 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`", as shown in Figure 3.2.

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

```
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
...
```

Figure 3.2. Selection of compiled ARM machine-code for C code in Figure 3.1. Some of the instructions include: Addition (`add`); Compare (`cmp`); Load Register (`ldr`); Logical Shift Left (`lsl`); Push to Stack (`push`); Store (`str`); Subtraction (`sub`).

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`.

3.1.2 Definition of CFI

Next, we define safety properties for a machine-code program, such that for a machine-code program for which our safety properties hold, the CFI for the program cannot be violated, and the

program possesses CFI. These safety properties are fundamental to our approach in this dissertation for enforcing and proving CFI. Later in this dissertation, we will discuss how these safety properties are formally specified and proved in §4 and §5. We first define these safety properties, then we explain how these safety properties are sufficient to ensure CFI holds, and we explain how these safety properties ensure CFI for a machine-code program targeted at the ARM architecture.

The safety policy of our enforcement approach comprises the following safety properties that we wish to enforce and prove for machine-code programs to ensure CFI. Our safety policy comprises the following safety properties:

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

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

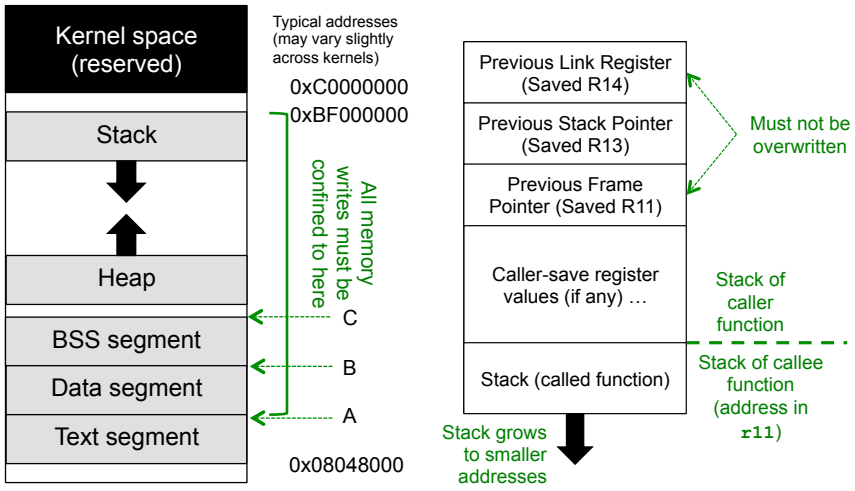
Property 3. Only initially-loaded instructions are executed.

Together, these properties eliminate the root-causes of CFI violations. We explain how these properties are sufficient to ensure that 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 (i.e., is contained within) the CFG captured by its source-code. 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 basic blocks of instructions, and edges representing control transfers between instructions. Then, our three safety properties 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 §1.3).

Table 3.1 summarizes the ways in which a (machine-code) program's CFG can be changed at run-time, resulting in a CFI violation. Each way in which the program's CFG can be changed represents a possible root-cause of a CFI violation. Then, we describe how the program's CFG is affected by each root-cause of a CFI violation, and we describe which of our three safety properties prevents each root-cause of a CFI violation. Note that arbitrary edges cannot be simply added to the CFG of a machine-code program, because adding an edge to the CFG corresponds to adding a jump target, which requires an instruction, i.e., a vertex in the CFG, to be changed.

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

Table 3.1. CFI via AUSPICE's Safety Theorem



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

Figure 3.3. CFI safety as provided by PCFIRE-C's enforcement (diagram from [128]).

3.1.3 Control-Flow Integrity for the ARM Architecture

Next, we concretely describe the behaviors that a Linux user-mode program executing on the ARM architecture must have for it to meet our three safety properties. First, consider the memory layout for a user-mode process in Linux for an architecture with a 32-bit address space (Figure 3.3(a)). To prevent overwriting of loaded program instructions (Property 1), all memory writes must be restricted to addresses from `0xBF000000`, the largest user-mode addressable address, to the location `A` in Figure 3.3(a), the 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 [28] (Figure 3.3(b)). In a function call, the prologue of a function saves the values of callee-save registers to the stack. Based on the ATPCS, these callee-saved registers include the link register (`r14`), stack pointer (`r13`) and frame pointer (`r11`). The link register stores the function return address of the caller function, and must not be overwritten, for Property 2 to hold. 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, and for a statically-linked program, this range of addresses can be determined statically.

3.2 Design: Source-code Enforcement of CFI

Safety-checks are needed before any potentially dangerous C statement (which we will call *suspect statements*). First, we explain how safety-checks can be constructed using purely source-code mechanisms in C programs to ensure that our 3 CFI safety properties (§3.1.2) hold. This prevents the root-causes of CFI from occurring, thus enforcing CFI in a preventative manner. Later, in §3.3, we explain how suspect statements are identified, and how we construct prescriptions of source-code safety-checks that are specific to each suspect statement. The C statements in our safety-checks presented here are not verified at the C level; instead, in our logic approach, the machine-code compiled from our check-and-branch statements can be verified to possess the safety properties that we defined (§3.1.2) to ensure that CFI holds.

Suspect statements may cause CFI violations, and hence need to be surrounded with safety-checks. Each safety-check is a *check-and-branch* statement in C: the safety-check *checks* if the suspect statement will cause a CFI violation when run: if so, it *branches* to an alternative statement (which could be a recovery action); if not, it allows the suspect statement to run. By running safety-checks before suspect statements, it is possible to recover before a CFI violation occurs.

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

saved to the stack, ensuring Property 2 holds will ensure that Property 3 holds. 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. Both Properties 1 and 2 require memory addresses being written to, to be within safe ranges. Hence, the check-and-branch statements need to (i) extract the addresses being written to by the C statement, and (ii) ensure that these addresses are within the required safe ranges. The safe address range for Property 1 can be statically obtained from the machine-code (e.g., using GNU `readelf` to extract the address and size of the `text` section), while the safe range for Property 2 requires the frame pointer value (in the link register, `r11`) to be extracted at run-time.

3.2.1 Check-and-Branch Statements for C Programs

We briefly describe how safety-checks comprising check-and-branch C statements can be constructed to prevent the root-causes of CFI from occurring.

The key idea of the safety-checks in our enforcement approach is to provide a guard that ensures that suspect statements with memory writes (which cannot be automatically proved to be safe in our logic approach) will execute only if the target address being written to is safe with respect to our safety policy (§3.1.2). We need to ensure that the safety guards, which are C statements, will compile to machine-code that can automatically be proved to be safe by our logic approach (which we will describe in §4).

Each safety-check, or guard, is an `if`-statement that surrounds a suspect statement. Then, the guard checks if the memory address(es) being written to by the suspect statement will violate our safety properties. Effectively, the guard checks if the memory address being written to by the suspect statement might modify loaded instructions, or if it might modify function return addresses saved to the stack. Each clause in the `if`-statement of the guard comprises two parts: (i) the address being written to by the suspect statement in the left-hand-side (LHS) of the clause, and (ii) the upper or lower bound allowed for the memory address being written to in the right-hand-side (RHS) of the clause. Then, the full guard comprises a conjunction of these safety-check clauses.

Extracting Memory Addresses Written To

The LHS of each clause in a guard contains the memory address being written to by a suspect statement. Table 3.2 shows what the LHS of each clause in a guard would be for extracting the

memory addresses written to by a suspect statement, for some examples of expressions that write to memory in a suspect statement. For large or complex memory-write expressions in suspect statements, it is possible to use a pointer to extract the address being written to (e.g. for "`<lhs expr> = <expr>;`", we can use "`tmp_ptr = &(<lhs expr>); (*tmp_ptr) = <expr>;`").

Statement type	Original expression	Expression for safety check (i.e., L-Value of 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 3.2. LHS expressions for safety guards.

Safe Address Bounds

Next, each clause in a guard requires an upper-bound or a lower-bound for allowed memory-write addresses. Table 3.3 lists the memory-write bounds required for each guard to enforce our safety properties, describes the value of each upper-bound or lower-bound on allowed memory-write addresses, and describes where the concrete value for the bound can be obtained. These bounds are constructed to enable our logic approach (§4) to automatically prove that our safety properties for CFI hold.

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 3.3. RHS values for safety guards.

Finally, additional code is required to dynamically extract the current value of the frame pointer from register `r11` before each 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 our logic approach is able to use the value in its comparison. Figure 3.4 shows the macro used to extract the current value of the frame pointer to a temporary register for checking against in the guard, and the C variable declaration required for a local variable stored in a register to store the frame pointer value. Unfortunately,

our method for extracting the value of the current frame pointer requires the ability to execute arbitrary assembly code fragments to directly read register values; to apply our approach to another programming language, we would need to be able to directly read the contents of a specified register.

```
#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 3.4. 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 3.5. Full safety-check for statement, "`s[i] = <expr>;`" in a function with 3 callee-saved registers.

Figure 3.5 puts together the full guard for a suspect statement, which writes to memory, "`s[i] = <expr>;`". The guard also allows programmers to specify their own recovery actions in the `else` branch of the safety guard, although we do not discuss recovery actions in this dissertation. 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.

3.3 Implementation: Prescriptions of CFI Safety-checks

Next, we describe the implementation of the PCFIRE-C tool, which realizes the enforcement approach in this dissertation. Figure 3.6 shows the workflow of PCFIRE-C. PCFIRE-C can operate in two modes: (i) **heuristic mode**, where it takes as input the machine-code of a program without safety-checks, and (ii) **proof-failure mode**, where it takes as input the locations of proof failures

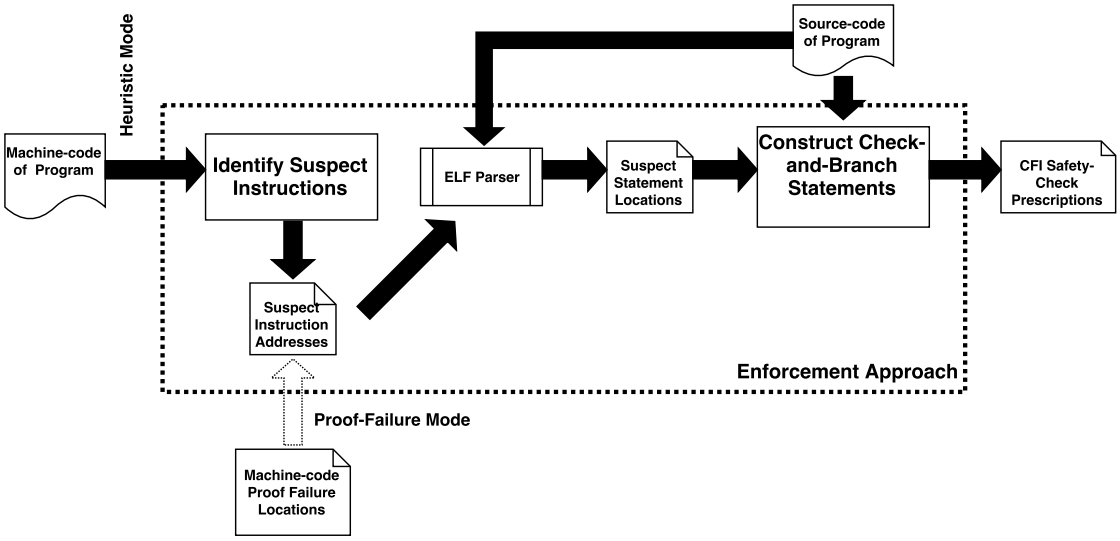


Figure 3.6. Overview of approach for prescribing CFI safety-checks.

in the machine-code of the program, where the logic approach of this dissertation failed in its attempted safety proof. Then, PCFIRE-C produces a list of safety-check prescriptions, where each prescription consists of the location of a suspect statement, and a safety-check C statement for the suspect statement.

First, PCFIRE-C needs a list of addresses of suspect machine-code instructions, whose corresponding source-code statements are suspect statements, and are in need of prescriptions of safety-checks. In the heuristic mode, PCFIRE-C identifies such suspect instructions using a heuristic about the targets of store instructions, which we describe later. In the proof-failure mode, PCFIRE-C receives a list of addresses of suspect machine-code instructions, which are the addresses of instructions that our logic approach fails to prove are safe. The key difference between the two modes, is that in the heuristic mode, PCFIRE-C is unable to identify if the machine-code already contains sufficient safety-checks for proving that a given suspect instruction is safe, whereas in the proof-failure mode, our logic approach will identify only suspect instructions that could not be proved to be safe. In other words, the heuristic mode of PCFIRE-C is an unsound (i.e., false-positives are possible, meaning that safety-checks could be prescribed even if a valid safety-check is already present) way to identify suspect statements, but is significantly faster than the proof-failure mode, since our logic approach utilizes a heavyweight proof assistant.

In both modes, PCFIRE-C then proceeds to identify the source-code location corresponding to each machine-code instruction by parsing the debug information using an off-the-shelf parser for ELF (Executable Link Format) and its DWARF debug information format. Then, PCFIRE-C

constructs check-and-branch statements for the suspect statements in each identified source-code location.

The identification of suspect instructions in PCFIRE-C's heuristic mode is implemented as a Python script which parses the assembly-text output from the `objdump` disassembler, while the construction of the check-and-branch safety-check prescriptions are implemented on top of the Clang [129] compiler front-end for C programs.

Next, we describe PCFIRE-C's heuristics for identifying suspect instructions in a machine-code program, and we describe PCFIRE-C's construction of its prescribed safety-checks for each suspect statement.

3.3.1 Identifying Enforcement Locations for CFI

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. This analysis is implemented using Python. We describe the analysis to identify instructions that would cause the CFI proof in our logic approach to fail. The main goal of this analysis is to identify dangerous memory-write instructions for which additional run-time source-code safety-checks are needed. Note that the analysis identifies potentially dangerous instructions, but is unable to check if the necessary safety-checks are already present.

First, we identify store instructions, which write to memory, i.e., `str`, `strb`, `strh`. Then, we consider the addresses in memory that are written to by the instruction. As described in §3.1.1, our main heuristic is that 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, while memory-write instructions writing to an arbitrary computed address cannot be automatically proved to be safe, and will need additional run-time safety-checks, which PCFIRE-C will prescribe.

Informally, at a high-level, instructions writing to a constant offset from the frame pointer can be automatically proved to be safe. From §3.1.3, based on our safety properties, we need to ensure that the memory addresses written to do not overwrite loaded program instructions, and that they do not overwrite function return addresses saved to the program's stack. Then, since the ATPCS [28] specifies the locations in memory where function return addresses are saved to in terms of the frame pointer, `fp`, memory addresses written to that are a constant offset from `fp` can be statically proved to not overwrite the saved function return addresses. Also, to prove that the memory addresses written to will not overwrite loaded program instructions, since the value of `fp` typically does not

change within the body of a function, we only need to prove that the value of `fp` is in a particular range such that the constant offsets are not able to overwrite loaded program instructions. This safe range of addresses for `fp` can then be added as a hypothesis to the safety proof for the function that the current instruction is in.

On the other hand, instructions that write to computed addresses (e.g., "`str r2 [r3]`") cannot always be automatically proved to be safe in the general case. This is because the computed address, e.g., the value held in register `r3`, is arbitrary in the general case, and could point to any location. Also, the computed address `r3` may depend on other values that either (i) depend on external inputs to the current function, e.g., arguments supplied to the function, or (ii) change in a loop prior to being used, which will then require manually-specified loop invariants for our logic approach to reason about, thus making them impossible to reason about automatically.

Thus, the analysis ignores writes to a constant offset from the frame pointer, as they can be automatically proved in our logic approach to not violate our CFI safety properties. Then, the analysis reports the address of any other store instruction. The analysis ignores the `push` instruction, as we observed that `gcc`-emitted code only uses `push` in function prologues. (In the event that the `push` instruction is used outside of the function prologue, or the frame pointer is modified outside of the function prologue, the proof attempt in our logic approach will fail to generate a CFI safety proof.) The analysis also extracts the number of callee-saved registers for each function from `push` instructions in function prologues to construct check-and-branch statements (§3.2.1).

To summarize, the analysis to identify suspect locations in a machine-code program is implemented as a regular expression search over the disassembled text of the machine-code program in Python, and is implemented in approximately 150 lines of Python script.

3.3.2 Construction of Safety-Check Prescriptions for C Programs

Next, we describe how PCFIRE-C constructs prescriptions of safety-checks for a C program. The input to PCFIRE-C's prescription-construction is a list of suspect statement locations, which comprise: (i) source file name, (ii) source line number, and (iii) type of suspect statement (which can be a safety-assertion failure, or an invocation of a system call wrapper, see §6.1 for more details). The algorithm, based on our heuristics, and its details are captured in Algorithms 3.1, 3.2 3.3, 3.4, 3.5 and 3.6.

The prescription of safety-checks in PCFIRE-C is implemented in C++ using the Tooling interface provided by the Clang [129] C compiler front-end, which parses each C source-code file and

provides an Abstract Syntax Tree (AST) for each source-code statement. Algorithm 3.1 summarizes the overall workflow in PCFIRE-C for constructing prescriptions for safety-checks. For each input source-code file, PCFIRE-C goes through the ASTs for the statements in the source-code. Then, PCFIRE-C retrieves the AST for each statement in the source-code file corresponding to each input suspect location. PCFIRE-C then constructs the safety-check prescription for each suspect location based on the retrieved AST for the source-code statement.

Algorithm 3.1 PCFIRE-C algorithm for generating safety-check prescriptions.

```
 1: function PRESCRIBESAFETYCHECKS(suspect_locations, src_files)
 2:     for curr_src_file ∈ src_files do
 3:         suspect_stmts ← ∅
 4:         for stmt ∈ curr_src_file do                ▷ Collect AST of suspect statements
 5:             stmt_loc ← COMPUTELOCATION(stmt)
 6:             if stmt_loc ∈ suspect_locations then
 7:                 suspect_stmts ← suspect_stmts ∪ stmt
 8:             end if
 9:         end for
10:         for curr_suspect ∈ suspect_stmts do
11:             CONSTRUCTPRESCRIPTION(curr_suspect)
12:         end for
13:     end for
14: end function
```

Next, PCFIRE-C constructs the safety-check prescription for each suspect location, given the (Clang-generated) AST of the C statement at that location. Algorithm 3.2 summarizes the construction of the safety-check prescription for each suspect location. There are two types of CFI safety failures for which safety-check prescriptions are constructed: (i) those for safety assertion failures, which occur when a suspect statement writes to memory (i.e., the C statement is compiled to machine-code instructions that include any one of the `str`, `strb`, or `strh` instructions), and (ii) those for invocations of system call wrappers, whose invoked system call results in user-mode memory being modified (e.g., the `read()` system call) (see §6.1 for more details about our handling of system calls). PCFIRE-C handles these two cases separately.

For each suspect statement, the prescription-construction (Line 32) selects the appropriate handler function (Lines 13, 20, 24) to handle the suspect statement based on whether the suspect statement contains a safety assertion failure or a system call wrapper invocation. Each of these functions performs two tasks: (i) they extract the expression for the LHS of the safety-check, which represents the memory address(es) that are written to by the statement, and (ii) they invoke a helper function (Line 1) to assemble the text of the prescription and print it out.

For safety assertion failures, PCFIRE-C handles only: (i) statements that contain an assignment (Line 14), whose memory-write effect applies to the left-hand-side of the assignment statement, and (ii) statements that contain a unary operation (i.e., prefix or postfix increment or decrement) (Line 20). System call wrapper invocations are handled in Line 24.

Next, each of the handler functions for safety assertion failures and system call wrapper failures invokes a single handler function, `HandleLHS`, which extracts the expression for the LHS of the safety-check. The `HandleLHS` handler function is described in Algorithm 3.4. `HandleLHS` takes as input the AST of the suspect statement, and it calls the appropriate traversal function to traverse the AST to obtain the LHS expression for the safety-check prescription. Before traversing the AST, the AST is first converted to a queue representation by a pre-order traversal of the tree, as described in Algorithm 3.3.

Then, Algorithm 3.5 describes the AST traversal to construct the LHS expression for the safety-check prescription for safety assert failures. The traversal for safety assert failures recursively traverses the AST (using the queue), and handles the different possible syntactic constructs to reconstruct the source-code expression for the memory address(es) that are written to by the suspect statement. To maximize the flexibility of our approach, we reconstruct the expression of the address written to, and use the C address-of operator (`&`) to obtain the address written to, and we suggest to the programmer to use a pointer to write to the location (Line 9 in Algorithm 3.2).

Algorithm 3.6 describes the AST traversal for system call wrappers. We currently support generating safety-check prescriptions only for our wrapper to the `read()` system call (as this is the only user-mode-effect-ful system call that we current support in our approach that results in writes to user-mode memory). Hence, in `TraverseQueueNode_SYSCALL_BUF`, we currently only identify buffers and have a simplified AST traversal (Line 2), although it is straightforward to extend the traversal in a similar way to Algorithm 3.5. Also, our logic approach supports only safety proofs where a concrete-sized buffer is written to (see §6.1), hence we only handle integer literals in `TraverseQueueNode_SYSCALL_LEN` (Line 24).

To summarize, PCFIRE-C constructs safety-check prescriptions given the source-code of a program, and a list of suspect locations (i.e., source file names and line numbers) in the source-code of the program. PCFIRE-C constructs these prescriptions on a per-suspect basis for two types of suspect statements (safety assertion failures and invocations of effect-ful system call wrappers), and extracts a source-code expression for the memory address written to, to be used in the construction of the prescription. PCFIRE-C then uses Clang's `DiagnosticsEngine` to print and highlight the

source-code expressions requiring safety-checks, and the prescribed safety-check for each suspect statement. The full implementation of PCFIRE-C comprises approximately 1500 lines of C++ code, and is built as a Clang tool using the Tooling interface.

3.4 Scope of Safety Check Prescriptions

Next, we discuss the C syntactic constructs which are supported, and those which are unsupported, by the PCFIRE-C tool, for which safety-check prescriptions can and cannot be generated respectively by our tool.

We refer to the C syntax for the C99 standard as described in Appendix B of [130]. A C program is comprised of one or more functions, whose bodies comprise one or more statements. Each C statement can be one of the following types:

- Expression statement: one or more assignment expressions separated by a comma (May be effect-ful)
- Labeled statement
- Compound statement: one or more declarations separated by ";"s (May be effect-ful)
- Conditional statement: `if` or `if-else` statement
- Iterative statement: `while`, `do`, or `for` loop
- Switch statement
- Break statement
- Continue statement
- Return statement
- Goto statement
- Null statement (single ";")

PCFIRE-C needs to generate safety-checks for "effect-ful C operations" that result in writes to memory that cannot be automatically proved to be safe. From the above list of syntactic constructs, effect-ful C behaviors (that alter memory) can occur in expression statements containing assignments (including effect-ful unary prefix and postfix operators such as "++" and "--"), in compound

Algorithm 3.2 Construction of Safety-Check Prescription for each Suspect Statement.

```
 1: function ConstructPrescriptionText(lhs, num_stack_saved, ty_expr)
 2:     remedy := "register unsigned int r11_val asm ("r4");\n"
 3:     remedy += ty_expr + " tmp_ptr = &(" + lhs + ");\n"
 4:     remedy += "GET_FRAME_POINTER(r11_val);\n"
 5:     remedy += "if ( ( (unsigned int) (tmp_ptr) >= MIN_SAFE_MEM) && \n"
 6:     remedy += " ( ( (unsigned int) (tmp_ptr)) <= MAX_SAFE_MEM) && \n"
 7:     remedy += " ( ( (unsigned int) (tmp_ptr)) < (r11_val - (" + num_stack_saved + " * WORD_SIZE))) && \n"
 8:     remedy += "(r11_val >= (" + num_stack_saved + " * WORD_SIZE))) {"
 9:     remedy += "*tmp_ptr = ...<RHS expression>... \n"
10:     remedy += "}"
11:     PrintRemedy(remedy)
12: end function
13: function HandleSafetyAssertFail_BinOp(binop, num_stack_saved)
14:     if binop→getOpcode( ) == BO_Assign then
15:         lhs := binop→getLHS( )
16:         (safety_check_lhs, ty_expr) := HandleLHS(lhs, SAFETY_ASSERT_BINARY)
17:         ConstructPrescriptionText(safety_check_lhs, num_stack_saved, ty_expr, ty_expr)
18:     end if
19: end function
20: function HandleSafetyAssertFail_UnOp(unop, num_stack_saved)
21:     (safety_check_lhs, ty_expr) := HandleLHS(unop, SAFETY_ASSERT_UNARY)
22:     ConstructPrescriptionText(safety_check_lhs, num_stack_saved, ty_expr)
23: end function
24: function HandleSyscallWrapperFail(callexpr, num_stack_saved)
25:     syscall_args := callexpr→getArgs( )
26:     if callexpr→getCallee( ) == "c_read" then
27:         (buf_name,ty_expr1) := HandleLHS(syscall_args[1], SYSCALL_WRAPPER_BUF)
28:         (buf_len,ty_expr2) := HandleLHS(syscall_args[2], SYSCALL_WRAPPER_LEN)
29:         ConstructPrescriptionText(buf_name+"+"+buf_len, num_stack_saved, ty_expr1)
30:     end if
31: end function
32: function ConstructPrescription(stmt)
33:     num_stack_saved := FailStmtFunctionStackSaved(stmt)      ▷ Look up number of stack-saved registers in
    function for given statement.
34:     if FailType(stmt) == SafetyAssertFail then
35:         if isa<BinaryOperator>(*stmt) then
36:             HandleSafetyAssertFail_BinOp((BinaryOperator *) stmt, num_stack_saved)
37:         else if isa<UnaryOperator>(*stmt) then
38:             HandleSafetyAssertFail_UnOp((UnaryOperator *) stmt, num_stack_saved)
39:         end if
40:     else if FailType(stmt) == SyscallWrapperFail then
41:         if isa<CallExpr>(*stmt) then
42:             HandleSyscallWrapperFail((CallExpr *) stmt, num_stack_saved)
43:         end if
44:     end if
45: end function
```

Algorithm 3.3 AST to Queue Conversion

```
1:  function AST2QUEUE(stmt, parent, ast_q, parent_q)
2:      ast_q.push_back(stmt)
3:      parent_q.push_back(parent)
4:      curr_idx := ast_q.size() − 1
5:      for child ∈ stmt.children() do
6:          (ast_q, parent_q) := AST2QUEUE(child, curr_idx, ast_q, parent_q)
7:      end for
8:      return (ast_q, parent_q)
9:  end function
10: function ASTQUEUECOMPUTECHILDREN(parent_q)
11:     child_q := ∅; i := 0
12:     while i < parent_q.size() do
13:         j := 0
14:         curr_children := ∅
15:         while j < parent_q.size() do
16:             if j ≠ i ∧ parent_q[j] == i then
17:                 curr_children := curr_children ∪ j
18:             end if
19:         end while
20:         i++
21:     end while
22:     return child_q
23: end function
```

Algorithm 3.4 Construction of memory-write address expression (i.e., LHS of safety-check).

```
1:  function HANDLELHS(stmt, stmt_type)        ▷ stmt is AST for memory-write expression
2:      ast_q := ∅; parent_q := ∅
3:      (ast_q, parent_q) := AST2QUEUE(stmt, -1, ast_q, parent_q)
4:      child_q := ASTQUEUECOMPUTECHILDREN(parent_q)
5:      if stmt_type == SAFETY_ASSERT_BINARY then
6:          TRAVERSEQUEUENODE_ASSERT(0, ast_q, parent_q, child_q, false)
7:      else if stmt_type == SAFETY_ASSERT_UNARY then
8:          TRAVERSEQUEUENODE_ASSERT(0, ast_q, parent_q, child_q, true)
9:      else if stmt_type == SYSCALL_WRAPPER_BUF then
10:         TRAVERSEQUEUENODE_SYSCALL_BUF(0, ast_q, parent_q, child_q)
11:     else if stmt_type == SYSCALL_WRAPPER_LEN then
12:         TRAVERSEQUEUENODE_SYSCALL_LEN(0, ast_q, parent_q, child_q)
13:     end if
14: end function
```

Algorithm 3.5 Construction of LHS expression for safety assertion failures.

```
1:  result_q := []                                          ▷ Global queue for accumulating results
2:  type_q := []
3:  function TraverseQueueNode_ASSERT(idx, ast_q, parent_q, child_q, is_unary)
4:      for child ∈ child_q[idx] do
5:          TraverseQueueNode_ASSERT(child, ast_q, parent_q, child_q, is_unary)
6:      end for
7:      curr_children := child_q[idx]
8:      if isa<DeclRefExpr>(ast_q[idx]) then
9:          result_q[idx] := (ast_q[idx])→getFoundDecl()→getNameAsString()
10:         type_q[idx] := ast_q[idx]→getType()
11:     else if isa<CastExpr>(ast_q[idx]) then
12:         result_q[idx] := result_q[curr_children[0]]
13:         type_q[idx] := ast_q[idx]→getType()
14:     else if isa<BinaryOperator>(ast_q[idx]) then
15:         curr_op := ast_q[idx]→getOpCodeStr()
16:         result_q[idx] := result_q[curr_children[0]] + curr_op + result_q[curr_children[1]]
17:     else if isa<UnaryOperator>(ast_q[idx]) then
18:         curr_op := ast_q[idx]→getOpcode()
19:         if curr_op == UO_Deref then
20:             result_q[idx] := "("+result_q[curr_children[0]]+")"
21:         else if curr_op == UO_PreInc ∧ not is_unary then
22:             result_q[idx] := "("+result_q[curr_children[0]]+" + 1)"
23:         else if curr_op == UO_PreDec ∧ not is_unary then
24:             result_q[idx] := "("+result_q[curr_children[0]]+" - 1)"
25:         else result_q[idx] := "("+result_q[curr_children[0]]+")"
26:         end if
27:     else if isa<IntegerLiteral>(ast_q[idx]) then
28:         result_q[idx] := ast_q[idx]→APIntStorage::getValue().toString()
29:     else if isa<ArraySubscriptExpr>(ast_q[idx]) then
30:         result_q[idx] := curr_children[0] + "[" + curr_children[1] + "]"
31:     else if isa<MemberExpr>(ast_q[idx]) then
32:         if ast_q[idx]→MemberExpr::isArrow() then
33:             result_q[idx] := curr_children[0] + "->" + curr_children[1]
34:         else
35:             result_q[idx] := curr_children[0] + "." + curr_children[1]
36:         end if
37:         type_q[idx] := "(" + ast_q[idx]→getType() + " *)"
38:     else if isa<ParenExpr>(ast_q[idx]) then
39:         result_q[idx] := "(" + curr_children[0] + ")"
40:     end if
41:     if idx == 0 then
42:         i := 0; type_str := ""
43:         while i < ast_q.size() do
44:             if type_q[i] != "" then
45:                 type_str := type_q[i]; break;
46:             end if
47:             i++
48:         end while
49:         return (result_q[0], type_str)
50:     end if
51: end function
```

Algorithm 3.6 Construction of LHS expression for syscall wrapper failures.

```
1:  result_q1 := []; type_q1 := [];                          ▷ Global queue for accumulating results
2:  function TRAVERSEQUEUENODE_SYSCALL_BUF(idx, ast_q, parent_q, child_q)
3:      for child ∈ child_q[idx] do
4:          TRAVERSEQUEUENODE_SYSCALL_BUF(child, ast_q, parent_q, child_q)
5:      end for
6:      if isa<DeclRefExpr>(ast_q[idx]) then
7:          result_q1[idx] := ast_q[idx]→getFoundDecl()→getNameAsString()
8:          type_q[idx] := ast_q[idx]→getType()
9:      end if
10:     for curr_child ∈ child_q[idx] do
11:         result_q1[idx] := result_q1[idx] + result_q1[curr_child]
12:     end for
13:     if idx == 0 then
14:         i := 0; type_str := ""
15:         while i < ast_q.size() do
16:             if type_q1[i] != "" then type_str := type_q1[i]; break;
17:             end if
18:             i++
19:         end while
20:         return (result_q1[0], type_str)
21:     end if
22: end function
23: result_q2 := [];                                          ▷ Global queue for accumulating results
24: function TRAVERSEQUEUENODE_SYSCALL_LEN(idx, ast_q, parent_q, child_q)
25:     for child ∈ child_q[idx] do
26:         TRAVERSEQUEUENODE_SYSCALL_LEN(child, ast_q, parent_q, child_q)
27:     end for
28:     if isa<IntegerLiteral>(ast_q[idx]) then
29:         result_q2 := ast_q[idx]→getValue().toString()
30:     end if
31:     for curr_child ∈ child_q[idx] do
32:         result_q2[idx] := result_q2[idx] + result_q2[curr_child]
33:     end for
34:     if idx == 0 then
35:         return (result_q2[0], "")
36:     end if
37: end function
```

statements where a declaration is followed by an effect-ful assignment, or in the sub-expressions of conditional and iterative statements that contain assignment expressions.

Labeled statements in themselves do not result in memory being written to during program execution, and only affect the generation of jump targets during compilation and linking.

Break, Continue, Return, and Goto statements do not typically result in memory being written to (except for returns of `structs`, which we will discuss in §8.1, and our logic approach does not support proving CFI safety for `struct` returns from functions).

Based on PCFIRE-C's algorithm for constructing safety-check prescriptions for each suspect statement (Algorithm 3.2), PCFIRE-C currently supports generating safety-check prescriptions for expression statements that contain a single assignment expression.

PCFIRE-C does not support effect-ful C operations that occur in compound statements. This is because the identification of suspect instructions (Figure 3.6) will associate each suspect instruction with a single source-code line, but in the case of compound statements occurring on the same source-code line, it is challenging to identify which of the compound statements corresponds to the particular suspect instruction.

Similarly, in the case of Switch, Conditional and Iterative statements whose conditional or looping expressions contain effect-ful operations, PCFIRE-C is unable to support generating a safety-check prescription. This is because PCFIRE-C's prescribed safety-check will require adding additional statements to perform the safety-check, but it will be challenging to insert a safety-check into the conditional or loop expression. In addition, we believe this will not be a significant problem for programmers, as any effect-ful operation (e.g., using a global variable as a loop counter) can be modified to use a local variable instead, and to perhaps save the value of the loop variable in a variable in a separate individual statement.

To summarize, PCFIRE-C supports the generation of safety-check prescriptions for C operations that are effect-ful on memory that occur in statements with a single assignment expression, because it is unable to associate a suspect instruction compiled from a line of C source-code with multiple possible effect-ful expressions. Also, PCFIRE-C does not support generating safety-check prescriptions for effect-ful C operations that occur in the conditional or looping expressions of Switch, Conditional, and Iterative statements because of the difficulty of constructing a safety-check prescription that can fit in the C syntax of Switch, Conditional, and Iterative statements.

3.5 Summary

This chapter described the enforcement approach for Control-Flow Integrity (CFI) in this dissertation, which is realized by the PCFIRE-C tool [126] that we developed. We concretely defined Control-Flow Integrity (CFI), and the safety properties that are verified in our logic approach in order to ensure CFI, as described later in §4.

Our approach to CFI enforces CFI by using source-code safety-checks that prevent the root causes of CFI violations. To enable programmers to ensure that these source-code safety-checks do not inadvertently change the functionality of their programs, our approach prescribes source-code safety-checks, and leaves it up to programmers to apply these prescriptions to their source-code.

To that end, we have developed the PCFIRE-C tool for prescribing source-code safety-checks for CFI, given the source-code and (initially compiled) machine-code of a program. The PCFIRE-C tool supports generating prescriptions of safety-checks in two modes: (i) by using heuristics to identify suspect machine-code instructions and their corresponding source-code statements that may result in CFI violations, and (ii) by using information about proof failures from our logic approach (which we will describe in later chapters). We also described the scope of C program constructs that are supported by the PCFIRE-C tool, and the limitations of the tool.

Chapter 4

Program Logic for Machine-code Safety-Property Proofs

In this chapter, and the next two chapters (§5, §6), we describe the various parts of the logic approach in this dissertation, as realized by the AUSPICE framework. AUSPICE automatically generates safety proofs for ARM machine-code programs [128], and supports machine-code programs with system calls [126].

This chapter describes the $\mathscr{L}_{LR}$ program logic, that we developed, which forms the foundation of AUSPICE. "LR" stands for "Local Reasoning", which is the principle we use to automate our safety property proofs. The $\mathscr{L}_{LR}$ program logic is designed for automatically reasoning about safety properties in machine-code programs, and it extends the program logic introduced by Myreen et al. [31, 32] by adding proof rules for reasoning locally about safety properties specified at a per-instruction level for machine-code programs. In the following two chapters (§5, 6), we describe how to automatically generate proofs of safety properties for machine-code programs in $\mathscr{L}_{LR}$ (§5) using AUSPICE, and we describe how to extend $\mathscr{L}_{LR}$ to support proofs for machine-code programs containing system calls, and how the proof automation process can be optimized (§6).

In this chapter, we begin by briefly introducing the "Cambridge ARM model" [131, 31, 32] in §4.1, which is a trustworthy formalization of the ARM Instruction Set Architecture (ISA) in Higher Order Logic (HOL). The Cambridge ARM model has been mechanized in the logic of the HOL4 proof assistant [38]. The Cambridge ARM model provides semantics for individual ARM machine-code instructions, and it provides a Hoare logic [132] for reasoning about ARM machine-code programs. In our logic approach, we leverage the Cambridge ARM model to provide us with a trustworthy semantics for reasoning about machine-code behavior. We illustrate how the semantics of ARM machine-code instructions are represented, and we describe the proof rules provided in the Hoare logic of the Cambridge ARM model. In §4.2, we present the proof rules for

$\mathscr{L}_{LR}$, which encode the "local reasoning" principle for reasoning about safety properties in machine-code programs in a way that is amenable to automated proof generation. In §4.3, we describe how PCFIRE-C's prescribed source-code safety-checks (§3.2) give rise to compiled machine-code programs for which our safety properties can be automatically proved in $\mathscr{L}_{LR}$. In §4.4, we present arguments for the soundness and correctness of the proof rules in $\mathscr{L}_{LR}$.

4.1 Hoare Logic for ARM Machine-code

In the logic approach in this dissertation, we use the HOL4 theorem prover [38], and the Hoare logic [31] for ARM machine-code programs [32] developed at Cambridge University, to prove safety theorems. The Cambridge ARM model has been extensively tested and validated [131], providing us with a strong, trustworthy foundation for our logic. The Cambridge ARM model uses Hoare triple theorems and separation logic [133] to describe the behavior of each instruction, and the model captures realistic details of ARM instructions, which we illustrate briefly. We begin by describing the semantics of ARM instructions provided by the Cambridge ARM model through its Hoare logic for realistically-modeled ARM instructions, and the Hoare logic rules provided by the Cambridge ARM model which we make use of in the logic approach in this dissertation.

4.1.1 Semantics and Hoare Logic for Realistically Modeled Instructions

The Cambridge ARM model decompiles each ARM instruction to a Hoare triple theorem of the form `(p) c (q)`, where `p` and `q` are predicates describing the state of the processor before and after executing code `c` respectively. 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`[1].

Then, the theorem `(p) c (q)` informally means that for a processor in a state satisfying `p` before running `c`, after running `c`, the processor will have state satisfying `q`. Processor state assertions `p` and `q` either assert the value of a given resource (e.g., value in a register), or they can be pure boolean assertions that capture the relationships between 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 they can be *assumptions* (labelled `cond`$(\cdot)$). While assumptions (i.e., `cond`) and pre-conditions

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

(i.e., precond) are logically equivalent in the Hoare logic of the Cambridge model, we differentiate between the two: when we use $\texttt{cond}(r)$ to indicate predicate r is a safety assertion in a Hoare triple theorem, we also add r to the hypotheses of the theorem.

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 3w } r3 \ * \ \texttt{aR 2w } r2 \ * \ \texttt{aPC } p \ * \ \texttt{aMEMORY } df\ f \ * \ \texttt{cond}((r3 \ \&\& \ 3w = \texttt{0w}) \wedge (r3 \in df))) \\
& \{(p, \texttt{0xE5832000w “str r2, [r3]”})\} \\
& (\texttt{aR 3w } r3 \ * \ \texttt{aR 2w } r2 \ * \ \texttt{aPC } (p + \texttt{4w}) \ * \ \texttt{aMEMORY } df\ ((r3 \mapsto r2)\ f))
\end{array}
$$

Figure 4.1. Example of a Hoare triple theorem, for the ARM instruction 0xE5832000 with mnemonic “str r2 [r3]”.

$$\vdash \quad \forall\, a\ b\ \cdot\ (a \mapsto b) = (\lambda f\ c\ \cdot\ \texttt{if } a = c \texttt{ then } b \texttt{ else } f\ c)$$

Figure 4.2. Definition of the “$\mapsto$” update operator in the HOL4 Update Theory.

As an example, consider the theorem for the ARM instruction 0xE5832000 (mnemonic “str r2 [r3]”), as shown in Figure 4.1. SPEC indicates that the theorem is a Hoare triple, while ARM_MODEL records the ARM-specific instruction semantics for the triple [32].

“aR 2w” and “aR 3w” are expressions that assert that a specified register stores the specified value, where “2w” and “3w” indicate the number of the specified register whose value is being asserted, and the suffix w indicates the register number is a fixed-width word. Then, the pre-state shows that: (i) the registers r2 and r3 contain the (symbolic) values $r2$ and $r3$ respectively, which each represent a fixed-width 32-bit word, (ii) the main memory contains the map f with domain df, and (iii) the program counter has some address p before running the instruction. After running the instruction, the values of registers r2, r3 remain unchanged, and the program counter advances to $p+4$. Also, “$\mapsto$” is the map-update operator, as defined in Figure 4.2, hence “r3 $\mapsto$ r2” indicates that the memory has been updated to store the value that was in register r2 at the address given by the value that was in register r3. The expression “$\texttt{cond}((r3 \ \&\& \ 3w = 0w) \wedge (r3 \in df))$” is an assertion that specifies our memory alignment requirement for writes to the address $r3$, and that $r3$ is in the domain of the memory map f.

“$*$” is the separating conjunction in Separation Logic [133] which asserts all other resources are unchanged. Note that $*$ in the Cambridge ARM model prevents assertions about repeated processor resources (e.g., the same register cannot be asserted about twice), but memory is asserted only once

because it is treated as a single map from memory addresses to stored values. Memory is represented as a map from 32-bit addresses to the byte stored at each address.

4.1.2 Composition rule in Hoare logic

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

$$\frac{\texttt{SPEC}\ x\ p\ c\ q}{\texttt{SPEC}\ x\ (p\ *\ r)\ c\ (q\ *\ r)}\ \texttt{FRAME}$$

Figure 4.3. Hoare logic's Compose and Frame rules in the Cambridge ARM model.

The Compose rule of Hoare logic [132] is shown above in Figure 4.3, which extends single instruction Hoare triple theorems to describe multiple instructions. Here, in "SPEC x p c q", "SPEC" is a relation indicating the presence of a Hoare triple, "x" indicates the parametrizable processor semantics that allows the Cambridge Hoare logic to support different machine architectures (but is always set to ARM_MODEL for our purposes) , and "p", "c", and "q" are the pre-state, code, and post-state of the Hoare triple respectively.

One detail of this rule critical to our analysis is that for code c_1 and c_2 such that c_2 runs immediately after c_1, the post-state of the theorem for c_1, q, must match the pre-state of the theorem for c_2, before the two single instruction theorems can be composed to form a single theorem describing the effects of the sequential execution of both instructions. One critical detail of this rule is that to apply the Compose rule to compose two Hoare triple theorems, the pre-state of the second theorem must be equal to the post-state of the first theorem. Conceptually, when instruction i_1 executes, followed by instruction i_2, as i_2 is executing immediately after i_1, the processor state just before i_2 executes is exactly the processor state after i_1 executes.

4.1.3 Pre-composition Tactic

A typical proof tactic for composing Hoare triple theorems for sequential instructions, i_1, i_2, with i_1 running immediately before i_2, into a single Hoare triple theorem, is given by the following steps:

1. Using the Frame rule (Figure 4.3), add machine state assertions in i_1's theorem, but not in i_2's theorem, to i_2's theorem;

2. Using the Frame rule, add machine state assertions in i_2's theorem, but not in i_1's theorem, to i_1's theorem;

3. Instantiate free variables in i_2 with the post-state machine resource values from i_1.

We call these steps the pre-composition tactic. This is similar to the "shift" operation described by Myreen et al. [32]. After carrying out the above theorem manipulation steps, the manipulated theorems i_1' and i_2' for both instructions will now have the post-state of i_1' matching the pre-state of i_2', allowing us to directly apply the Compose rule in Hoare logic.

For instance, consider the two instructions, i_1 ("`mov r3,r4`"), followed by i_2 ("`sub r2, r3, #16`"). We illustrate the use of the Compose rule to obtain a theorem describing the behavior of a program (or its fragment), $i_1 i_2$. The Hoare triple theorems for each of the two instructions are shown respectively in Figure 4.4.

$$\vdash \quad \texttt{SPEC ARM\_MODEL}$$
$$(\texttt{aR 3w } r3 \ast \texttt{aR 4w } r4 \ast \texttt{aPC } p)\ \{(p, \texttt{0xE1A03004w}\ \text{“}\texttt{mov r3, r4}\text{”})\}$$
$$(\texttt{aR 3w } r4 \ast \texttt{aR 4w } r4 \ast \texttt{aPC } (p + \texttt{4w}))$$

$$\vdash \quad \texttt{SPEC ARM\_MODEL}$$
$$(\texttt{aR 2w } r2 \ast \texttt{aR 3w } r3 \ast \texttt{aPC } p)\ \{(p, \texttt{0xE2432010w}\ \text{“}\texttt{sub r2, r3, \#16}\text{”})\}$$
$$(\texttt{aR 2w } (r3 - \texttt{16w}) \ast \texttt{aR 3w } r3 \ast \texttt{aPC } (p + \texttt{4w}))$$

Figure 4.4. Hoare triples for individual instructions before applying Compose rule.

Thus, in composing the two theorems i_1, i_2 in our above example, our pre-composition tactic will carry out the following steps on the theorems i_1, i_2:

1. Use the Frame rule to add `aR 2w` $r2$ to the Hoare triple theorem i_1 to get the Hoare triple theorem i_1' (as shown in Figure 4.5);

2. Use the Frame rule to add `aR 4w` $r4$ to the Hoare triple theorem i_2 to get the Hoare triple theorem i_2' (as shown in Figure 4.5);

3. Instantiate the value of p to $p + 4w$, and $r3$ to $r4$ in the Hoare triple theorem i_2' to get the Hoare triple theorem i_2'';

4. Apply Compose rule to Hoare triple theorems i_1', i_2'' to obtain the Hoare triple theorem in Figure 4.6.

$$\vdash \quad \texttt{SPEC ARM\_MODEL}$$
$$(\texttt{aR 2w } r2 \texttt{ * aR 3w } r3 \texttt{ * aR 4w } r4 \texttt{ * aPC } p)\ \{(p, \texttt{0xE1A03004w “mov r3, r4”})\}$$
$$(\texttt{aR 2w } r2 \texttt{ * aR 3w } r4 \texttt{ * aR 4w } r4 \texttt{ * aPC } (p + \texttt{4w}))$$

$$\vdash \quad \texttt{SPEC ARM\_MODEL}$$
$$(\texttt{aR 2w } r2 \texttt{ * aR 3w } r3 \texttt{ * aR 4w } r4 \texttt{ * aPC } p)\ \{(p, \texttt{0xE2432010w “sub r2, r3, \#16”})\}$$
$$(\texttt{aR 2w } (r3 - \texttt{16w}) \texttt{ * aR 3w } r3 \texttt{ * aR 4w } r4 \texttt{ * aPC } (p + \texttt{4w}))$$

Figure 4.5. Hoare triples for individual instructions after applying Frame rule, but before applying Compose rule.

$$\vdash \quad \texttt{SPEC ARM\_MODEL}$$
$$(\texttt{aR 3w } r3 \texttt{ * aR 4w } r4 \texttt{ * aPC } p \texttt{ * aR 2w } r2)$$
$$\{(p, \texttt{0xE1A03004w “mov r3, r4”}); (p + \texttt{4w}, \texttt{0xE2432010w “sub r2, r3, \#16”})\}$$
$$(\texttt{aR 2w } (r4 - \texttt{16w}) \texttt{ * aR 3w } r4 \texttt{ * aPC } (p + \texttt{8w}) \texttt{ * aR 4w } r4)$$

Figure 4.6. Composed Hoare triple theorem.

The pre-composition tactic prepares two suitable Hoare triples for reasoning about the effects of code on the same pre-state (i.e. pre-state of the first Hoare triple) by placing them in the same context (i.e. describing the effects of the code in both triples in terms of the pre-state variables of the first Hoare triple). Later, in §5.3, we will explain (i) how our local reasoning process for proving safety assertions is based on the pre-composition tactic described above, and (ii) how we can use global information in the pre-composition tactic to introduce global context to a Hoare triple for proving safety assertions.

4.1.4 Other Hoare Logic Rules

$$\frac{g \Rightarrow \texttt{SPEC } x\ p\ c\ r}{\texttt{SPEC } x\ (p \texttt{ * cond } g)\ c\ q}\ \texttt{SPEC\_MOVE\_COND}$$

$$\frac{\texttt{SPEC } x\ (p \texttt{ * cond } g)\ c\ q}{\texttt{SPEC } x\ (p \texttt{ * cond } g)\ c\ (q \texttt{ * cond } g)}\ \texttt{SPEC\_DUPLICATE\_COND}$$

Figure 4.7. Additional rules that have been proved in the Hoare logic in the Cambridge ARM model for manipulating pure boolean conditions.

Next, we describe some of the additional Hoare logic rules in the Cambridge ARM model that are relevant to our logic approach, as shown in Figure 4.7.

The first rule, `SPEC_MOVE_COND`, illustrates the relationship between pure boolean assertions (i.e., that mention only variables, but do not directly mention any machine state) in the pre-state of a Hoare triple, as marked by the "`cond`" syntactic label, and the hypotheses of the theorem. The `SPEC_MOVE_COND` rule makes explicit that pure boolean assertions "*g*" enclosed by the syntactic label "`cond`" in a Hoare triple are hypotheses of the Hoare triple.

The second rule, `SPEC_DUPLICATE_COND`, shows that pure boolean assertions that have been assumed to hold in the pre-state of the Hoare triple will still be assumed to hold in the post-state of the Hoare triple, because pure boolean assertions mention only variables but do not mention any machine state. The `SPEC_DUPLICATE_COND` rule appears counter-intuitive compared to typical Hoare logics, which use symbolic variables to directly represent the values of machine resources that are asserted. For instance, in a typical Hoare logic, the symbolic variable "*r2*" would represent the value held in register `r2`, and an instruction (with mnemonic) such as "`add r2,r2,#1`" would result in a post-state of "r2 := r2 + 1". However, in the Hoare logic of the Cambridge ARM model [31], the values held in machine resources are asserted using an assertion relation, such as "`aR`" for registers, "`aMEMORY`" for memory, and "`aS`" for status registers. Then, for the "`add r2,r2,#1`" instruction, the pre-state assertion for the value of register `r2` would be "`ar 2w` *r2*", while the post-state assertion would be "`ar 2w` (*r2* `+ 1w`)". Hence, symbolic variables in Hoare triples in the Cambridge ARM model are implicitly single static assignment (SSA) variables that are effectively initialized in the pre-state of each Hoare triple theorem. This is the key intuition that enables the `SPEC_DUPLICATE_COND` rule to work.

4.2 Design: The $\mathscr{L}_{LR}$ Program Logic

Next, we describe the design of the $\mathscr{L}_{LR}$ logic for reasoning about safety properties in ARM machine-code programs, that enables the automated generation of proofs, and we discuss the rationale behind our design decisions. Our program logic needs to fulfill three tasks:

1. Specify safety assertions for each instruction: A safety assertion of an instruction specifies the conditions that must be true before the instruction is executed for our safety properties to hold.

2. Ensure that the Hoare triple theorems for every instruction contain safety assertions, as introduced using pure boolean assertions in "cond" in the pre-state of the theorem.

3. Define, formally, the requirements for a program to possess our desired safety properties.

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

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

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

Figure 4.8. Logic rules for $\mathscr{L}_{LR}$. ms, cfi_1, cfi_2, cfi_3 are safety assertions for memory and control-flow isolation respectively. `MCSAt` is a syntactic label to group safety assertions.

4.2.1 Individual Instructions: Safety Assertion Specification

Figure 4.8 shows the `MEM_CFI_SAFE` rule for augmenting the Hoare triple theorem of a single instruction with its safety assertion (i.e., predicates ms, cfi_1, cfi_2 that capture our CFI safety properties as presented in §3.1.2) by including it as an assumption in the pre-state of the theorem using the "cond" relation. This rule overcomes the challenge of reasoning about safety properties at every instruction using Hoare logic. We add our safety assertions as a pure boolean condition to the pre-state of an instruction's Hoare triple. Then, when the Compose rule (§4.1.2) is applied to compose theorems of multiple instructions, the pre-states of successor instructions (q in the Compose rule) will be hidden, thus hiding our augmented safety assertions. Also, safety assertions that hold can be simplified to true and eliminated from the Hoare triple. Thus, for a Hoare triple describing a sequence of instructions, we cannot tell if the theorem contains safety assertions for every instruction.

The `MEM_CFI_SAFE` rule overcomes this challenge by ensuring that the Hoare triple for every instruction has been augmented with its safety assertions. This rule has two features. First, `MEM_CFI_SAFE` can be instantiated only from single instruction Hoare `SPEC` theorems, because code c in the `SPEC` theorem in the rule antecedent admits only a single instruction with the machine word `ins` located at address `offset`. The second rule, which generates the safe `MEMCFISAFE` theorem, `MEM_CFI_SAFE_COMPOSE`, does not admit Hoare triple `SPEC` theorems, and only allows the composition of `MEMCFISAFE` theorems. Second, the `MEM_CFI_SAFE` rule can be instantiated only when the

pre-state is augmented with our safety assertion (as given by the safety properties for CFI presented in §3.1.2), the pure boolean conjunction, $ms \wedge cfi_1 \wedge cfi_2$, in its pre-state. Thus, the `MEMCFISAFE` relation indicates the resulting Hoare triple has been augmented with our safety assertion in its instruction pre-state. `MCSAt` is a syntactic relation that associates our safety assertion, $ms \wedge cfi_1 \wedge cfi_2$, with the address *offset* which the assertion applies to. We also add the safety assertions ms, cfi_1, cfi_2 to the hypotheses of the theorem, to indicate that they are undischarged.

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

4.2.2 Sequential Code Blocks

Next, we describe how we obtain safety-augmented Hoare triple theorems for basic blocks of sequential code (which we refer to as safe basic block theorems). A basic block is a sequence of instructions which execute sequentially, with a single entry instruction and a single exit instruction. The two rules (Fig. 4.8) we need for building safe basic block theorems are `MEM_CFI_SAFE_COMPOSE`, and `MEMCFISAFE_FRAME` (proved using the Frame rule in separation logic). These two rules allow us to inductively build up a safe basic block theorem from safety theorems for individual instructions. The process of building up a safety theorem for a basic block of sequential code is the same as that of composing Hoare triple theorems (§4.1.2), except that only safety-augmented Hoare triple theorems can be composed. This process is repeated recursively for every instruction in a basic block to obtain a single safe theorem for the basic block. Our safe basic block theorems have the same semantics as Cambridge ARM Hoare triples, as proved in §4.2.1.

$$
\begin{array}{ll}
\vdash & \texttt{FUN\_SAFE}(addr, NODES, FUNCS, CFG_{pred}, CFG_{succ}, ICFG_{callpred}, ICFG_{callsucc}, \\
& \qquad ICFG_{retpred}, ICFG_{retsucc}, assns_{entry}, postcond_{exit}, prestate, poststate) \Leftrightarrow \\
(& (\forall node \cdot node \in NODES \Rightarrow (\texttt{min}(node, addr) = addr)) \\
\wedge & (\forall min \cdot min \in NODES \Rightarrow (CFG_{pred}(min) = \emptyset \wedge ICFG_{callpred}(min) = \emptyset \wedge ICFG_{retpred}(min) = \emptyset) \\
& \Rightarrow (\forall node \cdot (node \in NODES \Rightarrow node \neq min) \Rightarrow (\texttt{min}(node, min) = min)) \\
& \quad \Rightarrow \exists pd_1, x, c_1, p_1, q_1 \cdot \texttt{HOARE\_WITH\_ASSERT}(pd_1, assns_{entry}, min, node, x, c_1, p_1, q_1) \wedge \\
& \qquad (prestate = \texttt{aPC}\ min * p_1)) \\
\wedge & (\forall out \cdot out \in NODES \Rightarrow (CFG_{succ}(out) = \emptyset) \\
& \Rightarrow (\forall funcnode \cdot (funcnode \in FUNCS \Rightarrow out \notin ICFG_{callsucc}(funcnode))) \\
& \quad \Rightarrow \exists pd_1, assn_1, node, x, c_1, p_1, q_1 \cdot \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, out, node, x, c_1, p_1, q_1) \wedge \\
& \qquad (poststate = q_1) \wedge (pd_1 \Rightarrow postcond_{exit})) \\
\wedge & (\forall node, pred, succ \cdot \\
& \qquad (node \in NODES) \Rightarrow \\
& \qquad (pred \in CFG_{pred}(node)) \Rightarrow \\
& \qquad ((succ \in CFG_{succ}(node)) \vee (succ \in ICFG_{callpred}(node))) \Rightarrow \\
& \exists pd_1, assn_1, x, c_1, p, q, pd_2, assn_2, c_2, r \cdot \\
& \quad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, pred, node, x, c_1, p, q) \wedge \\
& \quad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, succ, x, c_2, q, r) \wedge (pd_1 \Rightarrow assn_2)) \\
\wedge & (\forall node, succ \cdot node \in ICFG_{callsucc}(succ) \Rightarrow succ \in ICFG_{callpred}(node) \Rightarrow \\
& \exists pd_1, assn_1, x, c_1, p, q, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_2, pd_2, r \cdot \\
& \quad \texttt{HOARE\_WITH\_ASSERT}(pd_1, assn_1, node, succ, x, c_1, p, q) \wedge \\
& \quad \texttt{FUN\_SAFE}(succ, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_2, pd_2, q, r) \wedge (pd_1 \Rightarrow assn_2)) \\
\wedge & (\forall node, pred, succ \cdot \\
& \qquad (node \in ICFG_{retsucc}(pred)) \Rightarrow \\
& \qquad (pred \in ICFG_{retpred}(node)) \Rightarrow \\
& \qquad ((succ \in CFG_{succ}(node)) \vee (succ \in ICFG_{callpred}(node))) \Rightarrow \\
& \exists pd_1, assn_1, x, c_2, p, q, pd_2, assn_2, r, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6 \cdot \\
& \quad \texttt{FUN\_SAFE}(pred, nodes, funcs, cfg_1, cfg_2, cfg_3, cfg_4, cfg_5, cfg_6, assn_1, pd_1, p, q) \wedge \\
& \quad \texttt{HOARE\_WITH\_ASSERT}(pd_2, assn_2, node, succ, x, c_2, q, r) \wedge (pd_1 \Rightarrow assn_2)) \quad)
\end{array}
$$

Figure 4.9. FSI rule: Judgment for Interprocedural Function Safety

4.2.3 Function Judgment for Local Reasoning

Global vs. Local Reasoning

In a typical correctness proof for a program using Hoare logic, we would repeatedly apply the Compose rule to the Hoare triple for every instruction in the program to obtain a single Hoare triple describing the entire program. This is a "global reasoning" process which identifies the final values of all registers, main memory, etc. at the end of the program's execution. In the presence of loops and function calls, loop invariants and pre- and post-conditions for functions will need to be manually provided.

For safety assertions to hold in a program, we only need to ensure that the safety assertions for each instruction hold locally at that instruction. For the safety assertions at instruction i_2 to hold, we consider every instruction i_1 that can execute immediately before i_2. The machine-resource values in the post-state of each i_1 must satisfy the safety assertions at i_2. This is analogous to the pre-composition process (§4.1.2). As long as the machine-resource values in the post-states of predecessor instructions i_1 enable the safety assertion at i_2 to be true, the safety assertion holds. Also, any pure boolean condition from the post-state of predecessor instructions i_1 will also apply to the pre-state of instruction i_2. Hence, safety properties hold on a per-instruction basis. To check if a safety assertion holds for an instruction i_1, we only need to perform "local reasoning" by considering the post-state and boolean conditions of all predecessor instructions of the instruction i_1.

Safe Function Judgment

We define the `FUN_SAFE` rule (Fig. 4.9). The purpose of this rule is to encode the requirements for our safety properties that imply CFI to hold at every instruction in a machine-code program. Note that while the `FUN_SAFE` rule states that our safety properties for CFI hold for every instruction of a program, it does not directly prove that the program possesses CFI. To formally prove that a program possesses CFI, in addition to proving the `FUN_SAFE` theorem for the program, we need to prove that our safety properties imply CFI, although it is not in the scope of this dissertation.

Informally speaking, the `FUN_SAFE` rule encodes what it means for a function to be safe. This rule encodes our "local reasoning" process for verifying that safety assertions hold. Thus, proving that the machine-code of a given function is safe involves proving that the `FUN_SAFE` theorem holds for the function.

First, we rearrange `MEMCFISAFE` theorems to form `HOARE_WITH_ASSERT` theorems, which make explicit the hypotheses (i.e., undischarged safety assertions) of the theorems, and rearrange machine resource expressions into tuples for pattern-matching.

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

Figure 4.10. `HOARE_WITH_ASSERT` rule for rearranging `MEMCFISAFE` safety-augmented Hoare triples into a form that can be pattern-matched against.

A function is comprised of basic blocks of instructions in the function. In a function's intra-procedural control-flow graph (CFG), nodes are basic blocks of the function's instructions, while edges are control transfers within the function. In a function's inter-procedural CFG, the nodes are (i) basic blocks that call other functions, (ii) basic blocks that are return-sites from callee functions, and (iii) callee functions, while edges are function calls or returns. To formally specify the requirements for a function to be safe, we consider the safety assertions that must be discharged at each edge in both the intra- and inter-procedural CFGs. We walk through each of the 6 conjunct clauses in the `FSI` rule in Figure 4.9.

Arguments to the `FUN_SAFE` relation

The `FUN_SAFE` relation is parameterized by the function address $addr$, a set of addresses of basic blocks in the function $NODES$, a set of addresses of callee functions $FUNCS$, and 6 maps CFG and $ICFG$ specifying the predecessors and successors of edges in the function's intra- and inter-procedural CFGs. `FUN_SAFE` also records, for a function, the safety assertions $assns_{entry}$, the conditions that hold at its exit $postcond_{exit}$, and the machine resource pre-state $prestate$ and post-state $poststate$. These maps of CFG predecessors and successors are explicitly stated in the final `FUN_SAFE` theorem proved, and can be directly inspected to ensure they correspond to the input program.

Function entry and exit specifications

The first clause states that the address of the function is the lowest basic block address for the function. While this may not be true in general, we assume that this is the case for our input programs, and our assumption has been borne out by our `gcc`-generated machine-code programs. The second clause states that the safety assertions $assns_{entry}$ and pre-state $prestate$ of the function are specified by the entry basic-block of the function. The third clause states that the function's guaranteed exit condition $postcond_{exit}$ and post-state $poststate$ are specified by the exit basic-block of the function.

Intra-procedural safety requirements

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

at the source of the edge, i.e., $(pd_1 \Rightarrow assn_2)$, where pd_1 represents the pre-condition of the predecessor CFG node along some CFG edge, and $assn_2$ represents the safety assertion of the successor CFG node along the same CFG edge. Also, in the spirit of the Hoare Compose rule, we require that the post-state of the predecessor instruction q, is equal to the pre-state of the successor instruction. This ensures that when we reason about two CFG nodes that are connected by a single edge in the CFG, the post-state of the predecessor node exactly matches the pre-state of the successor node, and is line with typical Compose rules in Hoare logics [132, 31], enabling us to describe sequentially executed pieces of code (albeit without actually producing a final "composed" theorem for the two composed code fragments).

Inter-procedural safety requirements

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

Compositional reasoning for functions

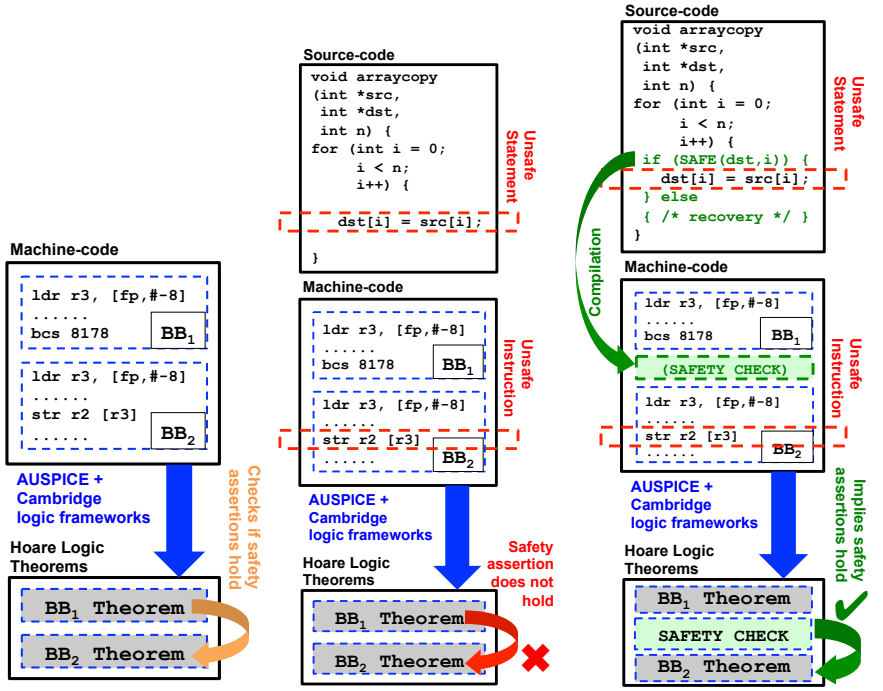
Although the `FSI` rule appears to be recursively defined without a base case, this rule actually collapses to include only the first four clauses for functions that do not call any other functions. This implies that our safety property proofs require the CFG of the program to have no cycles, i.e. we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

4.2.4 Safety Theorem for a Program

Thus, a safety proof for a program consists of a `FUN_SAFE` theorem for the entry function of the program, as constructed from the `FUN_SAFE` rule. The `FUN_SAFE` theorem for each function then requires `MEMCFISAFE` theorems for each of its basic blocks, as constructed using the `MEM_CFI_SAFE` rule for individual instructions, and the `MEM_CFI_SAFE_COMPOSE` rule for basic blocks. The construction of the `FUN_SAFE` theorem for each function F also requires `FUN_SAFE` theorems to be available for each callee function called in the function F.

While the top-level safety proof for a program consists of just the FUN_SAFE theorem for its entry function, the safety proof comprises safety theorems for all the functions reachable from the program's entry function, as well as safety theorems for all the instructions and basic blocks (i.e., MEMCFISAFE theorems) in the program that are reachable from the entry function of the program.

4.3 Provability of CFI in Machine-code for Source-code-Enforced CFI



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

Figure 4.11. How PCFIRE-C's prescribed source-code safety-checks yield machine-code programs whose CFI can be proved using $\mathscr{L}_{LR}$. BB_1 and BB_2 represent basic block addresses.

Next, we describe how the source-code safety-checks prescribed by our PCFIRE-C tool (§3.2) result in CFI that can be automatically proved at the machine-code level using the $\mathscr{L}_{LR}$ program logic presented in this chapter.

First, in our logic approach, the CFI safety properties to be proved at each instruction (as defined in §3.1.2) are instantiated at each instruction's Hoare triple theorem as assertions to be proved. Then, a proof search is carried out at the intra- and inter-procedural level (described later in §5.1): for each basic block, the proof automation algorithm in AUSPICE checks if the CFI safety assertions at every

instruction in each basic block can be discharged given the pre-conditions of all its predecessor basic blocks, as illustrated in Figure 4.11(a).

Second, for C statements (and their corresponding compiled instructions) with potentially unsafe operations, in the absence of our safety-checks as prescribed by PCFIRE-C, the Hoare triples for these instructions and their basic blocks will have safety assertions that cannot be automatically discharged by its predecessor blocks, leading to a failure in the safety proof, as illustrated in Figure 4.11(b).

With the prescribed safety-checks from PCFIRE-C inserted, the (recompiled) machine-code at the predecessor blocks of each potentially unsafe instruction will now have the post-conditions (or equivalently, `preconds`, by using the `SPEC_DUPLICATE_COND` rule as described in §4.1.4) sufficient to imply that the safety assertions at each unsafe instructions' Hoare triples hold. This is because: (A) the memory-address checks in PCFIRE-C's prescribed safety-checks have bounds values that match the bounds that are instantiated in Properties 1 and 2 of the $\mathscr{L}_{LR}$-instantiated safety assertions, and (B) the logic expressions for the checked and written addresses in (i) the safety-check, and (ii) the suspect C 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 in turn enables the CFI safety proof to succeed. This is illustrated in Figure 4.11(c). Note that our prescribed safety-checks at the C level must be carefully constructed to enable our machine-code safety proof to succeed, and we discuss these considerations in our prescribed safety-checks in §8.3.

To recap, the safety-checks prescribed by PCFIRE-C are constructed based on the CFI safety properties that are: (i) defined in §3.1.2, and (ii) instantiated as safety assertions at each instruction in $\mathscr{L}_{LR}$. Hence, we construct the source-code safety-checks in PCFIRE-C such that when compiled to machine-code, the Hoare triples of the machine-code of the source-code safety-checks provide exactly the logic post-conditions (or equivalently, `preconds`, as explained above) needed to discharge the safety assertions at unsafe instructions (as compiled from suspect statements).

4.4 Discussion: Soundness and Correctness of Proof Rules

4.4.1 Soundness of Proof Rules

The proof rules in $\mathscr{L}_{LR}$ for single instruction (§4.2.1) and basic block (§4.2.2) safety are sound, because we derive our `MEM_CFI_SAFE`, `MEM_CFI_SAFE_COMPOSE`, and `MEMCFISAFE_FRAME` proof

rules from the proof rules in the Cambridge ARM model, which Myreen et al. have shown to be sound [31]. Also, using the HOL4 proof assistant to define our proof rules further ensures they are sound, as our proof rules are proved and admitted as theorems in HOL4. In addition, we proved (§4.2.1) that safe single instruction and basic block theorems in $\mathscr{L}_{LR}$ derived from our proof rules have the same instruction semantics as the ARM machine-code semantics defined by the trustworthy, validated Cambridge ARM model [131, 32].

4.4.2 Correctness of Safety Rule

Next, we give a brief, informal argument of the correctness of our proof rule for safe programs. The `FUN_SAFE` theorem (Figure 4.9) can be proven for a program if and only if safety assertions are specified for every instruction, and if these safety assertions hold before that instruction begins executing (except for the first instruction, which relies on the OS to correctly initialize the processor state for the program). We argue this by Structural Induction on the Control-Flow Graph (CFG) of a program. The CFG of a function in a program consists of nodes and edges. Each node represents either (i) a basic block of a linear sequence of instructions (whose control can transfer out of the basic block only at the end of the basic block) in the function at a given address, or (ii) a callee function that is invoked in the function. Then, an edge in the CFG represents a transfer of control from one node in the CFG (which can be a basic block in the function, or a callee function), to another node in the CFG. A callee function node is associated with a single `FUN_SAFE` theorem which specifies the safety of the callee function. A basic block node (in the function) is associated with one `HOARE_WITH_ASSERT` triple for each possible target that control can transfer to after the basic block has been executed. Thus, for safety to hold at each CFG node, the safety-assertions at all associated `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems for that node must be discharged by the pre-conditions of all the associated `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems for all predecessor nodes of the node.

Base Case. The `MEM_CFI_SAFE` rule (§3.1 in [128]) ensures every instruction's theorem contains our safety assertions (§2.2 in [128]). The `MEM_CFI_SAFE_COMPOSE` rule ensures every basic block's theorem is built up only from single-instruction theorems with added safety assertions. The requirement that post-states of predecessor theorems and pre-states of successor theorems must be equal in `MEM_CFI_SAFE_COMPOSE` ensures every basic block's theorem accumulates the safety assertions for every composed safe instruction theorem. Then, for a program with only a single instruction or

basic-block, if the OS correctly initializes the processor state, the safety assertions will hold for the single instruction or single basic block.

Inductive Case. We take the CFG of a function, G, and partition its vertices into a single vertex, g, and all other vertices, G'. By the Inductive Hypothesis, the `FUN_SAFE` theorem holds for G'. Then, consider the edges E connecting G' to g. In the absence of function pointers and unstructured jumps (`longjmp`), the edges E are either (i) intra-procedural control-flow transfers between basic blocks in the function, (ii) function calls from a basic block in the function to a callee function, or (iii) function returns from a callee function to a basic block in the function.

Then, for `FUN_SAFE` to be true, the fourth to sixth conjunct clauses of the `FUN_SAFE` rule must be true. We will see how the 4th to 6th conjuncts cover all possible types of control-flows to and from the node g, so that the pre-conditions of the theorems of all predecessor vertices to g in the CFG discharge the safety assertions at g, making the safety assertions at g hold, for any type of possible control-flow transfer to g for all `FUN_SAFE` or `HOARE_WITH_ASSERT` theorems associated with g.

In the case of intra-procedural control-flow transfers to g, where g is a basic block within the function being analyzed, g is associated with one `HOARE_WITH_ASSERT` triple for each possible control-transfer target from node g. These targets can either be intra-procedural targets (i.e., the successor node to g is a basic block within the function), or inter-procedural targets (i.e, the successor node to g is a callee function outside the function). The 2nd precedent of the 4th conjunct, $\forall pred \cdot pred \in CFG_{pred}(node)$ ensures all predecessors to g are considered. The 1st disjunct of the 3rd precedent, $\forall succ \cdot succ \in CFG\_succ(node)$ ensures all possible intra-procedural successor nodes to g are considered. The 2nd disjunct of the 3rd precedent, $\forall succ \cdot succ \in ICFG_{callpred}(node)$ ensures all possible inter-procedural successor nodes to g (i.e., callee functions jumped to after g) are considered. Thus, all possible predecessors to the CFG node g are considered, and all possible control-transfer targets from CFG node g are considered.

In the case of inter-procedural control-flow function-calls to g (i.e., g is a callee function, as in the 5th conjunct), `FUN_SAFE` theorems for callee functions g are indexed only by the address of the function, but not by the address of the return-site from the function. Hence, we only need to consider all call-sites of the function. The two precedents in the 5th conjunct (i.e., $\forall node, succ \cdot node \in ICFG_{callsucc}(succ) \Rightarrow succ \in ICFG_{callpred}(node)$) ensure that all call-sites within the function to the callee function are considered.

In the case of inter-procedural control-flow function-returns to g (i.e., g is a basic block within

the function, as in the 6th conjunct, and its predecessor is a callee function), the first two precedents of the 6th conjunct, $\forall node, pred \cdot node \in ICFG_{retsucc}(pred) \Rightarrow pred \in ICFG_{retpred}(node)$, ensure that the function from which control is returned to node g is considered. Then, the two disjuncts in the 3rd precedent, $succ \in CFG_{succ}(node) \vee succ \in ICFG_{callpred}(node)$, ensure respectively that the `HOARE_WITH_ASSERT` triples for (i) all possible intra-procedural successor nodes to g, and that (ii) all possible inter-procedural successor nodes to g, are considered.

Thus, our `FUN_SAFE` rule ensures that we have captured all the possible control-flow transfers in a machine-code program. For `FUN_SAFE` to be correct, we require correct CFG predecessor and successor maps, which are straightforward to compute without function pointers and unstructured jumps.

4.5 Summary

This chapter described the $\mathscr{L}_{LR}$ program logic that we developed in this dissertation [128] for reasoning about safety-properties in machine-code programs. The $\mathscr{L}_{LR}$ program logic is formalized in Higher Order Logic, and mechanized in the HOL4 proof assistant. It forms the foundation of the logic approach in this dissertation, and has been designed to enable the proof automation techniques that we will discuss next in §5.

We began by describing the Cambridge ARM model [131, 32, 31], which $\mathscr{L}_{LR}$ builds on. First, we make use of the semantics of individual instructions from the Cambridge model. Then, we described the logic rules in $\mathscr{L}_{LR}$ for single-instruction (`MEM_CFI_SAFE`) and basic block safety (`MEM_CFI_SAFE_COMPOSE`) for ensuring that desired safety properties are asserted at every instruction. Next, we described the logic rule for whole-program safety at the function level (`FUN_SAFE`) that enables "local-reasoning" for safety properties at each program point, without requiring a whole-program proof that would require (manually specified) loop invariants and other inputs. The `FUN_SAFE` rule is the key building block of our proof automation in the AUSPICE framework (§5).

Then, we described how we can use the $\mathscr{L}_{LR}$ program logic that we developed to automatically prove safety properties for machine-code programs, whose safety-checks were inserted at the source-code level based on the prescriptions generated by our PCFIRE-C tool (§3.2). This is because both the construction of the source-code safety-checks and the instantiation of the machine-code safety assertions are based on the same set of safety properties as defined in §3.1. In addition, the same source-code expressions are used for the memory address being checked in each safety-

check, and for the memory address being written to. Also, the bounds values for each safety-check match the safe address ranges as specified by our safety properties.

We also discussed the soundness and correctness of the `FUN_SAFE` rule for ensuring that the safety properties we defined in §3 hold. We gave a brief argument of the soundness of our proof rules based on the soundness of the Cambridge ARM model and the HOL4 prover, and we constructed a sketch of the proof of correctness of the `FUN_SAFE` rule for ensuring that our safety properties hold. This in turn ensures that our safety proofs imply that CFI holds, based on our argument in §3.

Chapter 5

Automation of Safety-Property Proofs

This chapter describes how safety-property proofs in the $\mathscr{L}_{LR}$ program logic can be automated in our AUSPICE framework [128], which provides the logic approach in this dissertation. We first describe the overall proof automation workflow in AUSPICE (§5.1). Next, we describe how safety properties can be automatically specified at the single-instruction level (§5.2). Then, we describe the Selective Composition proof tactic, which is an important building block for the automated proof generation in AUSPICE (§5.3). Next, we describe the process by which safety property proof obligations are automatically discharged in AUSPICE using abstract interpretation (§5.4). Finally, we discuss a number of auxiliary challenges with automating our safety-property proofs, and how we overcame these challenges (§5.5).

5.1 Proof Automation Framework

We begin by describing the overall implementation of our automatic safety property proof generation workflow in our AUSPICE framework. AUSPICE consists of 128 lines of HOL4 definitions and 11.8 KLOC of proof scripts in ML. Algorithm 5.1 summarizes the overall workflow of the AUSPICE safety property proof process. First, AUSPICE constructs basic blocks and extracts function boundaries from the machine-code of the program (Line 12). Next, AUSPICE obtains the Hoare triple theorems from the Cambridge ARM model for each machine-code instruction (Line 13), adds safety assertions to the Hoare triple theorem for each instruction (§5.2), and composes the individual instructions' theorems into a single Safe Basic Block theorem for each basic block (Line 15) using the `MEM_CFI_SAFE_COMPOSE` rule (§4.2.2).

AUSPICE's proof process takes place on a per-function basis beginning from the entry-function of the program. For each function, all callee functions called by that function are analyzed before the

function itself is analyzed (Line 3). Next, AUSPICE applies the Selective Composition tactic (§5.3) to the safe basic block theorems to propagate branch conditions and function prologue information to the appropriate theorems for the function (Lines 4 and 5). The main process for discharging safety proof obligations is the `SafetyAssertionAnalysis` function (Line 6), which implements the proof search process using abstract interpretation [134] (§5.4). Then, the results of this analysis are applied to each of the basic blocks' theorems, and the `FSI_rule` function (Line 8) generates the `FUN_SAFE` safety theorem for the target function being proved to be safe.

We discuss a number of auxiliary challenges encountered in the workflow of `AUSPICE` as well. In §5.5.1, we discuss how we support reasoning for machine-code compiled from `switch` statements, as handled in `CambridgeARM_GetInstrModel` (Line 13). In §5.5.2, we discuss how the `FSI_rule` (Line 8) for generating safe function theorems performs "memory reconciliation" to support our local use of global information (§5.3.2).

Algorithm 5.1 Overall AUSPICE Workflow

1: **function** SAFEFUNCTIONANALYSIS(*function_name*, *bb_safe_thms* `list`)
2: (*cfg*, *func*) ← ∀*i* ∈ *bb_safe_thms* · COMPUTECFGANDCALLEES(*i*) ▷ Compute Control-Flow Graph for *function_name*
3: *func_safe* ← ∀*callee* ∈ *func* · SAFEFUNCTIONANALYSIS(*callee*, *bb_safe_thms*)
4: *bb_safe_thms* ← SC-FWDPROPAGATE-BRANCHCONDS(*bb_safe_thms*, *cfg*)
5: *bb_safe_thms* ← SC-FWDPROPAGATE-FUNCPROLOGUE(*bb_safe_thms*, *cfg*)
6: *assertion_info* ← SAFETYASSERTIONANALYSIS(*bb_safe_thms*, *func_safe*, *cfg*)
7: *bb_safe_thms* ← AUGMENTTHEOREMS(*bb_safe_thms*, *func_safe*, *assertion_info*)
8: *safety_theorem* ← FSI_RULE(*bb_safe_thms*, *func_safe*, *cfg*)
9: **return** *safety_theorem*
10: **end function**
11: **function** AUSPICE((*addr*, *instr*) `list`) ▷ List of machine-code instructions
12: (*bb* `list`) ← COMPUTEBASICBLOCKS(*instr*) ▷ Compute basic blocks in program
13: (*bb_instr_thms* `list`) ← ∀*instr* · CAMBRIDGEARM_GETINSTRMODEL(*instr*) ▷ Obtain Hoare triple theorem for each *instr* in each *bb*
14: (*bb_safe_instr_thms* `list`) ← *map* (λ*x*.ADDSAFETYASSERTIONS(*x*)) *bb_instr_thms*
15: *bb_safe_thms* ← *map* (λ*x*.COMPOSESAFEINSTRS(*x*)) *bb_safe_instr_thms*
16: **return** SAFEFUNCTIONANALYSIS(*main*,*bb_safe_thms*)
17: **end function**
18: AUSPICE(*program*)

5.2 Automatic Safety Property Specification

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

logic theorem from the decompiler as shown in Figure 5.1.

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

Figure 5.1. Hoare triple theorem for instruction in proof automation example.

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

5.3 Selective Composition Proof Tactic

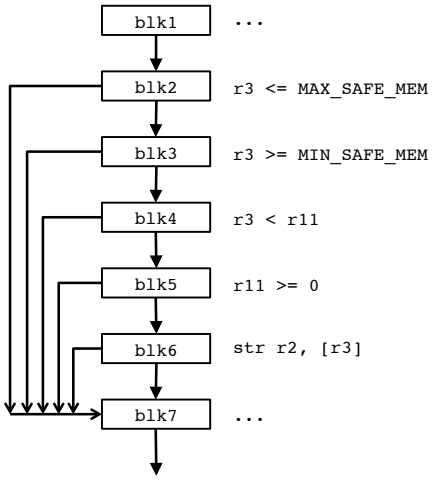


Figure 5.2. Possible structure for program with safe "`str r2 [r3]`".

Next, we discuss the steps for automatically proving that safety properties hold using $\mathscr{L}_{LR}$. After augmenting single instruction theorems with safety assertions (§4.2.1) and obtaining safe basic block theorems (§4.2.2), we need to prove that the antecedents in the `FSI` rule (Figure 4.9) hold. Each of the top-level conjuncts of `FSI` requires either a `HOARE_WITH_ASSERT` theorem for safe basic blocks or a `FUN_SAFE` theorem for safe functions. We also need to prove that the pre-condition pd_1 of each predecessor CFG node discharges the safety assertion $assn_2$ in the successor CFG node.

The Selective Composition proof tactic prepares our `HOARE_WITH_ASSERT` and `FUN_SAFE` theorems for our proof automation process. This proof tactic is used in two modes: (i) to carry out forward propagation of branch-conditions from compound `if`-statements whose `if` conditions contain a conjunction of multiple simple logical conditions, and (ii) to make global information available from function prologues available at local nodes in the CFG of a function (and its Hoare triple theorems). The propagation of information for both branch-conditions and for information from function prologues needs to be "selective" so as to avoid needing to perform "global" reasoning, which would impede our proof automation due to the requirement of manually-specified information such as loop invariants (§4.2.3).

5.3.1 Forward propagation of branch conditions

From §5.2, we can see that the safety assertion at each instruction contains three conjuncts. In a safe program, for the theorem of a given instruction i_2, its predecessor (safe basic block or function) theorem i_1 should have a pre-condition that implies that the safety assertion of i_2 holds. Observe that the safety assertion for each instruction has three conjuncts, and each of the range conjuncts (ms and cfi_1 in §5.2) is specified by two conjuncts: one each for the lower and upper bounds of the valid memory locations written to. Thus, the safety assertion at each instruction comprises multiple conjuncts. However, in a machine-code program, each basic block can only carry out one of the "elementary" arithmetic comparison operations (one of $<$, $>$, $\leq$, $\geq$, etc.), because each `cmp*` instruction is a branch and will mark the end of the basic block it belongs to. Hence, information from multiple predecessor basic blocks is required to discharge the safety assertion at each instruction.

In §4.2.3, we noted that we must use a *local reasoning* process to ensure our proof process is automatic, because global reasoning would require manually-specified information. However, our safety assertions contain multiple conjuncts, whereas each basic block in machine-code can provide only one conjunct in its pre-condition. This is because given a compound condition in an `if`-statement (i.e., the `if` condition comprises multiple conditions that are combined with a

Algorithm 5.2 Selective Composition: Branch-condition Forward Propagation

```
1:  function SC-FwdPropagate-BranchConds(bb_safe_thms list)
2:    info map ← ∅                          ▷ Conditions to propagate to each CFG node
3:    procedure PropagateOneStep(info map, last_info map, cfg)
4:      for all node ∈ cfg do
5:        curr_node_preds ← FindPreds(cfg,node)
6:        pred_preconds ← (map (λx.GetThmPreconds(x)) curr_node_preds)
7:        last_info_preconds ← (map (λx.last_info[x]) curr_node_preds)
8:        if length(curr_node_preds) == 1 then
9:          info[node] = pred_preconds ⋃ last_info_preconds
10:       end if
11:     end for
12:   end procedure
13:   repeat
14:     last_info ← info
15:     info ← PropagateOneStep(info, last_info, cfg)
16:   until last_info == info
17:   bb_safe_thms′ ← AugmentTheorems(bb_safe_thms,info)
18:   return bb_safe_thms′
19: end function
```

logical operator such as "`&&`"), each condition is checked using a branching instruction (such as `cmp` followed by `bhi` or `bls`). As a result, the basic block for that condition is effectively terminated by the branching instruction. This is as illustrated in Figure 5.2.

To enable our proof process to use pre-conditions from predecessors that are more than one edge away from a given basic block in the program's CFG, we selectively "propagate" the pre-conditions of basic blocks forward. We call this process "selective composition", where we apply the pre-composition tactic (§4.1.3) forward to successor theorems under certain conditions.

To illustrate the process of selective composition, consider, for example, the store instruction "`str r2 [r3]`". Figure 5.2 shows the CFG of the possible structure of the basic blocks in a program with safety checks to ensure that the store instruction is safe. Then, we need the pre-conditions from basic blocks $blk_2, blk_3, blk_4, blk_5$ to be available at blk_5 to discharge the safety assertion at blk_6. At each of the nodes $blk_2, blk_3, blk_4, blk_5$, there are two Hoare triple theorems: one where each blk_i executes blk_{i+1} next (for $i \in \{2,3,4,5\}$), and one where the safety check fails, and each blk_i goes on to execute blk_7. However, we do not compose $blk_2, blk_3, blk_4, blk_5$ to form a single Hoare triple theorem, because the resulting block of code will have multiple exits, which is not captured by our safe basic block theorem (the `MEM_CFI_SAFE_COMPOSE` rule), which only admits single-exit blocks. Instead, we iteratively apply the pre-composition tactic (§4.1.3) for basic blocks $blk_2, blk_3, blk_4, blk_5$.

This lets us place the analysis of the machine-code in blocks $blk_2, blk_3, blk_4, blk_5, blk_6$ in the context of the pre-state values of machine resources in blk_2. This then allows us to discharge the safety assertion at blk_6 with the combined pre-conditions of $blk_2, blk_3, blk_4, blk_5$ at blk_5. We call this process "selective composition" because we carry out the pre-composition process without applying the composition rule. Note that this selective composition process succeeds only when the target basic block which the pre-conditions are being propagated forward to have only one predecessor basic block. Only then is the pre-condition from the predecessor block blk_i the only pre-condition that will apply at the successor block blk_{i+1}. Note, also, that this process is enabled by the fact that symbolic variables in the Hoare logic in the Cambridge ARM model, which $\mathscr{L}_{LR}$ builds on, are effectively single-static assignment (SSA) variables.

Algorithm 5.2 describes the Selective Composition tactic for the forward propagation of branch-conditions in pseudocode. The tactic uses a fixed-point intra-procedural static-analysis over the Hoare triple theorems of a function. The static-analysis identifies branch conditions to propagate forward from each theorem to its successor theorems (Lines 3 to 16; `FindPreds` returns the predecessors for a given node in the CFG of the function, while `GetThmPreconds` returns the pre-conditions for a given Hoare triple theorem). The analysis also ensures that branch-conditions are propagated forward only when the target node has only one predecessor in the CFG (Line 8). Then, the tactic uses the results of the static-analysis and adds the required branch conditions to each theorem using the Frame Rule (Line 17), and returns the augmented theorems.

5.3.2 Local use of global information

Next, we describe the second instance of *selective composition*. Recall that we require the address of each instruction executed to be within the `text` section of the program. The address of the next instruction to be executed can be statically determined at every point of the program except where a function returns to its caller. Consider a typical machine-code instruction for returning from a function call "`pop {pc}`". Control is being returned from the function by restoring the saved link register value from the stack to the program counter. The instruction will be specified by the Hoare triple theorem shown in Figure 5.3.

Here, "`aMEMORY df f`" is an assertion that the main memory is represented by the map f which when applied to an address $addr$, returns the word stored at $addr$, and df is a set specifying the address domain of f. Thus, in the post-state of this instruction, we can see that the next instruction to be executed is at address $f\,r13$. However, the memory map f does not contain any information

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aPC}\ p\ *\ \texttt{aR 13w}\ r13\ *\ \texttt{aMEMORY}\ df\ f) \\
& \{(p, \texttt{0xE8BD8000w}\ \text{“}\texttt{pop \{pc\}}\text{”})\} \\
& (\texttt{aPC}\ (f\ r13)\ *\ \texttt{aR 13w}\ (r13 + \texttt{4w}) * \texttt{aMEMORY}\ df\ f)
\end{array}
$$

Figure 5.3. Hoare triple theorem for instruction that restores saved link register value to the program counter.

that enables us to determine the value of $f\ r13$. The return address for a (non-leaf) function is saved to the stack in the function prologue before any instructions in the function. An example of such an instruction is "`push {lr}`", with the following Hoare triple in Figure 5.4.

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 14w}\ r14\ *\ \texttt{aR 13w}\ r13\ *\ \texttt{aPC}\ p\ *\ \texttt{aMEMORY}\ df\ f) \\
& \{(p, \texttt{0xE92D4000w}\ \text{“}\texttt{push \{lr\}}\text{”})\} \\
& (\texttt{aR 14w}\ r14\ *\ \texttt{aR 13w}\ (r13 - \texttt{4w}) * \texttt{aPC}\ (p + \texttt{4w})\ *\ \texttt{aMEMORY}\ df\ ((r13 - \texttt{4w} \mapsto r14)\ f))
\end{array}
$$

Figure 5.4. Hoare triple theorem for instruction in function prologue.

The memory in the post-state of the function prologue is "`((r13 - 4w ↦ r14) f)`", which contains the value of the link register, `r14`, at the top of the stack, at the address `r13 - 4`. Hence, the information we need to discharge the safety assertion at the function exit is the memory expression at the post-state of the function prologue, and the new value of register `r13`. After substituting the post-state memory and register `r13` values of the function prologue into the return instruction, the program counter in the return instruction post-state will contain `((r13 - 4w ↦ r14) f) (r13 - 4w)` which simplifies to `r14`, and the safety assertion simplifies to $\texttt{r14} \in \{\texttt{addr} \mid \texttt{0x85F8} \leq \texttt{addr} \wedge \texttt{addr} \leq \texttt{0x85F4}\}$, which can be discharged by any caller of the function that supplies a concrete value of `r14`. As long as the prologue precedes every instruction in the function, and the function does not alter the callee-saved registers until its epilogue, this substitution is valid. Again, we can use the pre-composition tactic to substitute the value of the memory (and registers) at the post-state of the function prologue into every subsequent basic block in the function. Unlike the forward-propagation of branch-conditions, a fixed-point analysis is not required, and we directly substitute the information from the function prologue into every subsequent basic block in the function, before the safety proof obligations are automatically discharged in AUSPICE's abstract interpretation.

5.4 Automatic Discharge of Proof Obligations

There are two ways to discharge the safety assertions of a theorem. First, for a given safety theorem, the pure boolean conditions of the pre-state of the theorem preceding it may imply that the safety assertion holds for the current theorem. Second, if the former does not hold, then the safety assertion is added to the hypotheses of the preceding instruction, and the Frame rule in Hoare logic is used to add the undischarged assertion to the theorems of the preceding instructions. We use abstract interpretation [134] to identify safety assertions that cannot be discharged. At each instruction, our analysis records the safety assertions that need to be framed to the safe instruction theorem for that instruction.

Algorithm 5.3 Safety Assertion Analysis

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

We use a flow-sensitive backwards fixed-point analysis. Our analysis proceeds across all nodes in the CFG of a function in reverse topological order in each iteration. Each CFG node is a basic block in the function, and each node is associated with a safe basic block theorem (§4.2.2). At each node, the analysis checks that for each predecessor node, the instruction theorem for that node has pure boolean conditions that can discharge the safety assertions at the current node's theorem.

For safety assertions that the predecessor node's theorem cannot discharge, our analysis adds the assertion to the predecessor node's theorem, propagating the assertion backwards up the CFG. Our analysis is also inter-procedural, and context-sensitive. Each function is summarized at its call-site by a `FUN_SAFE` theorem for that particular call-site.

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

Algorithm 5.3 describes our static-analysis algorithm. `GetThmPreconds` (Line 6) and `GetThmAsserts` (Line 7) return the pure boolean conditions in the pre-state and the safety assertion at a node's theorem respectively. `PROVE` tries to discharge the given safety assertion, *assert*, using the given conditions *pred_preconds* from the predecessor theorem, and returns true if it can discharge the safety assertion, and false otherwise (Line 9). If the safety assertion cannot be discharged, it is added to the analysis information for the node's predecessor node (Line 10), so that it will be framed to the predecessor node's theorem after the analysis. The analysis information also records the path along which each assertion is propagated in *info.path* (Line 12). Then, the analysis checks if there is a cycle along the propagation path of the assertion (Line 13) in the function `AbortIfAssertPathIsCycle`, and terminates the AUSPICE proof process if a cycle is found. This is because if a cycle is found along which the pre-composition tactic causes the safety assertion term to change with each iteration, the analysis is likely to not terminate as it will keep adding new safety assertion terms to the analysis information on each successive iteration of the analysis.

5.5 Other Challenges in Proof Automation

5.5.1 Support for `Switch` Statements

Next, we describe how AUSPICE generates safety proofs for machine-code programs containing indirect jump tables compiled from `switch` statements in C. Additional steps are required in Line 13 in Algorithm 5.1 to prepare machine-code compiled from `switch` statements for safety proof generation.

Proving safety properties for machine-code compiled from C `switch` statements requires special handling in AUSPICE. This is because C `switch` statements are compiled to indirect jump tables in machine-code by a typical compiler. As a result, the value of the program counter after executing the jump table is a symbolic value that is computed from the input value supplied to the `switch` statement. However, recall that our definition of CFI safety (§3.1.2) requires the value of the program counter to always be within the range of addresses that program instructions are loaded to. Hence, to discharge the control-flow safety conjunct that asserts the safe address range for the program counter value (§5.2), the program counter value needs to be concrete.

Typical Structure for a Jump Table Compiled from a `Switch` Statement

```
int curr_char;
switch(c) {
case 0: curr_char = '0'; break;
case 2: curr_char = '2'; break;
case 4: curr_char = '4'; break;
case 5: curr_char = '5'; break;
case 6: curr_char = '6'; break;
case 7: curr_char = '7'; break;
case 8: curr_char = '8'; break;
case 9: curr_char = '9'; break;
}
```

Figure 5.5. Example of a C `switch` statement.

```
808c: e3530009  cmp r3, #9
8090: 979ff103  ldrls pc, [pc, r3, lsl #2]
8094: ea000021  b 8120 <main+0xac>
8098: 000080c0
809c: 00008120
...
80bc: 00008114
...
8120: e55b3005  ldrb r3, [fp, #-5]
```

Figure 5.6. Example compiled ARM machine-code for a `switch` table.

Typically, a `switch` statement with a small, finite number of possible switch target values is compiled to a jump table with four components:

1. A comparison instruction (`cmp` on ARM), that checks if the input value to the `switch` statement falls within the values for the jump table,
2. A jump table compute instruction (typically a conditional load `ldrls` which loads the jump table entry into the program counter), which performs the jump target computation,
3. An exit branch instruction if none of the `switch` cases are to be executed, and
4. The jump table itself, containing target addresses as constants within the program text.

Figure 5.5 shows an example of a C `switch` statement, and Figure 5.6 shows the ARM machine-code compiled from the C program. Then, the instruction at `0x808C` is the initial comparison instruction, the instruction at `0x8090` performs the jump table computation, the instruction at `0x8094`

jumps out of the jump table for non-matching inputs, and the instruction words at the addresses from 0x8098 to 0x80BC are jump table entries that contain the program counter values for the valid input values stored in register r3.

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aPC 32912w} * \texttt{aR 3w } r3 * \texttt{aS1 psrC } psrc * \texttt{aS1 psrZ } psrz * \texttt{precond}(\neg psrc \vee psrz) * \\
& \texttt{cond}((pc\_rel \;\&\&\; \texttt{3w} = \texttt{0w}) \wedge (r3 \ll 2 \neq \texttt{0xFFFFFFF8w}) \wedge (\texttt{32920w} + r3 \ll 2 \;\&\&\; \texttt{3w} = \texttt{0w}))) \\
& \{(\texttt{32912w}, \texttt{0x979ff103w}\text{ ``}\texttt{ldrls pc, [pc, r3, lsl \#2]}\text{''}); (\texttt{32920w} + r3 \ll 2, pc\_rel)\} \\
& (\texttt{aPC } pc\_rel * \texttt{aR 3w } r3 * \texttt{aS1 psrC } psrc * \texttt{aS1 psrZ } psrz)
\end{array}
$$

Figure 5.7. Hoare triple theorem for jump table target computation instruction "ldrls pc, [pc, r3, lsl #2]".

Challenge with Reasoning about a Switch Jump Table

The main challenge with reasoning about the compiled machine-code for the switch statement lies with the instruction that dynamically computes the jump target for the switch, ldrls. The ldrls instruction, as decompiled by the Cambridge ARM model, has a Hoare triple theorem whose post-state program counter has a symbolic value. Figure 5.7 shows the Hoare triple for the ldrls instruction at 0x8090 in Figure 5.6 for the condition where the condition flags have the value $(\neg psrc \vee psrz) = 1$ (as evaluated by the "cmp r3, #9" comparison instruction).

From the Hoare triple theorem in Figure 5.7, the program counter value after the instruction executes is *pc_rel*, which is a symbolic variable. This program counter value is the jump table entry as stored at the address given by the expression "$\texttt{32920w} + r3 \ll 2$", which is the jump table address computation performed by the ldrls instruction. Hence, the value of the program counter after the execution of the ldrls instruction depends on the value of register r3, which cannot be statically determined at compile-time in order to compute a concrete value for the program counter.

Concretization of Switch Jump Instruction

To enable reasoning about the ldrls jump table computation instruction, we need to concretize the value of the program counter in the post-state of the theorem for the instruction. This effectively treats each possible switch branch as a separate goto. There are three observations about a machine-code jump table compiled from a C switch statement:

1. There is a finite (and typically small) number of jump table entries.

2. The (register) value used in the jump table comparison (`r3` in our example in Figure 5.6) is typically zero-based. Even for `switch` statements in C whose possible case values may include non-zero values, the compiler (`gcc` in our observations) rebases the value of the register to begin from 0.

3. The jump table is indexed by a contiguous range of integers from 0 to the operand in the comparison instruction at the start of the compiled machine-code for the jump table.

From these observations, we can compute the possible index values into the jump table based on the comparison value in the comparison instruction at the start of the compiled code for the `switch` statement. Thus, we can compute a concretized version of the Hoare triple theorem in Figure 5.7 by constructing an N Hoare triple theorems, one for each possible value of the jump register (`r3` for our example), and concretely computing the jump table entry, and extracting the jump table entry to obtain a concrete program counter value, for each triple.

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aPC 32912w} * \texttt{aR 3w } r3 * \texttt{aS1 psrC } psrc * \texttt{aS1 psrZ } psrz * \texttt{precond}(\neg psrc \vee psrz) * \\
& \texttt{cond}((\texttt{32960w \&\& 3w} = \texttt{0w}) \wedge (\texttt{0w} \neq \texttt{0xFFFFFFF8w}) \wedge (\texttt{32920w \&\& 3w} = \texttt{0w}))) \\
& \{(\texttt{32912w}, \texttt{0x979ff103w } \text{“}\texttt{ldrls pc, [pc, r3, lsl \#2]}\text{”}); (\texttt{32920w}, \texttt{32960w})\} \\
& (\texttt{aPC 32960w} * \texttt{aR 3w 0w} * \texttt{aS1 psrC } psrc * \texttt{aS1 psrZ } psrz)
\end{array}
$$

Figure 5.8. One of the 11 concretized Hoare triple theorems for jump table target computation instruction “`ldrls pc, [pc, r3, lsl #2]`”. Note that the instruction word “`32960w`” is not actually an instruction, but contains an address for the jump table for the `switch` statement.

To concretize the jump instruction for a `switch` table, we take the following steps:

1. Identify jump table computation instruction for a jump table, note its address p.

2. Identify constant operand *cmp_opd*, and register operand *jmp_reg* in comparison instruction at address $p-4$.

3. Duplicate theorem for jump table computation instruction N times for $N = cmp\_opd + 1$.

4. For each theorem, substitute variable *jmp_reg* for each of values $0, \ldots, (N-1)$.

5. Look up jump table entries (i.e., constants in program text) at addresses $p+8, p+12, \ldots, p+((N+2)*4)$, ($p$ holds the jump table computation instruction, and $p+4$ holds the exit/fall-through instruction) and substitute them in the appropriate duplicate theorem from step 3.

This concretization process takes place as a wrapper around the `CambridgeARM_GetInstrModel` function (Line 13 in Algorithm 5.1). Figure 5.8 shows one of the 11 concretized Hoare triple theorems for the `switch` example that is obtained after we applied the above steps to concretize the theorem for the `switch` example presented. For each instruction, if the instruction is a jump table computation (i.e., `ldrls`), we carry out the above concretization process on the Hoare triple theorem returned from the Cambridge ARM model before proceeding to return the (resulting concretized) theorem to AUSPICE. Thus, the end result of our concretization process is an N-way branching instruction at the jump table instruction `ldrls`, where each branch has a concrete program counter value. This in turn enables AUSPICE to proceed with its automated safety property proof process, as symbolic program counter values have been concretized, enabling safety assertions about the program counter value to be automatically discharged.

5.5.2 Memory Reconciliation in `FSI_Rule`

Next, we describe the process of "memory reconciliation" in the implementation of `FSI_Rule` in AUSPICE, which enables safety proof generation to proceed for safety theorems that Selective Composition (§5.3.2) has been applied to.

In the Safe Function (`FUN_SAFE`) rule in our $\mathscr{L}_{LR}$ program logic (§4.2.3), the clauses specifying the requirements for safety to hold at the intra-procedural and inter-procedural levels both require the post-state of (the rearranged Hoare triple or `FUN_SAFE` theorems of) predecessor nodes to be equal to the pre-state of successor nodes at each CFG edge. This is in keeping with the spirit of the Compose rule in typical Hoare logics [132, 31] when reasoning about successively executed code at the same time.

One of the goals of the Selective Composition proof tactic in AUSPICE's proof automation is to make global information of a function, in the form of saved caller register values on the stack, available for local usage (§5.3.2) at particular CFG nodes in a function being analyzed. In particular, this involves making a substitution for the memory expression at every CFG node that comes after the prologue of the function in the CFG.

This use of the Selective Composition proof tactic creates a challenge for ensuring that the post-state of predecessor CFG nodes is equal to the pre-state of successor CFG nodes, as the pre- and post-state contain assertions about memory (using the "`aMEMORY`" relation). This challenge arises in the implementation of the `FSI_rule` (Line 8 in Algorithm 5.1).

Effects of Selective Composition on Memory Assertions on `FSI_rule`

To begin, we illustrate the effects of the Selective Composition proof tactic when used to make global information (i.e., register values saved in the function prologue) available for local reasoning at individual CFG nodes in a function (§5.3.2). These effects present challenges to our proof automation, when we wish to make global information (in the form of memory updates of the function's prologue) available at local nodes (i.e., all other basic blocks in the function other than its prologue basic block), that we now illustrate.

Consider a function prologue, $c_{prologue}$, whose behavior is described by the (simplified) Hoare triple in Figure 5.9. In this function, the prologue saves the value of the link register `r14` to its own stack (at address "$r13 - 4\mathtt{w}$") before any code in the function executes. Then, to make this saved value of register `r14` available at every CFG node in the function (following the function prologue in the CFG), the substitution $[(r13 - 4\mathtt{w} \mapsto r14)\, f/f]$ is applied to the Hoare triple theorem describing every CFG node in the function before the safety assertion analysis is carried out (Line 6 in Algorithm 5.1).

$$
\begin{array}{ll}
\vdash & \mathtt{SPEC\ ARM\_MODEL} \\
& (\mathtt{aR\ 13w}\ r13\ *\ \mathtt{aR\ 14w}\ r14\ *\ \mathtt{aMEMORY}\ df\ f)\ c_{prologue} \\
& (\mathtt{aR\ 13w}\ r13\ *\ \mathtt{aR\ 14w}\ r14\ *\ \mathtt{aMEMORY}\ df\ (r13 - 4\mathtt{w} \mapsto r14)\, f)
\end{array}
$$

Figure 5.9. Hoare triple for function prologue. f represents the contents of memory just before this block of instructions $c_{prologue}$ executes.

Then, suppose there are two adjacent basic blocks, with code c_1 and c_2, in this function, with the (simplified) Hoare triple theorems shown in Figure 5.10. Note that the first two steps of the Pre-Composition proof tactic (§4.1.3), also known as the "shift" operation [31], have already been carried out, and the resources asserted in c_1's theorem but not in c_2's theorem have already been added to c_2's theorem using the Frame rule in Hoare logic, and vice versa. Thus, for the pair of theorems in Figure 5.10, completing the Pre-Composition proof tactic prior to applying the Hoare Logic Compose rule is straightforward: we only need to apply the substitution $[(r13 - 8\mathtt{w} \mapsto r0)\, f/f]$ to the theorem for c_2.

On the other hand, after applying the Selective Composition proof tactic to this pair of theorems for code c_1, c_2, the two theorems will be as shown in Figure 5.11. If we proceed with the third step of the Pre-Composition proof tactic, we would apply the substitution $[(r13 - 8\mathtt{w} \mapsto r0)\ (r13 -$

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ f)\ c_1 \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ (r13 - \texttt{8w} \mapsto r0)\ f) \\
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ f)\ c_2 \\
& (\texttt{aR 0w 0w} * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ f)
\end{array}
$$

Figure 5.10. Simplified Hoare triple theorems for example code c_1 and c_2.

$\texttt{4w} \mapsto r14)\ f/f]$ to the theorem for c_2. This will result in the theorem in Figure 5.12, but we can see that the post-state of c_1's theorem in Figure 5.11 is no longer equal to the pre-state of c_2's theorem in Figure 5.12 due to the now mismatched memory expressions.

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ (r13 - \texttt{4w} \mapsto r14)\ f)\ c_1 \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ (r13 - \texttt{8w} \mapsto r0)\ (r13 - \texttt{4w} \mapsto r14)\ f) \\
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 0w } r0 * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ (r13 - \texttt{4w} \mapsto r14)\ f)\ c_2 \\
& (\texttt{aR 0w 0w} * \texttt{aR 13w } r13 * \texttt{aMEMORY } df\ (r13 - \texttt{4w} \mapsto r14)\ f)
\end{array}
$$

Figure 5.11. Simplified Hoare triple theorems for example code c_1 and c_2, after applying the Selective Composition proof tactic.

Hence, in functions whose CFG nodes' Hoare triple theorems have had the Selective Composition proof tactic applied to them, we need to perform an additional step that we call "memory reconciliation", so that the post-state of predecessor Hoare triples can be made equal to the pre-state of successor Hoare triples, before the `FSI_rule` can be successfully completed.

Memory Reconciliation

Memory Reconciliation is a step taken in the `FSI_rule` function, that proves a rewrite theorem to "reconcile" the memory expression in a predecessor Hoare triple's post-state with the memory expression in its successor Hoare triple's pre-state. Memory Reconciliation resolves the differing memory expressions, that we illustrated above, due to the use of the Selective Composition proof

$$
\begin{array}{ll}
\vdash & \texttt{SPEC ARM\_MODEL} \\
& (\texttt{aR 0w } r0 * \texttt{ aR 13w } r13 * \\
& \texttt{aMEMORY } df\ (r13 - \texttt{4w} \mapsto r14)\ (r13 - \texttt{8w} \mapsto r0)\ (r13 - \texttt{4w} \mapsto r14)\ f) \\
& c_2 \\
& (\texttt{aR 0w 0w} * \texttt{ aR 13w } r13 * \\
& \texttt{aMEMORY } df\ (r13 - \texttt{4w} \mapsto r14)\ (r13 - \texttt{8w} \mapsto r0)\ (r13 - \texttt{4w} \mapsto r14)\ f)
\end{array}
$$

Figure 5.12. Hoare triple theorem for c_2 after fully applying Pre-Composition proof tactic to it.

tactic to make global information (function prologue memory updates to store callee-saved registers to the stack) available at the CFG nodes in a function's body.

Essentially, Memory Reconciliation proves that the memory expression in the post-state of a predecessor theorem is equal to the memory expression in the pre-state of its successor theorem. Thus, Memory Reconciliation proves theorems of the form (where $\{freevars(a), freevars(c)\} \cap \{freevars(b), freevars(d)\} = \emptyset$, and $freevars(\cdot)$ indicates the free variables in each expression, which we have found to be true in practice in the machine-code programs that we analyzed):

$$\vdash (a \mapsto b)(c \mapsto d)(a \mapsto b)f = (c \mapsto d)(a \mapsto b)f$$

These theorems can be used to simplify the successor Hoare triple in a pair of theorems. In our example, for the Hoare triple in Figure 5.12, we wish to generate the theorem:

$$\vdash (r13 - \texttt{4w} \mapsto r14)\ (r13 - \texttt{8w} \mapsto r0)\ (r13 - \texttt{4w} \mapsto r14)\ f = (r13 - \texttt{8w} \mapsto r0)\ (r13 - \texttt{4w} \mapsto r14)\ f$$

We make use of the proof rules in `UpdateTheory` in the HOL4 library, as shown in Figure 5.13, to prove the above rewrite theorem. The `UPDATE_EQ` rule is an "overwrite" rule: for memory that is expressed as a list of memory updates, newer updates overwrite older updates. The `UPDATE_COMMUTES` rule specifies that for addresses that do not overlap, i.e., there is no aliasing of addresses, we can reorder the updates in an update expression. The mechanism we have adopted for our memory updates is part of the Cambridge model [31], although we can explore more efficient techniques for representing and capturing state updates to memory using techniques from the ODL extension of dynamic logic [135] by using "non-rigid" function symbols to represent the state being updated, i.e., main memory, in our case.

$$\frac{(a\mapsto c)((a\mapsto b)f)}{(a\mapsto c)f}\ \texttt{UPDATE\_EQ}$$

$$\frac{a \neq b \quad (a\mapsto c)((b\mapsto d)f)}{(b\mapsto d)(a\mapsto c)f}\ \texttt{UPDATE\_COMMUTES}$$

Figure 5.13. Proof Rules in Update Theory in HOL4.

The key observation of the differing memory expressions in the pre-state of a predecessor theorem and the post-state of its successor theorem, is that the memory updates in the function prologue are repeated in the memory expression of the successor theorem. Hence, for each repeated memory update in the memory expression of the successor theorem, we can make use of a combination of: (i) one or more applications of the `UPDATE_COMMUTES` rule to swap distinct and non-overlapping memory updates, and (ii) the `UPDATE_EQ` rule to eliminate the repeated memory update.

Thus, Memory Reconciliation proceeds as follows, for each pair of Hoare triple theorems from adjacent nodes in the CFG of a function:

1. Extract memory expression asserted by the "`aMEMORY`" relation from the post-state of predecessor Hoare triple and pre-state of successor Hoare triple: proceed if the two memory expressions differ, otherwise no Memory Reconciliation is needed.

2. Next, Memory Reconciliation is carried out on the memory expression with repeated updates to the same address (this is typically the memory expression in the pre-state of the successor's Hoare triple).

3. For each outermost memory update "$(a\mapsto b)$", check if the update "$(a\mapsto b)$" is repeated later in the same memory expression. If so, apply the `UPDATE_COMMUTES` rule to swap the outermost memory update with the next memory update.

4. For each repeated memory update, recursively swap the outermost memory update with the next memory update (by proving that the two memory updates are to addresses that are different) until the repeated memory update is adjacent to its repeated update (i.e., we obtain the expression "$(a\mapsto b)((a\mapsto b)f)$"). Then, apply the `UPDATE_EQ` rule to eliminate the repeated memory update.

Thus, a series of rewrite theorems is obtained from the above steps (which could be simplified using non-rigid functions as in the ODL extension of dynamic logic for verifying object-oriented

programs [135]). We use HOL4's built-in `METIS` automatic prover, supplying the rewrite theorems to the prover, to prove that the memory expressions asserted in the post-state of the predecessor theorem and the pre-state of the successor theorem are equal, thus obtaining our desired rewrite theorem.

We can then simplify the Hoare triples with the obtained rewrite theorem, which completes the process of ensuring that the post-state of the predecessor Hoare triple is equal to the pre-state of the successor Hoare triple, thus enabling us to reason about combined behavior of the two adjacent CFG nodes, when the Selective Composition proof tactic (§5.3.2) is used.

Feasibility of Memory Reconciliation

Note that there are two classes of memory-write addresses that occur: (i) writes to local variables of the current function, whose addresses are constant offsets from the stack pointer (i.e., $r13 - N$ for some constant N), and (ii) possibly "unsafe" writes to memory to computed addresses that are stored in an intermediate register, e.g., *r3*.

Writes to local variables can always be automatically reconciled in this above process, as their stack addresses will not overlap with the callee-saved register values on the stack, hence their addresses will not overlap with the address range where callee-saved registers are saved. As a result, we can always automatically prove that the written addresses are not equal to the repeated addresses where callee-saved register values are saved.

On the other hand, writes to "unsafe" locations will have had additional source-code safety-checks prescribed by PCFIRE-C (§3.2.1). The third and fourth clauses in the prescribed safety-checks will provide the logic pre-conditions that are sufficient to enable us to prove that the unsafe addresses do not overlap with the addresses where callee-saved register values are located on the stack in memory. This then allows the `UPDATE_COMMUTES` rule to be applied, and the Memory Reconciliation process to be carried out successfully.

5.6 Summary

In this chapter, we described the proof automation algorithms in AUSPICE [128], which realize the logic approach in this dissertation. We described the overall automatic proof generation algorithm of AUSPICE, and we described details of the components of the proof automation algorithm. The overall goals of AUSPICE's proof automation algorithm are to: (i) specify safety assertions that

need to be discharged, (ii) find logic pre-conditions that can discharge each safety assertion, and (iii) generate proofs that safety assertions are discharged, and construct Safe Function theorems using the `FUN_SAFE` rule.

The proof automation of AUSPICE focused on rearranging Hoare triple theorems from the Cambridge model [31], which the $\mathscr{L}_{LR}$ logic that we developed extends, to enable reasoning about dynamic program behavior in the presence of our automatically specified safety assertions. The bulk of the automation work focused on obtaining Hoare triple theorems of the appropriate shape for use in our $\mathscr{L}_{LR}$ proof rules (§4). This proof automation work could be simplified in dynamic logic, which enables specifying both the program as well as properties in the same logic [108], as we would not need to concern ourselves with obtaining appropriately-shaped theorems, as the semantics of the program are captured in the semantics of the logic rather than as the pre-state and post-state assertions that make up Hoare triples.

First, our safety properties for ensuring CFI are automatically specified as safety assertions at the Hoare triple theorem for every instruction, and the `MEMCFISAFE` rule ensures every Hoare triple theorem contains its safety assertions. Second, the Selective Composition proof tactic in AUSPICE prepares safe basic block theorems for the automatic discharge of safety proof obligations by ensuring that: (i) logic pre-conditions needed to discharge safety assertions, that are located in multiple basic blocks preceding the basic block whose safety assertion needs to be discharged, are propagated forward to the basic block where they are needed, and (ii) global information in the form of callee-saved register values from the function prologue are propagated forward to enable proving that function returns are safe. Third, AUSPICE uses abstract interpretation to search for pre-conditions to discharge safety assertions automatically.

Finally, we described some of the challenges faced by AUSPICE in its proof automation, and how we addressed them. We described how proofs can be automated for machine-code jump tables compiled from switch statements, whose Hoare triples have program counter values that are symbolically computed, preventing automatic proof discharge. We described our algorithm for concretizing the program counter value for such machine-code jump tables to enable automated proofs. Second, we described the Memory Reconciliation algorithm in the `FSI_rule` construction of Safe Function theorems, which enables `FUN_SAFE` theorems to be proved when the Selective Composition proof tactic is used for propagating global information in a function to all CFG nodes in the function.

Chapter 6

Proofs for Realistic Embedded Programs

This chapter describes the extensions [136] to the AUSPICE automated safety property generation framework in the logic approach of this dissertation, in order to support: (i) safety proofs for user-mode machine-code programs that contain system calls, and (ii) larger-sized programs, with safety proofs generated in less time, through optimizations to the proof automation in AUSPICE. In §6.1, we describe our axiomatic approach to support reasoning about the user-mode-visible effects of system call invocations serviced by the underlying operating system (OS) kernel, and in §6.2 we describe our optimizations to the proof automation algorithm in AUSPICE.

6.1 Safety Proofs for Machine-code with System Calls

6.1.1 Approach and Design

There are two main steps to support safety proofs for ARM user-mode machine-code programs containing system calls (syscalls).

First, we need to model the ARM supervisor call instruction (`svc`), whose effects occur in both user-mode, and in supervisor-mode where the OS services the syscall. As our focus is 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-mode to supervisor-mode, and that the OS correctly services the syscall (§1.3). 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 then introduce as hypotheses in our model. Note that for a program containing syscalls, our generated proofs of CFI safety properties for these programs will retain these unproved

Hoare triples for each syscall in its antecedents/hypotheses. Thus, the generated proofs of CFI safety properties for programs with syscalls is not unconditional, and is dependent on our assumptions of the behavior of the syscalls present in the particular program. Our current goal is to make our assumptions about syscall behavior explicitly, and these assumed triples can currently be examined manually. In future, we envision that these assumptions can be analyzed mechanically.

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 (e.g., for data processing and branching) in user-mode programs, the proven Hoare triples for each instruction contain enough information for computing concrete safety assertions for each instruction (Line 14 in Algorithm 5.1). 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 up to the `svc` instruction, and in the callers of the syscall. Note that in our approach, for syscalls that write to the memory of a user process, we support reasoning about safety properties for CFI only when the syscalls are invoked with a concrete buffer address and a concrete number of bytes, as our approach needs to identify the exact memory locations written to, in order to determine that our CFI safety properties are not violated.

In AUSPICE, 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.

6.1.2 Modeling of System Calls in User-mode Programs

Rationale Behind Model

Our model of syscall behavior must capture the user-mode-visible effects of the invocation of the syscall, as these may affect the safety properties proved in our logic approach.

First, we consider 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 [137]), 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 [137]).

Second, we need to know what the user-mode-visible effects of each syscall in user-mode are. We need to (i) retrieve the number of the syscall being invoked (as passed in register `r7`, based on the Linux Application Binary Interface (ABI) for ARM [138]), (ii) identify the syscall being invoked (e.g., from the Linux kernel's documentation and/or source-code), and (iii) retrieve the arguments passed to the syscall (either via user-mode registers, or on the stack of the user code). Then, we can identify the behavior of each syscall invoked based on its specification. We can then instantiate our safety-assertions based on the user-mode-observed effects of each syscall invocation.

Axiomatization of System Call Effects

Next, we "axiomatize" the Hoare triples for syscalls. We "axiomatize" the Hoare triples for syscalls by constructing an unproven Hoare triple for each syscall, which we then introduce as an assumption. Note that we do not introduce our constructed, and unjustified Hoare triples, as axioms into our proofs, and we merely introduce them as assumptions. These unproven Hoare triples are then collected as hypotheses of the final safety proof, and they formalize our assumptions of each syscall's effects on user-mode state, based on the syscall's specification.

Figure 6.1 shows an example of an axiom for the `write` syscall. `aR` asserts the value of the specified register, `aPC` asserts the value of the program counter, and `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 values of the other registers are asserted to be symbolic variables (*r0*, *r1*, *r2*, *r14*), as they are unknown when we analyze the `svc` instruction on its own. We will instantiate these symbolic variables with concrete values later when analyzing the instructions leading up to the syscall invocation (details in §6.1.3).

Note that the Hoare triple is repeated on the left-hand-side of the turnstile "⊢", indicating that the Hoare triple is a hypothesis. Note also that these unproved Hoare triples will remain in the hypotheses/antecedents of the safety theorem of CFI safety properties that our logic approach generates. 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 supports for which there are no effects that are directly visible in user-mode: `open`, `close`, `mmap`, `munmap`, `nanosleep`.

$$
\begin{array}{rcl}
\texttt{SPEC ARM\_MODEL} & \vdash & \texttt{SPEC ARM\_MODEL} \\
(\texttt{aR 0w } r0 \ast \texttt{ aR 1w } r1 \ast \texttt{aR 2w } r2 \ast \texttt{aR 7w 4w} \ast & & (\texttt{aR 0w } r0 \ast \texttt{ aR 1w } r1 \ast \texttt{aR 2w } r2 \ast \texttt{aR 7w 4w} \ast \\
\texttt{aPC } p \ast \texttt{aR 14w } r14 \ast \texttt{aMEMORY } df\ f) & & \texttt{aPC } p \ast \texttt{aR 14w } r14 \ast \texttt{aMEMORY } df\ f) \\
\{(p, \texttt{0xEF000000w ``svc \#00000000''})\} & & \{(p, \texttt{0xEF000000w ``svc \#00000000''})\} \\
(\texttt{aR 0w } rv \ast \texttt{aR 1w } r1 \ast \texttt{aR 2w } r2 \ast \texttt{aR 7w 4w} \ast & & (\texttt{aR 0w } rv \ast \texttt{aR 1w } r1 \ast \texttt{aR 2w } r2 \ast \texttt{aR 7w 4w} \ast \\
\texttt{aR 14w } r14 \ast \texttt{ aPC } (p+4\texttt{w}) \ast \texttt{aMEMORY } df\ f) & & \texttt{aR 14w } r14 \ast \texttt{ aPC } (p+4\texttt{w}) \ast \texttt{aMEMORY } df\ f)
\end{array}
$$

Figure 6.1. 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.

In contrast, consider our constructed axiom for the `read` syscall in Figure 6.2. `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 "$\texttt{cond}(addrs \subseteq df)$" 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`.

Quantifiers on Opaque System Call Effects

In our constructed assumed Hoare triples for syscalls, opaque syscall effects, specifically return values from syscalls and memory effects, are universally quantified. Thus, for example, our constructed Hoare triples describe syscalls that can simultaneously return 0 and 1, however, such a syscall cannot exist, as each syscall returns only one specific value. As a result, our assumed syscall triples cannot be justified, and hence are false. Thus, for programs containing syscalls, the antecedents/hypotheses of the proofs generated by our logic approach contain Hoare triples (that we construct and leave unjustified) that cannot be justified, and hence are false. However, our assumed syscall triples do not affect the validity of our generated proofs of CFI safety properties, as we explain below.

$$
\begin{array}{rcl}
\texttt{SPEC ARM\_MODEL} & \vdash & \texttt{SPEC ARM\_MODEL} \\
(\texttt{aR 0w } r0 * \texttt{aR 1w } r1 & & (\texttt{aR 0w } r0 * \texttt{aR 1w } r1 \\
* \texttt{aR 2w } r2 * \texttt{aR 7w 3w} * \texttt{aR 14w } r14 & & * \texttt{aR 2w } r2 * \texttt{aR 7w 3w} * \texttt{aR 14w } r14 \\
* \texttt{aPC } p * \texttt{cond}(addrs \subseteq df) * \texttt{aMEMORY } df\ f) & & * \texttt{aPC } p * \texttt{cond}(addrs \subseteq df) * \texttt{aMEMORY } df\ f) \\
\{(p, \texttt{0xEF000000w } \text{“}\texttt{svc \#00000000}\text{”})\} & & \{(p, \texttt{0xEF000000w } \text{“}\texttt{svc \#00000000}\text{”})\} \\
(\texttt{aR 0w } rv * \texttt{aR 1w } r1 * \texttt{aR 2w } r2 * \texttt{aR 7w 3w} * & & (\texttt{aR 0w } rv * \texttt{aR 1w } r1 * \texttt{aR 2w } r2 * \texttt{aR 7w 3w} * \\
\texttt{aR 14w } r14 * \texttt{aPC } (p+4\texttt{w}) * \texttt{aMEMORY } df\ (g\ f)) & & \texttt{aR 14w } r14 * \texttt{aPC } (p+4\texttt{w}) * \texttt{aMEMORY } df\ (g\ f))
\end{array}
$$

Figure 6.2. Constructed Hoare triple axiom for the `read` syscall.

In our proof automation, the proof tactics and proof steps that we have developed do not make use of the fact that our assumed syscall triples are unjustifiable (and hence are false). Our proof tactics and proof steps, as described in §4.2, retain any antecedents/hypotheses in each of the theorems being manipulated. Thus, by inspection of our proof tactics and proof steps in §4.2, our generated proofs of CFI safety properties remain valid, although our generated proofs are based on unjustifiable antecedents/hypotheses.

Moving forward, this issue can be addressed in two ways. First, we can existentially quantify syscall effects instead of universally quantifying them. However, introducing an existential quantifier changes the shape of the Hoare triples, which would require significant re-engineering of our proof automation algorithms. This is a limitation of using Hoare logic, in which proof steps are dependent on the shapes of theorems, and this issue could be alleviated if we had reasoned about the program in Dynamic Logic instead [106]. Second, we can also bound the possible opaque and universally quantified effects by specifying an oracle in the pre-state of each assumed Hoare triple that supplies the return values and memory transformations for a syscall given a number of transitions undertaken on the underlying system state.

We relegate these explorations to future work, and we intend to explore a more robust construction of our assumed Hoare triples for syscalls, by both existentially quantifying syscall effects, and using oracles on underlying system state to avoid relying on false triples as justifications for our CFI safety proofs.

Algorithm 6.1 Hoare triple extraction for individual ARM instructions, with support for unproven triple construction for syscalls.

```
1: function GETINSTRUCTIONMODEL_WITHSYSCALLS(addr, addr_to_func map, instr)
2:     if !(instr = 0xEF000000) then
3:         return CAMBRIDGEARM_GETINSTRUCTIONMODEL(instr)
4:     else
5:         func_containing_instr ← addr_to_func[addr]
6:         return CONSTRUCTSYSCALLTRIPLE(instr,addr,func_containing_instr)
7: end function
```

Algorithm 6.2 Algorithm for unproven Hoare triple construction for syscalls.

```
1: function CONSTRUCTSYSCALLTRIPLE(instr, addr, func_containing_instr)
2:     syscall_num ← GETSYSCALLNUMFORSYSCALL(func_containing_instr)
3:     num_syscall_args ← GETSYSCALLNUMARGS(syscall_num)
4:     register_asserts ← ∅
5:     for i ∈ {0,...,num_syscall_args} do
6:         curr_reg_assert ← MAKEDEFAULTARGASSERT(i)
7:         register_asserts ← register_asserts ∪ curr_reg_assert
8:     end for
9:     register_asserts ← register_asserts ∪ MAKEARGASSERT(7, syscall_num)
10:    mem_prestate ← "f"
11:    if DOESSYSCALLMODIFYMEMORY(syscall_num) then
12:        mem_poststate ← CONSTRUCTOPAQUEMODIFIEDMEMEXPR(syscall_num, addr)
13:    else
14:        mem_poststate ← "f"
15:    end if
16:    (pre_state_pc, post_state_pc) ← (MAKEPCASSERT(addr),MAKEPCASSERT(addr + 4))
17:    unproven_syscall_triple ← LIST_MK_COMB("SPEC ARM_MODEL", [register_asserts ∪
       mem_prestate ∪ pre_state_pc, "0xEF000000", register_asserts ∪ mem_poststate ∪
       post_state_pc])
18:    syscall_triple_assumption ← ASSUME(unproven_syscall_triple)
19:    return syscall_triple_assumption
20: end function
```

Implementation

The construction of unproven Hoare triples for each syscall (Algorithm 6.1) is implemented as a wrapper around the model construction for individual instructions in the Cambridge ARM model, and replaces Line 13 in Algorithm 5.1. When a `svc` instruction (`0xEF000000`) is detected, AUSPICE constructs an unproven Hoare triple based on the name of the function that the instruction is in. `ConstructSyscallTriple` (Algorithm 6.2) implements the unproven Hoare triple construction process described above. `MakeDefaultArgAssert` constructs a register assertion that asserts that the register with the given register number contains its default symbolic value; `MakeArgAssert` constructs a register assertion that asserts that the given register number contains the given value; `ConstructOpaqueModifiedMemExpr` constructs a symbolic expression for memory updates by the syscall for syscalls that modify user-mode memory; and `MakePCAssert` constructs an assertion of the value of the program counter. `List_Mk_Comb` and `Assume` are functions provided by the meta-logic in HOL4. We initially support modeling the following syscalls for simple I/O operations: `read`, `write`, `open`, `close`, `mmap`, `munmap`, `nanosleep`.

6.1.3 Supporting Safety Proof Automation for System Calls

Next, to support automated safety proofs in AUSPICE 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` (Algorithm 5.3) reasons about memory addresses individually. To concretize the symbolic effects of a syscall's unproven triple, AUSPICE examines the arguments the syscall is invoked with when running `SafeFunctionAnalysis` (Algorithm 5.1) on the caller of the syscall. We first illustrate how the arguments to system calls are interpreted, using the `read` syscall. Then we discuss how the symbolic effects are concretized, before we describe how these are implemented in AUSPICE's analysis.

System Call Arguments

The Linux Programmer's Manual [29] 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. 6.3 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. 6.4 shows the C prototype of the assembly-code wrapper.

```
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
...   ...
```

Figure 6.3. Example ARM machine code invoking the `c_read` wrapper to the `read` syscall.

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

Figure 6.4. Prototype of C function wrapper to `read` syscall.

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 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 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.

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 enumerates the byte-addresses written to by the syscall.

For the example in Fig. 6.3, 3 bytes are written to at the address `0x10250`. Hence, we substitute *addrs* with $\{$`0x10250w`; `0x10251w`; `0x10252w`$\}$, and the update function g with the expression shown in Figure 6.5, where `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.

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

Figure 6.5. Concretized memory-update expression for the `read` syscall in Figure 6.3.

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

Algorithm 6.3 describes the updated Safe Function analysis algorithm in AUSPICE, incorporating the unproven Hoare triple axiomatization (Line 15), and the concretization of symbolic effects (Line 5). In functions that call syscalls, `SafeFunctionAnalysisWithSyscalls` is first called on each syscall callee (Line 3). 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 function's basic blocks, *bb_safe* (Line 5). This concretization must take place before `SafetyAssertionAnalysis` (Line 8). 1300 lines of ML proof scripts were added to AUSPICE's code-base of 11.8 KLOC of ML to support proof automation for machine-code containing syscalls.

Algorithm 6.3 Updated Safe Function analysis in AUSPICE with support for safety proofs for machine code with syscalls. Added or changed steps are highlighted in blue.

1: **function** SAFEFUNCTIONANALYSISWITHSYSCALLS(*function_name*, *bb_safe_thms* `list`)
2: (*cfg*, *func*) ← ∀*i* ∈ *bb_safe_thms* · COMPUTECFGANDCALLEES(*i*) ▷ Compute Control-Flow Graph for *function_name*
3: *func_safe* ← ∀*callee* ∈ *cfg* · SAFEFUNCTIONANALYSISWITHSYSCALLS(*callee*, *bb_safe_thms*)
4: *syscall_callees* ← ∀*callee* ∈ *func* | IS_SYSCALL(*callee*)
5: *func_safe*′ ← ∀*c* ∈ *syscall_callees* · CONCRETIZEARGS(*func_safe*[*c*], *bb_safe*)
6: *bb_safe_thms* ← SC-FWDPROPAGATE-BRANCHCONDS(*bb_safe_thms*, *cfg*)
7: *bb_safe_thms* ← SC-FWDPROPAGATE-FUNCPROLOGUE(*bb_safe_thms*, *cfg*)
8: *assertion_info* ← SAFETYASSERTIONANALYSIS(*bb_safe_thms*, *func_safe*′, *cfg*)
9: *bb_safe_thms* ← AUGMENTTHEOREMS(*bb_safe_thms*, *func_safe*′, *assertion_info*)
10: *safety_theorem* ← FSI_RULE(*bb_safe_thms*, *func_safe*′, *cfg*)
11: **return** *safety_theorem*
12: **end function**
13: **function** AUSPICE((*addr*, *instr*) `list`) ▷ List of machine-code instructions
14: (*bb* `list`) ← COMPUTEBASICBLOCKS(*instr*) ▷ Compute basic blocks in program
15: (*bb_instr_thms* `list`) ← ∀*instr* · GETINSTRUCTIONMODEL_WITHSYSCALLS(*instr*) ▷ Obtain Hoare triple theorem for each *instr* in each *bb*, with support for syscalls
16: (*bb_safe_instr_thms* `list`) ← *map* (λ*x*.ADDSAFETYASSERTIONS(*x*)) *bb_instr_thms*
17: *bb_safe_thms* ← *map* (λ*x*.COMPOSESAFEINSTRS(*x*)) *bb_safe_instr_thms*
18: **return** SAFEFUNCTIONANALYSISWITHSYSCALLS(*main*,*bb_safe_thms*)
19: **end function**
20: AUSPICE(*program*)

6.2 Optimizing Safety Proof Automation

AUSPICE optimizes `SafeFunctionAnalysis` (Algorithm 5.1) and `SafetyAssertionAnalysis` (Algorithm 5.3), to improve the speed of its safety-proof generation, so that larger programs can be verified in less time. AUSPICE leverages (i) common patterns in `gcc`-generated machine code, to speed up `SafetyAssertionAnalysis`, and (ii) the behavior of safety-assertions for local-variable-writes in callee functions.

6.2.1 Common Compiler Conventions

AUSPICE's `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 is effectively an exercise in symbolic execution (i.e., we need to repeatedly perform variable substitutions to simulate the transformation of program state by the program, in the spirit of the Hoare logic Compose rule and the pre-composition tactic, as described in §4.1.3) along the propagation path.

We leverage two observations in (`gcc`) compiled code (or any compiler that generates such code). First, there are two classes of memory-writes: to local variables (to a constant offset from the frame pointer `r11`, or stack pointer `r13`), and to arbitrarily-computed addresses (typically stored in registers). Second, `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 in function bodies. To improve the speed of `SafetyAssertionAnalysis`, for writes to local variables, AUSPICE: (i) reduces the number of assertion terms analyzed, and (ii) skips the propagation-cycle check.

First, we represent the safety assertions for local-variable writes using range predicates, e.g., for a safety assertion "$\{r13 - \texttt{21w};\ r13 - \texttt{22w};\ r13 - \texttt{23w};\ r13 - \texttt{24w}\} \subseteq \{addr \mid addr < r11\}$", the addresses that are offset from `r13` are where the local variable is stored on the stack, and we use the range predicate "$\texttt{24w} \leq r13 < r11 + \texttt{24w}$". 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 (§3.1.2). 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.

Algorithm 6.4 Optimized analysis step for `SafetyAssertionAnalysis` in AUSPICE. 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.

```
1:  function SafetyAssertionAnalysis(bb_safe_thms map, cfg)
2:    info map ← ∅
3:    procedure AssertionAnalysisStep(info map, last_info map, cfg)
4:      for all node ∈ cfg do
5:        for all pred ∈ FindPreds(cfg, node) do
6:          pred_preconds ← GetThmPreconds(pred) ⋃ last_info[pred]
7:          node_asserts ← GetThmAsserts(node) ⋃ last_info[node]
8:          (range_pds, other_pds) ← partition is_range pred_preconds
9:          (localvar_asserts, other_asserts) ← partition is_localvar node_asserts
10:         for all assert ∈ localvar_asserts do
11:           curr_range_term ← narrow(compute_range_predicate(assert), range_pds)
12:           (prev_range_term, other_terms) ← partition is_range info.term[pred]
13:           info.term[pred] ← other_terms ⋃ narrow(curr_range_term, prev_range_term)
14:         end for
15:         for all assert ∈ node_asserts do
16:           if PROVE(pred_preconds, assert) == False then
17:             info.term[pred] ← info.term[pred] ⋃ assert
18:             a_path ← FindAssertPath(last_info.path[node], assert)
19:             info.path[pred] ← info.path[pred] ⋃ a_path
20:             AbortIfAssertPathIsCycle(a_path)
21:           end if
22:         end for
23:       end for
24:     end for
25:   end procedure
26:   repeat
27:     last_info ← info; info ← AssertionAnalysisStep(info, last_info, cfg)
28:   until last_info == info
29:   return info
30: end function
```

Second, 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.

Algorithm 6.4 describes the optimized version of the inner analysis step in `SafetyAssertionAnalysis` (Algorithm 5.3), which replaces `AssertionAnalysisStep` in Algorithm 5.3.

6.2.2 Context-Sensitivity of Analysis

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

Figure 6.6. Example program for inter-procedural analysis.

`SafeFunctionAnalysis` (Algorithm 5.1) is an inter-procedural analysis which constructs a distinct Safe Function (i.e., `FUN_SAFE`) theorem for every call to each callee function. We use the program in Figure 6.6 to illustrate our approach. For instance, 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` (Algorithm 5.1) needed to successfully generate a safety proof.

Context-insensitive inter-procedural analysis provides the lowest level of precision: we analyze each function once 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. 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 continues to analyze wrapper functions for syscalls using call-site context-sensitive (CSCS) inter-procedural analysis. This is because AUSPICE needs to generate a unique `FUN_SAFE` theorem to correctly consider each set of arguments passed to the syscall at each distinct call-site.

6.2.3 Other Engineering Optimizations

Next, we discuss some of the engineering optimizations throughout the AUSPICE workflow to improve the time taken to automatically generate safety proofs. The main engineering optimizations are in the caching of various intermediate results that need to be computed multiple times, so that complex computations are performed once, and their results retrieved from the cache for subsequent uses.

Caching of Proved Theorems

The safety proof generation process in AUSPICE takes as input Hoare triple theorems from the Cambridge ARM model (Line 13 in Algorithm 5.1), and tries to prove that the `FUN_SAFE` theorem holds for the particular input program. Then, proving the `FUN_SAFE` theorem involves: (i) augmenting Hoare triples for individual instructions with safety assertions, (ii) composing "safe" Hoare triples (i.e., `MEMCFISAFE` theorems) for basic blocks, and (iii) proving the "$precond \Rightarrow assert$" safety proof obligations, as required in each clause of the `FUN_SAFE` rule.

The proving of safety proof obligations first takes place in the `SafetyAssertionAnalysis` algorithm (Algorithm 5.3), when AUSPICE checks if each safety assertion can be discharged by all of its predecessors' pre-conditions during its abstract interpretation. However, the proofs of these safety obligations are required later in the `FSI_rule` function, which constructs the actual `FUN_SAFE` theorem and discharges the proof obligations automatically generated by HOL4's interactive proof process (which AUSPICE automates).

Thus, to save the time taken to search for discharging pre-conditions later in `FSI_rule`, the proofs of safety proof obligations (i.e., theorems of the form "*precond* $\Rightarrow$ *assert*") are saved in a cache by the abstract interpretation, and later retrieved in `FSI_rule` (Line 8 in Algorithm 5.1) during the construction of the `FUN_SAFE` theorem.

AUSPICE uses the `Binarymap` library in ML to implement its cache. The theorems saved to the cache are keyed by the parsed string representation of the proof goal, so that in `FSI_rule`, having access to the goal term to be proved is sufficient to retrieve the proved theorem, without having to make a repeated call to HOL4's prover to re-prove a result that has already been proved.

Caching of Parser Results

The storage and retrieval of safety proof obligations to and from the theorem cache in AUSPICE makes heavy use of HOL4's `Parser` structure to parse theorems and terms to strings in order to generate cache keys. HOL4's `Parser` structure is designed to format theorems and terms in HOL4 in colour for readable on-screen print-outs, and hence can be computationally intensive.

To save the computational effort taken to parse theorems and terms to strings, AUSPICE also caches the results of HOL4's parser for theorems and terms. Our parser cache is implemented as a linear list, and lookups need to potentially walk the entire length of the list. The cache of parsed strings has a Least-Recently-Used (LRU) eviction policy, so that we store only the N most recently used parsed strings, and we rotate each retrieved cache entry to the head of the cache on each cache retrieval.

For terms, we key our cache entries by a single term; for theorems, we key our cache entries by a list of terms, where the last term is the conclusion of the theorem, and all terms prior to the last term in the list are the hypotheses of the theorem.

In our implementation, we use a cache of 50 entries for both parsed terms and theorems. We have found, in practice, that the hit rate of our parser cache is over 90% for our configured cache size.

6.3 Support for Compiler Optimized Programs

Next, we discuss the extent to which compiler-optimized machine-code programs are supported by the logic approach of this dissertation. We discuss compiler optimizations that we have found to be amenable to the automated safety proofs by AUSPICE, i.e., machine-code programs containing the optimizations could still be proved safe by AUSPICE. We also discuss a number of compiler optimizations that cannot be supported, such that programs containing such compiler optimizations cannot be automatically proved by AUSPICE to have our desired safety properties.

For our study into the extent to which compiler optimizations are supported by AUSPICE, we make use of the compiler optimization flags supported by the `gcc` compiler. In particular, we studied the individual `-f*` compiler optimization flags that are automatically enabled at the `-O1` level of optimization for `gcc`. In future, we intend to generalize our results to study compiler optimizations implemented by different compilers. We also intend to study the performance impact of programs compiled with `gcc`'s `-O0` level of optimization as compared to its `-O1` level of optimization, to quantify the performance impact of programs needing to be compiled at the `-O0` level of optimization to render them amenable to our approach.

To investigate which compiler optimizations are supported, we identified our test programs (as described in §7) whose compiled machine-code was successfully optimized by each compiler optimization flag, i.e., their compiled machine-code was different with versus without the particular optimization flag. Then, we ran AUSPICE's safety proof generation on the optimized machine-code program to study if a safety proof could be generated for the optimized program. Based on these findings, we report the compiler optimization flags that are compatible with AUSPICE's safety proof generation, and the ones that are not.

6.3.1 Unsupported Compiler Optimizations

We begin by discussing the `gcc` compiler optimizations for which AUSPICE does not support generating safety proofs for. First, there are a number of `gcc` optimization flags that are enabled as part of the `-O1` optimization level that are not supported by AUSPICE. This led us to investigate the ability of AUSPICE to support individual `-f*` optimization flags that are enabled at the `-O1` level. We begin by discussing some of the `-f*` optimization flags enabled under `-O1` that are not supported by AUSPICE.

Branchless Conditional Jumps (`-fif-conversion`, `-fif-conversion2`)

As described in §1.7, the logic approach in this dissertation does not support safety proofs in machine-code programs containing conditionally-executed, non-branching instructions, as they induce multi-graphs in the Control-Flow Graphs (CFG) in our analysis and proof generation.

Since the `-fif-conversion` and `-fif-conversion2` optimization flags try to reduce machine-code size by transforming "conditional jumps into branch-less equivalents" [139], and non-branching conditionally-executed instructions are not supported in our logic approach, we are unable to support the automatic generation of safety proofs for these flags.

Frame Pointer Omission (`-fomit-frame-pointer`)

The safety properties that are specified and proved in our logic approach make direct use of the frame pointer: to ensure CFI, AUSPICE proves that memory writes must be to addresses smaller than the current frame pointer value (§3.1.3). The omission of the frame pointer will invalidate the specification of safety properties in AUSPICE. Hence, the `-fomit-frame-pointer` flag, which treats the frame pointer register as a regular register, will prevent AUSPICE's safety property specification and proof from taking place, as the frame pointer value is required for AUSPICE to identify safe addresses in memory where writes are allowed.

6.3.2 Supported Compiler Optimizations

Supported `-O1` Optimization Flags

Next, we studied the individual `gcc` optimization flags that are enabled at the `-O1` optimization level. We identified our test programs that are successfully optimized by each of our tested optimization flags, i.e., the sizes of the compiled programs changed after each optimization flag was enabled. Table 6.1 summarizes the test programs in our investigation, and indicates the test programs whose compilation was successfully optimized (i.e., machine-code changed) by each compiler optimization flag.

Optimization Flag	`arrcpy`	`sort`	`stringsearch`	`matmult`
`-fmerge-constants`			✔	
`-fsplit-wide-types`				✔
`-ftree-ter`	✔	✔	✔	

Table 6.1. Test programs that were successfully optimized by each `gcc` compiler optimization flag.

We do not discuss the reasons why each particular optimization flag in Table 6.1 was able to optimize each particular test program, as the effects of compiler optimizations are beyond the scope of this dissertation.

With that, we briefly describe each optimization flag. The `-fmerge-constants` flag merges constants that may be defined at multiple different program points, but which are identical, to save space. The `-fsplit-wide-types` flag enables types such as "`long long`" on a 32-bit architecture to be split up, allocated, and stored in non-contiguous registers. The `-ftree-ter` flag performs temporary expression replacement during the code generation process for single-use expressions to enable more efficient code generation.

We found that the above flags did not interfere with the ability of AUSPICE in our logic approach to still successfully generate safety proofs for our test programs, as they did not change the compiled machine-code of our safety-checks, or the compiled machine-code of the suspect statements that required safety-checks.

The above list of supported compiler optimization flags is non-exhaustive, as it is challenging to produce test programs that can successfully exercise each optimization flag that we wish to investigate. In general, compiler optimizations that do not affect the compiled machine-code of our safety-checks or of suspect statements, and that do not eliminate or optimize away important information, such as the frame pointer, that is used in the safety proof process, will be supported.

Register Allocation

Register Allocation (§9.1 in [140]) refers to the process during compilation in which a set of program variables are selected to reside in registers at a point in the program (as compared to residing in memory on the function's stack). While Register Allocation is not specifically enabled via a compiler flag in `gcc`, it is an important optimization during compilation.

The logic approach in this dissertation is able to support safety-proof generation for programs with variables that (i) are allocated to reside in registers, that also (ii) contain the destination address for a memory write instruction (i.e., the variable is used as a pointer). For register-allocated local variables that are not pointers, no safety assertions are generated, since the manipulation of primitive variable types (e.g., `int`s) do not make use of loads and stores. For register-allocated local variables that are pointers, when the address that the pointer variable is pointing to is written, a safety-check will be prescribed by PCFIRE-C. Then, with the appropriate safety-check in place, our logic approach will be able to automatically prove that the memory-write is safe, as this memory-write will

look similar to other unsafe memory-writes that make use of an intermediate register to store the computed memory-write address, except that a register that is associated with a specific source-level variable is used instead of an intermediate register.

6.4 Summary

In this chapter, we described a number of extensions to AUSPICE in our logic approach to support features in real-world programs.

First, we described an axiomatic approach to automatically prove safety for ARM machine-code programs containing system calls. We modeled the user-mode-visible outcomes of an OS kernel servicing a system call on behalf of the user-mode program. Then, we automatically generated safety assertions for these system call invocations based on: (i) the POSIX specification of the system call's behavior, and (ii) the arguments supplied to the system call, as extracted from the Hoare triples leading up to the system call. These safety assertions are initially symbolic and opaque, because the semantics of the system call are available only at the inter-procedural level, and we use a delayed approach to concretize these safety assertions later in the analysis, before performing automatic discharge of the safety proof obligations.

Second, we described how we optimized the automated safety proof process to speed up the process, and enable AUSPICE to support larger programs. We leveraged common compiler conventions to abbreviate and reduce the number of proof terms required to reason about writes to local variables on the stack of a function, and we used a novel form of context-sensitive analysis known as "Single-Function Context-Sensitive" (SFCS) analysis to reduce the number of analyses required for large programs. Then, we described how we made use of caching in AUSPICE to reuse computation results and hence save computation time.

Finally, we investigated the ability of AUSPICE to support compiler-optimized machine-code programs. We identified two compiler optimization flags in gcc's `-O1` level of optimization that are not compatible with AUSPICE's proof automation as they drop critical pieces of information required for AUSPICE's safety proofs. We also empirically identified three compiler optimization flags that are enabled as part of `gcc`'s `-O1` level of optimization, that AUSPICE is compatible with, and we explain how AUSPICE is able to prove safety in programs that have local variables allocated to registers.

Chapter 7

Experimental Evaluation and Case Studies

This chapter presents the results of the experimental evaluation of our approach to Control-Flow Integrity (CFI). To evaluate our approach, we sought to investigate the kinds of programs that our CFI approach can be applied to, to both prescribe source-code CFI safety-checks (our enforcement approach), as well as automatically prove the CFI of the resulting programs (our logic approach). We sought to evaluate our approach by the kinds and sizes of programs supported.

Research Questions

The research questions that we sought to answer in our experimental evaluation were:

- Can our CFI approach support programs with commonly-used C constructs? (§7.1)
- Can our CFI approach support programs containing simple file-based input/output (I/O) behavior? (§7.2)
- Can our CFI approach support embedded programs containing hardware I/O behavior? (§7.3)
- Can the scalability of our CFI approach be improved by our proposed proof automation optimizations? (§7.4)
- Can our CFI approach be applied to security vulnerabilities? (§7.5)

At the same time, in each of our research questions, we also sought to evaluate the feasibility of our approach, as measured by the times taken to generate our safety-check prescriptions, and to prove the CFI of programs that our approach has been applied to. In addition, we also sought to understand the costs associated with our approach to CFI, in terms of the run-time overheads of our

prescribed safety-checks, as well as the increased source-code and machine-code sizes of programs containing our prescribed safety-checks.

Overall Methodology

We begin by describing our overall methodology for answering the first three of our research questions. First, we selected a number of representative test programs for each of the three classes of programs, for which we wish to demonstrate that our CFI approach supports. Second, for each of our test programs, we provide CFI enforcement using PCFIRE-C [126] (§3) to obtain prescriptions of safety-checks for enforcing CFI. We manually insert these prescribed safety-checks into our test programs. Third, we use AUSPICE [128] (§4, 5, 6) to automatically generate safety proofs of the CFI of the test programs.

To evaluate the feasibility of our approach, we measured the following: (i) the times taken for PCFIRE-C to construct its safety-check prescriptions, and (ii) the times taken for AUSPICE to complete its safety proof generation. To measure the costs of our approach, we measured the run-time overheads introduced by the safety-checks prescribed by PCFIRE-C, and the increases in the sizes of the source-code and machine-code of our test programs.

7.1 Benchmark Programs

First, we evaluate the ability of our CFI approach to provide CFI enforcement, as well as automatically prove the CFI, of C programs containing a variety of C constructs. Our aim is to demonstrate that our CFI approach can support a wide variety of C constructs (and their corresponding compiled machine-code), and thus can be applied to different kinds of real-world programs with various C constructs.

We begin by describing our evaluation methodology and experimental setup (§7.1.1). Then, we present and discuss our experimental results on the times taken by our approach to prescribe safety-checks (§7.1.2) and to generate CFI safety proofs (§7.1.3). Finally, we measure and present the space overheads of our added safety-checks (§7.1.4), and the run-time overheads of the safety-checks inserted in our test programs (§7.1.5).

7.1.1 Evaluation Methodology

To demonstrate the variety of C programs supported, we evaluate our approach on C programs selected from two benchmarks: (i) the MiBench [33] suite of commercially representative benchmark embedded programs, that contains embedded programs written in C that implement a number of common algorithms and tasks, and (ii) the WCET Benchmarks [34] which, while designed for benchmarking WCET algorithms, contains programs with various different C constructs, and are suitable for our evaluation. We selected 2 programs from the MiBench suite, and we selected 20 programs from the WCET benchmark suite. We excluded 15 programs from the WCET benchmark suite that were incompatible with our approach, as they contained features such as recursive function calls, floating point instructions, and unstructured control-flow jumps. In addition, we also wrote a small number of test programs containing particular C constructs, and we also include an implementation of the `memcpy` function from the Bionic C library for Android to demonstrate the ability of our approach to support real-world C programs.

Table 7.2 describes our test programs, and characterizes our test programs by the features found in each test program, and the sizes of the compiled machine-code and the number of source-code lines.

Experimental Setup

Each test program was compiled using `gcc` 4.6.3. All test programs were compiled with debug symbols included, and all test programs were statically linked, with no external libraries.

All safety-check prescriptions were generated on an Intel 2.6 GHz Core i7 (Quad-core) with 16 GB of RAM. All safety proofs were generated on an `r3.large` instance on the Amazon Elastic Compute Cloud (EC2), which has 2 virtual CPUs on the Intel Xeon E5-2670 v2 (Ivy Bridge) processor and 15.25 GB of RAM.

To evaluate the overheads of our inserted safety-checks, we ran our test programs on the Raspberry Pi 1 Model B+ with a 700 MHz ARMv6 processor and 512 MB RAM with Linux 3.18.

7.1.2 Results: Safety-Check Prescription Times

Table 7.3 summarizes the results of our evaluation of our enforcement approach, as measured by the time taken by PCFIRE-C [126] to prescribe safety-checks for each program.

The times taken by PCFIRE-C to prescribe safety-checks for each program was less than 1 second for most (20 out of 26) programs. PCFIRE-C took the longest time (7 minutes and 16 seconds) to prescribe safety-checks for the `nsichneu` program, which had significantly more source-code lines (3136 lines of C code before safety-checks were added) than the other test programs. Other than the `nsichneu` program, PCFIRE-C took 53 seconds to prescribe safety-checks for `crc32`, because `crc32` had a large input array included its code, and PCFIRE-C took 39.5 seconds and 8.4 seconds respectively to prescribe safety-checks for `statemate` and `adpcm`, which had 1276 lines and 706 lines of C code respectively. We believe that the time taken by PCFIRE-C to generate safety-check prescriptions for most programs makes it feasible for use by programmers.

7.1.3 Results: Safety Proof Times

Table 7.3 also summarizes the results of our evaluation of our logic approach, as measured by the time taken by AUSPICE [128] to generate a CFI safety proof for each program.

The times taken by AUSPICE to generate CFI safety proofs for each program ranged from 9 minutes for `arrcpy`, to 18.65 hours for `adpcm`, to 322.5 hours for `statemate`. For the remaining programs, AUSPICE took between 10 to 30 minutes to generate CFI safety proofs for small programs (e.g., `arrcpy`, `sort`, `memcpy`, `crc32`, `ns`, `fir`, `prime`), and between 1 to 2 hours to generate CFI safety proofs for the larger, more complex (as measured by the number of CFG edges and nodes in each program) programs (e.g., `stringsearch`, `jfdctint`, `edn`, `fdct`). We believe that the time taken by AUSPICE to generate CFI safety proofs is feasible for use by programmers. We believe that the run-times of AUSPICE are acceptable given that the proofs that it generates are strong (in a formal logic) and foundational (about the behavior of machine-code, and built on trustworthy, validated semantics as provided by the Cambridge ARM model [131, 31, 32]).

The CFI safety proof generation did not complete in a timely manner for `nsichneu`: AUSPICE's abstract interpretation algorithm for the automatic discharge of proof obligations (Algorithm 5.3, as described in §5.4), completed in 53.5 hours. However, HOL4's set simplifier (`pred_setLib`), which instantiates the `FUN_SAFE` rule with the Control-Flow Graph information supplied to it, was not able to generate all the proof obligations that need to be discharged for proving `nsichneu` safe: `pred_setLib`'s built-in set simplifier could not complete the generation of all proof obligations within 720 hours (30 days), at which point we stopped the proof generation process. This is because `nsichneu` consists of a single function that contains over 3000 lines of C code, and over 21000 machine-code instructions, with an extremely complex Control-Flow Graph (4147 nodes,

7579 edges). In addition, the inordinately long proof time for `statemate` was due to a number of functions having complex Control-Flow Graphs as well (e.g., the three most complex functions each respectively had: 945 nodes / 1615 edges, 809 nodes / 1416 edges, 378 nodes / 652 edges), causing HOL4's set simplifier to take an extremely long time to generate proof obligations, accounting for a vast majority of the proof-generation time.

Comparison with other CFI Techniques

We compare the time taken by AUSPICE in our logic approach to generate CFI safety proofs, to the time taken by ARMor [18] to generate its SFI safety proof, for the `stringsearch` program from the MiBench [33] benchmark.

ARMor reported taking 8 hours to generate its safety proof for `stringsearch` on an Intel Core i7 2.7 GHz using HOL4. In contrast, AUSPICE took 1.06 hours to generate its CFI safety proof for the `stringsearch` program. Hence, AUSPICE is able to generate safety proofs automatically in a much shorter amount of time than prior techniques that generate strong and foundational safety proofs.

7.1.4 Results: Space Overheads

In addition to the times taken to prescribe safety-checks and generate safety proofs, Table 7.3 also summarizes the increase in the sizes of the source-code and compiled machine-code of our test programs.

The additional number of lines of source-code introduced due to safety-checks is determined by the number of suspect statements in each program that require additional safety-checks. Hence the number of additional lines of source-code is dependent on the specific behavior of each program.

We argue that the increase in the size of the machine-code of each test program is the more important metric, as it determines the amount of memory required for loading each program to memory before it can be executed. From Table 7.3, we can see that the increase in the size of the compiled machine-code of each program ranged from 0% for benchmark programs that did not need any safety-checks (e.g., `ns`), to 156% for the `nsichneu` program, which had a large number of unsafe memory writes to global variables. For most of the remaining test programs, the increase in the sizes of their compiled machine-code ranged from 2% (`fir`) to 59% (`compress`), with most programs experiencing an increase in compiled machine-code size of between 15% and 50%. We

believe that the increases in the sizes of our compiled test-programs when safety-checks are added is modest, and that our approach is feasible for use with most common programs.

7.1.5 Results: Run-time Overheads

Program	Source	Run-time Slowdown
`arrcpy`	Own	113.3%
`sort`	Own	72.3%
`matmult1`	Own	0.78%
`memcpy`	Bionic [141]	800.8%
`stringsearch`	MiBench [33]	213.2%
`crc32`	MiBench [33]	43.1%

Table 7.1. Run-time slowdowns in our test programs after safety-checks were introduced.

Next, we evaluate the run-time costs of the CFI safety-checks in our approach by measuring the run-time slowdowns in our benchmark programs due to our introduced safety-checks. We selected only our self-written programs, the `memcpy` implementation in the Bionic C library [141], and our two test programs from the MiBench embedded software benchmark [33]. This is because the WCET benchmark programs were designed for testing source-code-based WCET analysis tools, as a result, they had some characteristics that made them unsuitable for studying their run-time performance, e.g., they had very small input data sizes, and their run-times were extremely short, making it difficult to measure the run-time overheads meaningfully.

Table 7.1 summarizes our results of measuring the run-time overheads of a selection of our test programs. For each test program, we measured its run-time over 1000 iterations, and we compared the run-time for each program with and without safety-checks inserted. Then, the run-time slowdown is calculated as:

$$\text{slowdown} = \frac{(\text{runtime with safety-checks}) - (\text{runtime without safety-checks})}{(\text{runtime without safety-checks})}$$

From our results in Table 7.1, we can see that the run-time slowdown ranged from 0.78% for `matmult1` to 800.8% for `memcpy`. The amount by which each test program's run-time slowed down depended on the proportion of each test program's workload that invoked suspect statements for which safety-checks were prescribed, and added. `matmult1` had the smallest run-time slowdown, as its workload was mainly computation. `memcpy` had the highest run-time slowdown, as it is a worst-case scenario for PCFIRE-C's safety-checks in terms of run-time performance, since

`memcpy`'s workload is made up entirely of memory writes that require safety-checks. Hoisting our memory-address-write checks out of the loop would help reduce our run-time overheads, although this would require manually-specified loop invariants in our safety proofs. For test programs with mixed workloads (e.g., `stringsearch`, `crc32`, `sort`), the run-time slowdown ranged from 43% to 213%.

Comparison with other CFI Techniques

We contrast the run-time slowdowns due to CFI enforcement mechanisms in previous CFI techniques, with the run-time slowdown due to our prescribed CFI safety-checks.

Previous CFI techniques that detect CFI violations after-the-fact, such as Abadi et al. [15], XFI [16], and CCFIR [25], incurred average run-time slowdowns of 16%, 11%, and 3.6% respectively. On the other hand, ARMor [18], like PCFIRE-C, uses safety-checks (albeit for SFI [46] rather than CFI) that are preventative in nature (although ARMor does not allow for customizable recovery actions). ARMor incurred run-time slowdowns of 240% for the `stringsearch` program from the MiBench [33] benchmark, which is comparable to our run-time slowdown of 213% for the same program.

This suggests that preventative CFI techniques, such as PCFIRE-C, fundamentally incur higher overheads as compared to CFI techniques that detect violations after-the-fact. This is because preventative CFI techniques require safety-checks at all suspect memory-writes, whereas after-the-fact CFI techniques need safety-checks only at indirect jumps, and there are typically many more memory-writes than indirect jumps in a program.

Nonetheless, we believe that the trade-off of higher run-time slowdowns, in exchange for the ability to prevent the root-causes of CFI violations, and hence enable customizable recovery actions, is acceptable for applications that may be safety-critical that require robust recovery actions when (potential) CFI violations occur.

	Program	Source	S	L	N	A	B	Bytes	LOC	Description
1	`arrcpy`	Own	✔	✔		✔		1442	17	Array-copy example in §3.1.1.
2	`sort`	Own		✔	✔	✔		1390	25	Our implementation of the selection sort algorithm.
3	`matmult1`	Own	✔	✔	✔			3481	30	Our implementation of matrix multiplication.
4	`memcpy`	Bionic [141]	✔	✔		✔		4238	98	Efficient implementation of `memcpy` from the Bionic C library for Android.
5	`stringsearch`	MiBench [33]		✔	✔	✔		23532	68	Implementation of the Boyer-Moore string search algorithm.
6	`crc32`	MiBench [33]		✔		✔		5480577	91	Implementation of the CRC32 checksum algorithm.
7	`nsichneu`	WCET [34]		✔				45691	3136	Automatically-generated code simulating an extended Petri Net.
8	`statemate`	WCET [34]		✔				23871	1276	Automatically-generated code from the State-Chart Real-time Code Generator, STARC.
9	`adpcm`	WCET [34]		✔				21361	706	Implementation of the Adaptive Pulse Code Modulation algorithm.
10	`ns`	WCET [34]		✔	✔	✔		7862	436	Search in a multi-dimensional array (returns from middle of a 4-deep nested loop).
11	`compress`	WCET [34]		✔	✔	✔		11041	413	Data compression, adapted from the SPEC95 benchmark.
12	`jfdctint`	WCET [34]	✔	✔		✔		5293	309	Discrete-cosine transformation on an 8x8 pixel block.
13	`fir`	WCET [34]		✔	✔	✔		10121	263	Implementation of the Finite Impulse Response filter signal processing algorithm.
14	`edn`	WCET [34]	✔	✔	✔	✔	✔	9558	256	Implementation of filter calculations for the Finite Impulse Response filter operation.
15	`fdct`	WCET [34]	✔	✔		✔	✔	6011	171	Implementation of the Fast Discrete Cosine Transform algorithm.
16	`matmult`	WCET [34]	✔	✔	✔	✔		4018	119	Matrix multiplication of two 20 x 20 matrices.
17	`ud`	WCET [34]	✔	✔	✔			5010	138	Calculation of matrices with 3-deep nested loops.
18	`expint`	WCET [34]	✔	✔	✔			4012	133	Series expansion for computing an exponential integral function.
19	`cnt`	WCET [34]		✔	✔	✔		4329	69	Count non-negative numbers in a matrix.
20	`crc`	WCET [34]	✔	✔		✔	✔	4438	110	CRC checksum computation on 40 bytes of data.
21	`bsort100`	WCET [34]		✔	✔	✔		4074	96	Implementation of the bubble sort algorithm.
22	`bs`	WCET [34]		✔		✔		2773	86	Binary search of an array of 15 integer elements.
23	`prime`	WCET [34]	✔	✔				3739	46	Calculation of whether a given number is prime.
24	`insertsort`	WCET [34]		✔	✔	✔		2416	64	Implementation of the insertion sort algorithm.
25	`fibcall`	WCET [34]	✔	✔				2243	64	Iterative calculation of the Fibonacci series, used to calculate `fib(30)`.
26	`janne_complex`	WCET [34]	✔	✔	✔			2337	57	Program with nested loops.

Table 7.2. Summary of benchmark programs that we evaluated our CFI approach on, and the benchmark suite each program was from ("Own" refers to self-written test-cases). WCET benchmark program descriptions adapted from [34]. Legend: S = always single path program (no potential flow dependency on external variables), L = contains loops, N = contains nested loops, A = uses arrays and/or matrices, B = uses bit operations. Bytes refers to number of bytes in compiled program (without safety-checks), LOC = lines of C code in program (without safety-checks).

Program	Source	Source-code LOC (without safety-checks)	Source-code LOC (with safety-checks)	Machine-code bytes (without safety-checks)	Machine-code bytes (with safety-checks)	Size Increase	PCFIRE-C Prescription Time	AUSPICE Proof Time
arrcpy	Own	17	33	1442	1714	18.9%	0.22 s	9.3 m
sort	Own	25	46	1390	1718	23.6%	0.25 s	13.3 m
matmult1	Own	30	47	3481	3605	3.6%	0.56 s	2.87 h
memcpy	Bionic [141]	98	143	4238	4891	15.4%	0.36 s	27.4 m
stringsearch	MiBench [33]	68	100	23532	23592	0.25%	0.71 s	1.06 h
crc32	MiBench [33]	91	119	5480577	5480841	0.005%	53 s	19.2 m
nsichneu	WCET [34]	3136	3662	45691	117079	156%	7 m 16 s	DNF
statemate	WCET [34]	1167	1304	23879	63627	162%	39.5 s	322.5 h
adpcm	WCET [34]	706	826	21361	33573	52.7%	8.4 s	18.65 h
ns	WCET [34]	436	436	7862	7862	0%	0.2 s	10.66 m
compress	WCET [34]	413	447	11041	17529	58.8%	2.4 s	9.21 h
jfdctint	WCET [34]	309	345	5293	6953	31.4%	0.56 s	1.65 h
fir	WCET [34]	263	270	10121	10325	2.0%	0.35 s	22.2 m
edn	WCET [34]	256	312	9558	12070	26.3%	1.0 s	1.89 h
fdct	WCET [34]	171	208	6011	7363	22.5%	0.59 s	1.98 h
matmult	WCET [34]	123	138	4034	4782	18.5%	0.27 s	44.5 m
ud	WCET [34]	144	168	5030	6152	22.3%	0.36 s	49.7 m
expint	WCET [34]	133	133	4012	4012	0%	0.16 s	18.0 m
cnt	WCET [34]	69	80	4329	5057	16.8%	0.30 s	48.7 m
crc	WCET [34]	110	127	4438	5219	17.6%	0.30 s	31.5 m
bsort100	WCET [34]	103	111	3352	3916	16.8%	0.33 s	16.6 m
bs	WCET [34]	86	86	2773	2773	0%	0.16 s	3.4 m
prime	WCET [34]	46	51	3739	3999	7.0%	0.22 s	28.65 m
insertsort	WCET [34]	65	78	2416	3864	60.0%	0.32 s	52.6 m
fibcall	WCET [34]	64	64	2243	2243	0%	0.16 s	3.3 m
janne_complex	WCET [34]	57	57	2337	2337	0%	0.16 s	4.0 m

Table 7.3. Evaluation results of source-code and machine-code size increases and prescription and proof times for our benchmark programs. (“DNF” = Did not finish in less than 30 days.)

7.2 File-based I/O

Next, we investigate how our approach to CFI in this dissertation can be applied to user-mode programs that contain file input/output (I/O) behavior. For user-mode programs to perform file I/O operations, they need to be able to invoke system calls (syscalls), as the underlying operating system (OS) needs to perform the file I/O operations on behalf of the user-mode program. In §6.1, we described how we extended AUSPICE in our logic approach to support automated safety-property proofs for target programs that contain syscall invocations. At the same time, these target programs also need to be developed in a way that renders them amenable to the automated generation of safety-property proofs in AUSPICE.

In §7.2.1, we describe the process of developing our target programs that contain syscalls, so that our target programs are amenable to the automated safety-property proof process in AUSPICE. At a high-level, we did not make use of standard C library functions, and we directly called our syscall wrappers, as described in §7.2.1. To simplify our software development process, we wrote simple helper functions around our syscall wrappers.

Then, in §7.2.2, we describe our target programs and the experimental setup for evaluating the ability of our CFI approach to support programs containing file-based I/O behavior, and we discuss the results of our evaluation in §7.2.3 and §7.2.4. Finally, we investigate the relationship between input buffer sizes and the trade-off between proof times and run-time overheads of our test programs in §7.2.5.

7.2.1 Developing Automatically-Provable Programs with File-based I/O

In typical software development projects that use the C programming language, the development environment consists of a compiler suite, such as `gcc`, which performs a sequence of steps such as preprocessing of directives, compilation of source-code to (relocatable) object-code, and linking of object code to produce machine-executable binary programs. In addition, the development environment, often in cooperation with the underlying OS, typically provides libraries for performing common functions. For software development in C, compiler suites such as `gcc` are often designed to work with a "standard" library, such as the GNU `libc` library. Such a standard library typically provides various functionalities, including file-based I/O, and any syscall invocations required to perform I/O functionality are usually transparently performed on behalf of the programmer in the standard library.

Before describing our experimental evaluation of the application of the approach to CFI in this dissertation to programs containing file-based I/O, we describe some of the practical steps taken to develop programs that support file-based I/O, so that the resulting programs are amenable to the automated safety proof generation of the logic approach in this dissertation.

Limiting Unsupported and Opaque Code

The logic approach in this dissertation generates safety proofs whose reasoning about inter-procedural program behavior is precise and context-sensitive: all functions and instructions that are reachable from the entry function to the program must be reasoned about in a program's safety proof, as explained in §4.2.4. As a result, all instructions that can be executed in a program must be statically linked and included in the program binary (as stated in §1.3), and all instructions that are statically linked and included in the program binary must meet the requirements stated in §1.3, 1.7 (e.g., no multi-threaded behavior).

In a typical C project, a compiler suite such as `gcc` automatically links to and includes support code from a C standard library such as `libc`, which performs startup tasks to prepare the user program for execution. Concretely, the `gcc` compiler calls the GNU `ld` linker with a default linker script. This default linker script specifies that the startup function in the final program executable will be the `libc`-provided startup function that: (i) prepares data structures for multi-threading, and (ii) copies command-line arguments into the stack of the entry function of the user program, and (iii) calls the entry function (i.e., `main()`).

However, the default `libc` startup function performs a number of tasks, many of which contain behavior unsupported by the logic approach in this dissertation (e.g., multi-threading behavior). In addition, the startup function typically calls a number of other functions, and this adds to the number of instructions that would need to be verified by our logic approach if we tried to prove the CFI of programs compiled and statically linked with the default `libc` startup code (and all of the other functions it calls).

In this dissertation, we take a "clean-slate" approach to the software compilation process to eliminate the standard `libc` support code that: (i) adds to the scalability challenge for our logic approach by introducing additional instructions whose behavior needs to be verified, and (ii) contains behavior (e.g., concurrency and multi-threading related behavior) that is not supported by our logic approach. Instead, to minimize opaque behavior of the kind introduced by GNU `libc` in our

target programs, and to minimize the size of the programs verified in our logic approach, we take the following steps:

1. We provide a custom GNU `ld` linker script which avoids running GNU `libc`'s program initializer (`__libc_init()`) before a user program's `main()` function.

2. We provide a custom initializer function, consisting of 20 lines of C code, that only prepares command-line arguments (i.e., `argc`, `argv`) for the target program. This minimizes the amount of code that is added to the target program.

Enabling System Calls and File I/O Behavior

Next, to perform file-based I/O, user-mode programs need to invoke syscalls to request the underlying OS to perform the desired I/O operations on behalf of the user-mode program. As we require all our test programs to be statically compiled and linked (§1.3), and typical C standard library implementations such as GNU `libc` have large library functions that provide rich functionality, using `libc`-provided I/O helper functions will result in large amounts of library code being added to our target programs that our logic approach will need to prove the safety of.

Instead, in this dissertation, we also adopt a "clean-slate" approach to enabling our target programs to carry out file-based I/O behavior. Instead of relying on GNU `libc`-provided helper functions to perform I/O operations, we have implemented thin wrappers of hand-written assembly code around a small number of common syscalls that we believe are needed to support basic file-based I/O operations. Each assembly code wrapper prepares the arguments for the syscall for the syscall invocation, and is between 15 to 20 lines long. This minimizes the amount of code that is statically compiled and linked into each test program, thus providing AUSPICE in our logic approach with compact programs that support file-based I/O, without introducing excessively many instructions that need to be proved safe.

We then provide C function prototypes for these wrappers in a C header file which can be included by programmers to enable them to use these wrappers as regular C functions in their programs to perform file-based I/O operations. We implemented assembly code wrappers for the following syscalls: `open`, `close`, `read`, and `write`, to support file-based I/O.

We envision that in future, this "clean-slate" approach to file-based I/O can be extended to writing helper functions that provide similar functionality as some of the helper functions in the GNU `libc` standard library, but with added safety-checks that are prescribed by PCFIRE-C in our

enforcement approach, so that the resulting library code can be proved to be safe by AUSPICE in our logic approach.

In addition to enabling programmers to directly invoke syscalls, we also wrote simple versions of the following helper functions typically provided by GNU `libc`: `strlen`, `strpos`, `strrev`, `itoa`, `atoi`, `memset`, `memcpy`. We completed the implementation of each function before using PCFIRE-C to prescribe safety-checks for each function, after which we inserted the safety-checks in each function. We also implemented a helper function, `read_line`, which reads up to a given number of bytes from a given file descriptor into a given buffer, until a newline character is encountered in the input stream.

7.2.2 Evaluation Methodology

To evaluate the ability of our CFI approach to provide CFI enforcement and safety proof generation for programs with file-based I/O behavior, we implemented simple versions of three common file I/O utilities found in Linux. We implemented simple versions of the `cat`, `wc`, and `grep` text utilities using the software development methodology described in §7.2.1. The functionality of each text utility is as follows:

- `cat`: Outputs contents of a file.
- `wc`: Counts number of words in a file.
- `grep`: Prints lines in a file containing given a string.

These programs contained the `read`, `write`, `open`, and `close` syscalls.

When implementing our three text utilities, we first implemented each text utility, after which we used PCFIRE-C to generate prescriptions of safety-checks required for each utility (which also generated the safety-checks required for statically linked helper functions). We then measured the time taken by PCFIRE-C to prescribe safety-checks, and the time taken by AUSPICE to generate safety proofs of CFI for each test program. We also measured the run-times of each utility with and without the prescribed safety-checks to measure the run-time overheads due to the safety-checks.

Experimental Setup

We wrote each utility in C, and we compiled each program using `gcc` 4.6.3 on Linux 3.18 on the ARM platform. All programs were compiled with debug symbols included, and all programs were statically linked.

Similarly to our experiments in §7.1, all safety-check prescriptions were generated on an Intel 2.6 GHz Core i7 (Quad-core) with 16 GB of RAM. All safety proofs were generated on an `r3.large` instance on the Amazon Elastic Compute Cloud (EC2), which has 2 virtual CPUs on the Intel Xeon E5-2670 v2 (Ivy Bridge) processor with 15.25 GB of RAM.

To evaluate the overheads of the inserted safety-checks, we ran our test programs on the Raspberry Pi 1 Model B+ with a 700 MHz ARMv6 processor and 512 MB RAM with Linux 3.18.

7.2.3 Results: Safety Proof Times and Space Overheads

Program	Source-code LOC (without safety -checks)	Source-code LOC (with safety -checks)	Machine-code bytes (without safety -checks)	Machine-code bytes (with safety -checks)	Size Increase	PCFIRE-C Prescription Time	AUSPICE Proof Time
`cat`	275	337	6400	6780	5.9%	0.24 s	43.6 mins
`wc`	292	354	11680	12608	7.9%	0.57 s	2.15 h
`grep`	293	355	8434	8858	5.0%	0.48 s	1.07 h

Table 7.4. Safety-check prescription times, safety proof generation times, and source-code and machine-code size increases due to safety-checks for file-based I/O programs.

Next, we present the results of our evaluation of our CFI approach on our file-based I/O utilities. Table 7.4 summarizes the results of our evaluation for our test programs containing file-based I/O.

The compiled binary files for our file-based I/O utilities were between 6.7 kilobytes to 12.6 kilobytes in size. The added safety-checks, as prescribed by PCFIRE-C, resulted in the compiled machine-code of our file-based I/O utilities increasing by between 5% and 8%. PCFIRE-C took less than 1 second to prescribe safety-checks for all of our file-based I/O programs. AUSPICE took less than 2.15 hours to automatically prove the safety of each of our test programs.

Hence, our results show that: (i) the added safety-checks in our enforcement approach cause only small increases in the sizes of our file-based I/O utilities, (ii) the PCFIRE-C tool in our enforcement approach prescribed safety-checks in very short amounts of time, and (iii) AUSPICE in our logic approach can prescribe safety-checks and prove safety automatically in realistic programs that implement useful I/O functionality using syscall invocations.

7.2.4 Results: Run-time Overheads

Table 7.5 summarizes the run-time overheads of the safety-checks in our file-based I/O utilities. We ran each program on a 10 MB input file over 5 iterations and we report the average run-times. We

	Run-times			Slowdown	
Program	Safe	Unsafe	Original	vs. Unsafe	vs. Original
`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 7.5. Run-time overheads of file-based I/O utilities.

report the run-time slowdown of each "safe" utility, as compared to an "unsafe" version without the PCFIRE-C-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-C's safety-checks in more realistic programs with a mixed workload is likely to be much less than shown in §7.1.5 for our benchmark programs.

Since our goal was to evaluate our safety-checks on programs with I/O behavior, 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.

7.2.5 Impact of Input Buffer Size on Proof Times and Run-time Overheads

Buffer Size	Proof Time	Run-time Slowdown vs. Original
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 7.6. Improved run-time slowdown but slower proof times with larger buffer sizes for `cat`.

Next, we investigated the cause of the discrepancy between the run-times of our implementations of file-based I/O utilities, and the system-provided versions of our file utilities. If the cause of the slower run-times of our implemented file-based I/O utilities were due to our PCFIRE-C-prescribed safety-checks, the run-time performance of the "unsafe" versions of our implemented utilities (i.e., which did not contain the PCFIRE-C-prescribed safety-checks) would be similar to that of the system-provided versions of each utility. However, the "unsafe" versions of our file-based I/O utilities performed significantly worse than their system-provided versions.

We hypothesized that the cause of the slower run-times in our versions of the our utilities, as compared to the system-provided versions, was due to differences in the way file I/O was handled in our implementations, as compared to in the GNU `libc` C standard library. In particular, `libc` provides a wrapper around the `read` syscall which performs buffered reading of data from a given file descriptor, whereas in our implementation, we directly invoked the `read` syscall with no input buffering.

To validate our hypothesis, we implemented a buffered version of our `read_line` function for reading input from a file descriptor, with a configurable buffer size. Then, we measured the effects of increasing input buffer sizes on the run-times of safe versions of our implementation of the `cat` utility.

Table 7.6 summarizes our results. As the buffer size in `read_line` is increased from 1 to 30 bytes, the run-time slowdown for `cat` improved significantly from 2549% to 56.5%, even with PCFIRE-C-prescribed safety-checks included in the program. On the other hand, the safety proof time increased by 4x to 165.6 minutes. The safety proof time increased with larger input buffers as AUSPICE needs to check that each byte in the input buffer is safe. On the other hand, the run-time of `cat` improved as the input buffer size is increased, as the high cost (in terms of number of processor cycles used) of making a context switch to invoke the `read` syscall is amortized over a larger number of characters read.

Hence, there is a trade-off between run-time performance and safety proof times in AUSPICE for file-based I/O utilities, due to the inverse relationship between verification cost, as measured by the safety-proof generation time of AUSPICE, and run-time overheads.

7.3 Hardware I/O

Next, we investigate how our approach to CFI in this dissertation can be applied to embedded software that contains hardware input/output (I/O) behavior. In particular, we evaluate our support for embedded software that performs hardware I/O using the Raspberry Pi single-board-computer. We develop our test programs for the Raspberry Pi that interface with various hardware input sources and hardware outputs via the General Purpose I/O (GPIO) interface on the Raspberry Pi.

We begin by describing how we developed our test programs that support hardware I/O behavior, that are also amenable to the automated safety proof generation in our approach (§7.3.1). Then, we

describe our evaluation methodology and experimental setup (§7.3.2). Next, we present and discuss the results of our evaluation (§7.3.3).

7.3.1 Support for Hardware-based I/O

To support the development of embedded software that contains hardware I/O behavior, we extended our approach in §7.2.1 for supporting file I/O behavior.

First, for our evaluation in this dissertation, we selected the Raspberry Pi single-board-computer, as it uses an ARMv6 processor (where ARMv6 is the ISA supported by our logic approach), and because it provides an easy-to-use General Purpose I/O (GPIO) digital interface for connecting to hardware input devices such as sensors, and output devices. In addition, the ARM architecture is widely used in Internet-of-Things devices [142].

Second, to support hardware I/O in user-mode programs on the Raspberry Pi, we make use of memory-mapped I/O to address the Raspberry Pi's GPIO interface. Thus, to enable user-mode programs to perform memory-mapped I/O, we make use of our assembly wrapper for the `open` syscall (as described in §7.2.1), and we wrote additional wrappers for the `mmap` and `munmap` syscalls. In addition, we also wrote an assembly wrapper for the `nanosleep` syscall to enable user-mode programs to request delays of its own execution from the OS, e.g., to implement polling behavior.

Third, to ease the process of writing C code to perform hardware I/O operations, we adapted the WiringPi C library [143]. WiringPi is a convenience library to help programmers write C programs that interface with hardware for the Raspberry Pi. We adapted a small subset of the WiringPi library to invoke syscalls via our assembly wrappers instead of via the interface provided by the GNU `libc` C standard library. This required minimal changes to the subset of WiringPi that we adapted. In addition, we used PCFIRE-C to prescribe safety-checks for the subset of the WiringPi library that we adapted, and we added the prescribed safety-checks to our adapted library.

7.3.2 Evaluation Methodology

Next, we implemented 4 programs containing hardware inputs and outputs on the Raspberry Pi to evaluate our CFI approach in this dissertation. Our test-programs contained the `mmap`, `munmap`, `open`, `close`, and `nanosleep` syscalls. Our 4 test programs are:

- `blink`: Controls an LED, and turns it on and off repeatedly to blink the LED.

- `led`: Uses an analog light sensor (with a capacitor arranged in series) to sense the amount of ambient light, and turns on an LED when the amount of ambient light sensed falls below a given threshold.
- `lcd`: Initializes and prints a string to an 16x2 monochrome LCD.
- `fall-det`: Implements the fall-detection algorithm in [144] and detects falls in persons using an ADXL345 accelerometer, e.g., when the accelerometer is worn by a person. Figure 7.1 shows images of the accelerometer (connected to a Raspberry Pi) being dropped from a height in our drop test, while Figure 7.2 shows screenshots of the output from the `fall-det` program running on the Raspberry Pi.

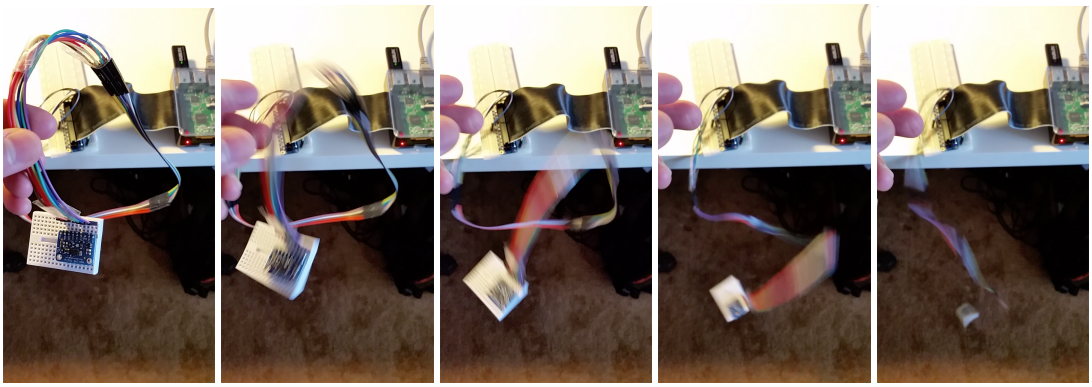


Figure 7.1. Snapshots of accelerometer being dropped from a height to simulate a fall.

Figure 7.2. Screen output showing fall detection. "$F_i \rightarrow F_j$" indicate state-transitions in the state-machine algorithm for fall detection in [144]. A "Critical Fall" is reported when there is no movement for a certain amount of time after a fall is first detected and the "Alert Fall" state is triggered.

Then, we measured the times taken for PCFIRE-C in our enforcement approach to prescribe safety-checks for each test program, and we measured the times taken for AUSPICE in our logic approach to automatically prove the safety of each test program. We did not measure the run-time overheads of our safety-checks on our hardware I/O test programs, as we focused on ensuring that each test program could perform their intended hardware I/O operations.

Experimental Setup

We wrote each test program in C, and we compiled each program using `gcc` 4.6.3 on Linux 3.18 on the ARM platform. All programs were compiled with debug symbols included, and all programs were statically linked.

Similarly to our experiments in §7.1, all safety-check prescriptions were generated on an Intel 2.6 GHz Core i7 (Quad-core) with 16 GB of RAM. All safety proofs were generated on an `r3.large` instance on the Amazon Elastic Compute Cloud (EC2), which has 2 virtual CPUs on the Intel Xeon E5-2670 v2 (Ivy Bridge) processor with 15.25 GB of RAM.

In addition, we ensured that each of our test programs performed their intended hardware I/O behavior by running each test program on a Raspberry Pi 1 Model B+ with a 700 MHz ARMv6 processor and 512 MB RAM with Linux 3.18, with the required hardware input and output devices connected via the Raspberry Pi's GPIO interface.

7.3.3 Results

Program	Source-code LOC (without safety -checks)	Source-code LOC (with safety -checks)	Machine-code bytes (without safety -checks)	Machine-code bytes (with safety -checks)	Size Increase	PCFIRE-C Prescription Time	AUSPICE Proof Time
`blink`	291	407	13546	13572	0.2%	1.3 s	1.07 h
`light`	312	428	13930	14920	7.1%	1.3 s	1.64 h
`lcd`	625	908	20764	25076	20.8%	4.4 s	37.5 h
`fall-det`	693	925	23021	31032	34.8%	5.2 s	71.8 h

Table 7.7. Safety-check prescription times, safety proof generation times, and source-code and machine-code size increases due to safety-checks for hardware I/O programs.

Next, we present and discuss the results of our evaluation for programs containing hardware I/O behavior. Table 7.7 summarizes our evaluation results for the sizes of the source-code and machine-code of our test programs before and after safety-checks were added, and for the times taken by PCFIRE-C to prescribe safety-checks, and for AUSPICE to generate safety proofs.

First, the added safety-checks introduced between approximately 100 (for `blink` and `light`) to 300 (for `lcd` and `fall-det`) lines of C code. The compiled test programs containing safety-checks had binaries that were between 0.2 % and 7.1% larger (for `blink` and `light`) to 20% to 35% larger (for `lcd` and `fall-det`).

Second, PCFIRE-C prescribed safety-checks for all four of our hardware I/O test programs in less than 5 seconds. AUSPICE generated the safety proofs for the `blink` and `lcd` test programs in under 2 hours, and this proof time is comparable to that of our file I/O examples in §7.4. On the other hand, the proof times for the `lcd` and `fall-det` test programs are significantly longer. This is because both programs are significantly larger, and have much deeper call-trees. For instance, for our file I/O test programs, the deepest call-tree path has a depth of 4, and for the `blink` and `light` examples, the deepest call-tree path has a depth of 3. In contrast, `lcd` and `fall-det` both have a maximum call-tree depth of 6, as they have more complex functionality. Since the run-time of AUSPICE's inter-procedural analysis is exponential in the depth of the call-tree, the proof times are significantly longer for `lcd` and `fall-det`.

We note that `lcd` and `fall-det` have 2229 and 3331 instructions respectively, and are significantly larger than the largest reported test-programs for which safety properties have been automatically proved using a foundational approach that considers the full semantics of the instructions (1104 instructions using ARMor [18], which uses the Cambridge ARM model [32]).

7.4 Proof Optimization

Next, we investigated the impact of our safety proof optimizations, as described in §6.2, on the times taken to generate safety proofs in AUSPICE. First, we investigate the effects of leveraging common compiler conventions to optimize the safety assertion analysis in AUSPICE (§6.2.1) on proof times. Second, we investigate the effects of weakening the context-sensitivity of AUSPICE's overall analysis (§6.2.2) on the number of analysis iterations needed.

7.4.1 Safety Assertion Analysis Optimizations

To quantify the impacts of the optimizations to AUSPICE's safety assertion analysis, we compared the safety proof times on a small number of our benchmark test programs (described in §7.1) using AUSPICE's original, unoptimized safety assertion analysis (§5.4), as compared to the optimized safety assertion analysis (§6.2.1).

Experimental Setup

For this experiment, We compared the safety proof times for 3 programs: `arrcpy`, `sort`, and `stringsearch`. For this experiment, all safety proofs were generated on an Intel Core i7 2.6 GHz

quad-core processor with 16 GB RAM (note that this is a different setup from our experiments in §7.1, which used a slower CPU).

Results

Program	Size (Bytes)	Unoptimized Proof Time	Optimized Proof Time	Optimized: X% faster
`arrcpy`	1714	16.4 m	6.5 m	252%
`sort`	1718	2 h	9.4 m	1297%
`stringsearch`	23592	6.05 h	0.76 hours	796%

Table 7.8. Comparing AUSPICE's safety proof times before and after optimization of the safety assertion analysis in AUSPICE.

Table 7.8 summarizes our results. AUSPICE's proof optimizations significantly improved the times taken for automated safety proofs for all of our test programs. With our optimized safety assertion analysis in AUSPICE, the safety proof times for our test programs improved by between 252% (for `arrcpy`) to 1297% (for `sort`).

7.4.2 Optimization of Inter-procedural Analysis

		Iterations of Analysis		
Program	Program Size (Bytes)	CSCS (Unoptimized)	SFCS (Optimized)	Optimization
`lcd`	25076	2799	751	73%
`fall-det`	31032	5069	845	83%

Table 7.9. Comparison of number of iterations of analysis of Call-site Context-Sensitivity (CSCS) vs. Single-Function Context-Sensitivity (SFCS) (§6.2.2).

Next, to show the optimization gains from AUSPICE's SFCS inter-procedural analysis (§6.2.2), we compared the number of iterations that the `SafeFunctionAnalysis` function (Algorithm 5.1) needs to be run in our optimized SFCS analysis to the number of iterations it would have taken in the unoptimized CSCS analysis. The savings in the number of analysis iterations is greatest in programs that make repeated calls to non-syscall-wrapper functions in the program.

We simulated the number of iterations the inter-procedural analysis needs to run for the `lcd` and `fall-det` test programs by analyzing their function-level call-trees. Table 7.9 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 and complex test programs such as `lcd` and `fall-det`, which have repeated calls to the same callee functions (e.g., helper functions) in each function.

7.5 Case Study: Prevention of Security Vulnerabilities

Next, we carried out a small-scale case study to apply our approach to CFI in this dissertation to a known security vulnerability in a piece of commodity software. The objective of this case study was to investigate if our approach can both: (i) provide CFI enforcement through our safety-check prescriptions, to prevent a known security vulnerability, and (ii) provide an automatically generated safety proof stating that the resulting program, after adding our safety-check prescriptions, now possesses the safety property of CFI, and hence will no longer be susceptible to the (now mitigated) security vulnerability.

In this case study, we make use of a buffer overflow vulnerability in the WU-FTPD daemon for FTP servers. This vulnerability was announced in CVE-1999-0878, and is part of a corpus of known security vulnerabilities for evaluating software static-analysis tools curated by Zitser et al [145].

We first describe the target software, the WU-FTPD daemon, and the details of the buffer overflow vulnerability in WU-FTPD (§7.5.1). Then, we describe the process and results of applying our CFI approach to the target software (§7.5.2). Finally, we conclude the case study by discussing the impacts of our prescribed CFI safety-checks on the behavior of the target program (§7.5.3).

7.5.1 Buffer Overflow Vulnerability in WU-FTPD

Zitser et al. [145, 146] assembled a corpus of buggy C code fragments containing actual buffer overflow vulnerabilities from three popular open-source applications: the BIND DNS server, the WU-FTPD FTP server, and the Sendmail mail transfer agent (MTA). Each buffer-overflow vulnerability in the corpus contained buggy C code fragments, a small driver program for invoking the vulnerable code fragment, as well as a "correct" code fragment that repaired the root-cause of the buffer overflow vulnerability. This corpus consisted of 15 buffer overflow vulnerabilities, and we selected one of these fragments of C code to apply our CFI approach to.

We selected the C code fragment for the "WU-FTPD mapped `chdir`" vulnerability from Zitser's corpus. This vulnerability was announced in the Common Vulnerabilities and Exposures (CVE) database as CVE-1999-0878, and is a remotely exploitable vulnerability in versions of WU-FTPD

older than or equal to 2.5. Listings A.1 and A.2 in §A.1 show the vulnerable fragments of C code in WU-FTPD, as curated by Zitser [146].

The "mapped `chdir`" vulnerability was caused by calls to the `strcpy()` and `strcat()` functions provided by the GNU `libc` standard library for C that copied tainted pathnames into a buffer. WU-FTPD failed to check that the supplied pathnames being copied could fit in their destination buffers before calling `strcpy()` and `strcat()`, thus making it possible for a buffer-overflow to occur. Specifically, there were three separate buffers that could be overflowed as a result of the unchecked calls to `strcpy()` and `strcat()`.

7.5.2 Applying our CFI Approach

To apply our CFI approach in this dissertation, we performed two steps: (i) we compiled the buggy code fragment with its driver code; (ii) we obtained prescriptions of safety-check code by running the PCFIRE-C tool on the resulting compiled program and the buggy source-code fragment, after which we inserted the prescribed safety-checks into the buggy source-code, and (iii) we recompiled the resulting code with the prescribed safety-check code inserted, and ran the AUSPICE tool on the compiled machine-code to try to obtain a safety proof of the CFI of the resulting code.

PCFIRE-C generated a total of 7 safety-check prescriptions in 0.82 seconds. AUSPICE took 2.46 hours to complete its safety proof generation on the resulting compiled program containing the prescribed safety-checks on an `r3.large` instance on the Amazon Elastic Compute Cloud (EC2), which has 2 virtual CPUs on the Intel Xeon E5-2670 v2 (Ivy Bridge) processor with 15.25 GB of RAM.

Thus, we demonstrated that our CFI approach can be applied to programs containing buffer overflow vulnerabilities to both: (i) insert safety-checks to prevent exploitable buffer overflow vulnerabilities, and (ii) automatically prove that the resulting program can no longer be exploited (by proving that the CFI safety property holds for the program).

7.5.3 Impacts on Program Behavior

Next, we discuss the impacts of the PCFIRE-C-prescribed safety-checks on the behavior of the buggy program in our case study. We first describe Zitser's proposed fixes to eliminate the exploitable buffer overflows in the "mapping `chdir`" vulnerability in WU-FTPD. Next, we describe our safety-checks as prescribed by PCFIRE-C, and compare them with Zitser's proposed fixes. Fi-

nally, we describe the run-time behavior in programs containing our prescribed safety-checks, and contrast this with the run-time behavior of Zitser's fixed version.

Fixes in Zitser's "Correct" Version

Listing A.3 in §A.2 lists Zitser's fixed version of the WU-FTPD code fragment that contains the "mapped `chdir`" buffer-overflow vulnerability.

At a high-level, the code fragment for the WU-FTPD vulnerability includes the `string.c` source file (which provides the string functions `memcpy`, `strchr`, `strcpy`, `strcat`, `strrchr` and `strlen`), and Zitser's proposed fixes do not modify any of these functions, presumably because they are provided by the GNU `libc` standard library.

Then, Zitser's fixes eliminate the buffer overflows in the code fragment in the following ways:

- The `mapping_getwd()` function is replaced with `mapping_getcwd()`; the new `mapping_getcwd()` function has an added size parameter. Then, the previous call to `strcpy()` is replaced with the length-guarded `strncpy()`, which uses this supplied size parameter, and the destination buffer which is written to is manually null-terminated in this new function.

- In the `do_elem()` function, a check is added to ensure that `strcat()` is called only if the length of the buffer to be concatenated on to the destination buffer will not exceed the length of the destination buffer; also, the length-guarded `strncat()` function is used instead of `strcat()` in another concatenation operation.

- In the `pwd()` function, a canary is used to detect if the path buffer has been overflowed [44], after a function whose buffer can potentially be overflowed (`getwd()` or `getcwd()`) is called. Then, if the buffer has been overflowed, a warning message is printed.

- In addition, the `pathspace` and `old_mapped_path` buffers are declared as global variables, so that if these buffers are overflowed, they cannot result in function return pointers saved to the stack being overwritten.

Thus, Zitser's fixes try to eliminate potential buffer overflows either by using length-guarded versions of string functions provided by the GNU `libc` C standard library, such as `strncpy()` and `strncat()`, or by checking the lengths of destination buffers before calling the length-unguarded versions of string functions such as `strcpy()` and `strcat()`.

Comparing Our Safety-Checks with Zitser's "Correct" Version

Next, we describe the safety-checks prescribed by PCFIRE-C. Listings A.4 and A.5 in §A.3 list the source-code fragments with our prescribed safety-checks added.

Unlike in Zitser's fixes, we do not assume that the GNU `libc` C standard library cannot be modified; instead, our approach would involve writing safe versions of the equivalent functions provided by the C standard library.

Our prescribed source-code safety-checks are added directly at the source-code statements where strings (i.e., arrays of `chars`) are written to. This includes the `memcpy`, `strcpy`, and `strcat` functions in Listing A.5. In addition, our prescribed source-code safety-checks are also added to the source-code statements in the `do_elem()` and `mapping_chdir()` functions in Listing A.4 where path buffers are written to.

The main difference between our safety-checks and Zitser's fixes is that our prescribed safety-checks address the root-causes of potentially dangerous memory writes, and that we allow safety-checks to be inserted in (modified) library functions such as `strcpy`, whereas Zitser's approach assumes that library functions will not be modified, and their fixes work around existing library functions.

In addition, in using the length-guarded versions of the GNU `libc`-provided string manipulation functions, such as `strncpy()` and `strncat()`, Zitser's fixes implicitly assume that the length argument supplied to the `mapping_getcwd()` function is correct, and that the buffer addresses supplied are valid `char` arrays.

However, as we illustrated in our example in §3.1.1, length-guarded versions of functions such as `strcpy()` can still be susceptible to buffer-overflow vulnerabilities when the wrong buffer length is supplied to such functions, or when buffer addresses supplied are incorrect or invalid. As a result, AUSPICE was unable to prove that the buggy code fragment which contains Zitser's fixes possessed the CFI safety property. On the other hand, the safety-checks prescribed in our approach that we added prevent all root-causes of buffer-overflows that may lead to CFI violations, and this is borne out by the CFI proofs generated successfully by AUSPICE.

Comparing Run-time Behavior During Buffer Overflow

The safety-checks prescribed by the PCFIRE-C tool in our approach, when added to the target program as-is, have the following features: (i) they prevent potential CFI violations from occurring,

and (ii) they do not specify any recovery action. As a result, the prescribed safety-checks will cause potentially dangerous memory writes (e.g., to arrays) to not be executed when they are not safe, resulting in the memory writes being silently ignored.

Concretely, the modified string-manipulating functions such as `memcpy()`, `strcpy()`, and `strcat()` that have our prescribed safety-checks added to them will silently stop copying or concatenating characters to their destination buffers if a memory write may result in a buffer overflow that can lead to a CFI violation.

On the other hand, if an invalid destination buffer or an invalid buffer length is passed to the string-manipulating functions in Zitser's fixed version of the target program, a buffer overflow can still occur. If the destination buffer and buffer length provided to functions such as `mapping_getcwd()` are valid in Zitser's fixed version of the target program, but a buffer overflow can still occur (e.g., destination buffer too small), the behavior of Zitser's fixed version will be determined by the behavior of the `strncat()` and `strncpy()` functions, i.e., copying stops when the destination buffer is full.

The advantage of Zitser's approach to fixing potential buffer overflows by using the length-guarded variants of the GNU `libc` string functions (e.g., `strncpy()`, `strncat()`) is that the semantics of these functions are well-understood. However, these functions are still not foolproof for preventing CFI violations, as compared to our approach. A more robust way of handling potential CFI violations is to use our approach for CFI enforcement, and combine it with sensible error handling behavior, e.g., returning a special error code from the `else` clause of our prescribed safety-checks. However, such error handling behavior is application-specific, and as such, it is outside of the scope of this dissertation.

7.6 Discussion: Supported Programs

In this chapter, we have demonstrated the range of programming language features in C that our approach to CFI is amenable to. For instance, our benchmark programs have demonstrated that our approach can support programs with function calls, loops, nested loops, arrays, and bit operations. Our file I/O programs have demonstrated that our approach can support programs that use the `open`, `close`, `read`, and `write` syscalls to read and write files. Our hardware I/O programs have demonstrated that our approach can support programs that additionally use the `nanosleep`, `mmap` and `munmap` syscalls to read and write from and to hardware devices using memory-mapped I/O.

Through our implementations of our file I/O and hardware I/O test programs, we have demonstrated that it is possible to write programs with realistic and useful features that are still amenable to our approach for prescribing CFI safety-checks, and proving CFI.

However, were unable to directly make use of the C standard library due to its large size, and due to our requirement that all our target programs must be statically compiled. In addition, we had to make some changes to the WiringPi [143] library that simplifies hardware I/O programming on the Raspberry Pi. These suggest that our approach is not amenable to being directly applied to existing programs without any changes. In addition, our approach suffers from a number of limitations (as discussed in §8.1), such as not supporting writes to variables in the stacks of caller functions, and supporting safety proofs only for syscalls that write to concretely-specified buffers whose sizes are statically known at compile-time. Such behavior is likely to be found in existing programs.

Nonetheless, as we demonstrated in our experiments, it is still possible to write programs that perform useful functionality even with these limitations. Moving forward, we envision that our approach can be used to develop new software for which CFI is desired, with high-assurance that the CFI holds.

7.7 Summary

In this chapter, we described various experiments that we performed to evaluate the ability of our approach to CFI in this dissertation to be applied to various kinds of target programs. We described the methodology and setup of our experiments, and we presented and discussed the results of our experiments. We evaluated our CFI approach on a different types of target programs to illustrate the variety of programs that can be supported by our approach.

First, we showed that our approach to CFI can support programs containing a wide variety of C constructs (§7.1). We demonstrated that our CFI approach was able to prescribe safety-checks and automatically prove the safety for 23 different benchmark programs drawn from the MiBench suite of realistic embedded software benchmarks [33], the WCET benchmarks [34], the Bionic C library for Anrdoid [141], and for 3 of our self-authored test programs. We also showed that the times taken to prescribe CFI safety-checks (less than 8 minutes for all test programs) and automatically prove safety in our approach (less than 19 hours for all but one test program) were feasible.

Second, we showed that our approach to CFI can support programs that contain simple file I/O behaviors (§7.2). We described a methodology for developing C programs containing file I/O

behaviors that are also amenable to the automated safety proof process in the logic approach of this dissertation. Then, we implemented simple versions of 3 common Unix-based text utilities (`cat`, `wc`, and `grep`), and showed that our approach was able to prescribe safety-checks and automatically prove the safety of these text utilities that contained file I/O behaviors. Our safety-check prescription took less than 1 second for our 3 utilities, and our safety-check proofs took less than 2.15 hours for each of our 3 utilities. In addition, we also showed that there is an inverse relationship and a trade-off between run-time overheads and proof times for varying input buffer sizes for our test programs.

Third, we showed that our approach to CFI can support programs that contain hardware I/O behaviors (§7.3). We described how our methodology for developing C programs containing file I/O behaviors that are amenable to our automated safety proofs can be extended for developing C programs containing hardware I/O behaviors. We then developed 4 test programs for the Raspberry Pi single-board-computer containing hardware inputs such as light sensors and accelerometers, and hardware outputs such as LEDs and LCD screens. We evaluated our approach to CFI on these 4 programs, and we showed that we could prescribe safety-checks and automatically prove the safety of these programs containing hardware I/O behavior. Our safety-check prescription took less than 6 seconds, and our automated safety proofs took less than 2 hours for 2 of our test programs, and 38 and 72 hours respectively for our more complex test programs that performed LCD outputs and implemented a fall detector using an accelerometer respectively.

Fourth, we demonstrated the effectiveness of the safety proof optimizations (§7.4) described in §6.2, and showed that our optimizations improved the safety assertion analysis in AUSPICE by between 250% and 1290%, and improved the inter-procedural analysis for large programs (such as our LCD and fall-detector examples for hardware I/O) by between 70% and 83%.

Finally, we performed a small-scale case study on applying our approach to CFI in this dissertation to known buffer overflow vulnerabilities in commodity software (§7.5). We showed that we were able to automatically prescribe safety-checks for a piece of C code with a known, exploitable buffer overflow vulnerability in the WU-FTPD daemon, and that we could automatically prove the safety of the resulting program. We also qualitatively compared the proposed fixes for the buffer overflow vulnerabilities by Zitser et al. [146] to our prescribed safety-checks, and we argued that our approach to CFI provides a more comprehensive "clean-slate" approach to ensuring CFI.

Chapter 8

Discussion

This chapter discusses a number of issues concerning the approach to Control-Flow Integrity presented in this dissertation. First, we describe a number of behaviors in C programs that our approach is unable to support (§8.1). This is because of the nature of the automated safety proof generation in our logic approach. Second, we perform a qualitative security analysis of our approach (§8.2). We discuss the security measures that would need to be deployed alongside our approach to attain software security, as well as residual security threats that are possible in spite of our approach. Third, we discuss challenges in enforcing and proving CFI at different levels of abstraction, at the source-code and machine-code levels respectively (§8.3).

8.1 Limitations to Program Behavior

8.1.1 Writable Memory Regions

Our approach to CFI in this dissertation defined 3 safety properties (§3.1) such that when they hold for a program, the CFI of the program is ensured. We defined these 3 safety properties in a way that they enable our logic approach in this dissertation to automatically generate safety proofs for programs. These safety proofs in turn imply that the CFI of the programs is ensured.

While our safety properties are sufficient to ensure CFI, they are stricter than necessary to ensure the CFI of a program. This is to enable our prescribed run-time safety-checks to be simple, and to enable our safety proofs to be automatically generated. To ensure CFI, we do not need to restrict memory writes in each function to addresses that are smaller than the current function's frame pointer value (Property 2 in §3.1.2); we only need to ensure that the memory addresses in callers' stacks where callee-saved register values are stored are not written to. However, this would

require complex stack analyses, which may not be feasible at run-time, and it would be difficult to automatically prove that such complex stack analyses will indeed ensure safety (and CFI).

Hence, as a result of our strict safety properties to ensure that our logic approach can automatically prove CFI, the memory regions that instructions in a given function are allowed to write to is restricted.

Restrictions on Writable Memory Regions

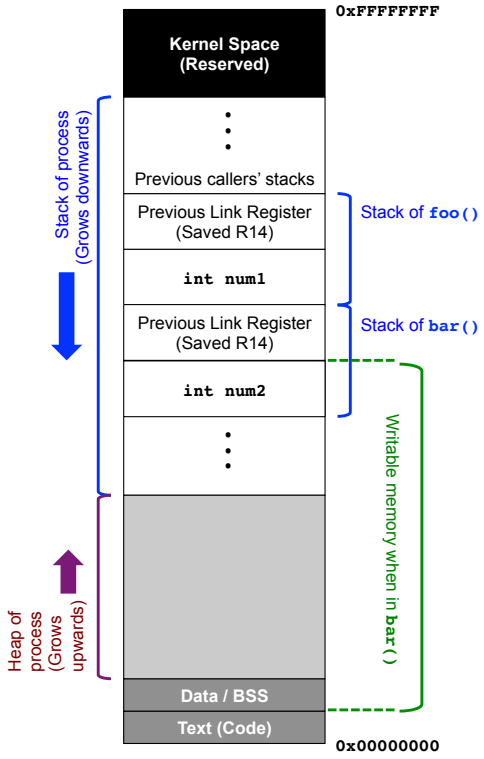


Figure 8.1. Writable memory for instructions in a given function, `bar()`.

Figure 8.1 shows the memory layout for a user-mode process in Linux, and illustrates the memory areas that an instruction in the function `bar()` is allowed to write to. To ensure that Safety Property 2 (§3.1.2) in our enforcement approach holds, it must not be possible for instructions in the function `bar()` to overwrite any callee-saved registers for `bar()` nor for any function in the call-stack that ran before `bar()`. Then, to enable this property to be automatically provable in our logic approach, we ensure (in our safety-checks prescribed by PCFIRE-C, §3) that instructions in `bar()` must not be able to write to *any* memory larger than the frame pointer of `bar()`.

Then, from Figure 8.1, the stacks of all callers of `bar()` appear at addresses larger than the frame pointer of `bar()`. As a result, all local variables of caller functions of `bar()` will be located

at memory addresses that are larger than the frame pointer of `bar()`. For instance, in Figure 8.1, the local variable `num1` in the function `foo()`, which is a caller of `bar()`, will not be (provably) safely writable by an instruction in `bar()`. Hence, even if `foo()` passes `bar()` a pointer to `foo()`'s local variable (e.g., as a way of receiving a return value), a safety-check prescribed by our PCFIRE-C tool will prevent any instruction in `bar()` from modifying any memory in its callers' stacks. Then, any attempt to write to a caller function's local variables will result in a "silent" failure of the attempted memory write, as the prescribed safety-check will prevent it.

Passing Data by Reference Between Functions

Typically, arguments are passed-by-reference from a caller function to its callee function when a programmer wants to be able to access the changes made to the arguments after the callee function's execution. Arguments are typically passed-by-reference from a caller to its callee using a pointer to a memory location containing the argument. Typically, caller functions pass arguments by reference using a pointer to a local variable stored on the caller function's own stack. However, in our approach to CFI, callee functions will not be able to modify the memory pointed to by such a pointer to a local variable of a caller function.

Thus, to pass arguments by reference, a caller function must use memory that is writable by both itself, as well as by its callee function. In Figure 8.1, we can see that the data section of the process's memory, where global variables are stored, and the heap of the process, will remain writable to all functions regardless of their position in the call stack. On the other hand, each function cannot modify the memory that contains the stacks of its callers. Hence, to pass arguments by reference, functions must use pointers to either the heap, or to global variables in the data section.

In our approach, we do not make use of the standard C library, as it introduces extraneous instructions and behaviors that are not supported by our logic approach (§7.2.1). As a result, we currently do not have a heap allocator (i.e., an implementation of the `malloc` heap memory allocator). Thus, functions must use global variables to pass data by reference between functions. Note that this applies only if callee functions must be able to modify the memory (pointed to by a pointer from a caller) passed to it by its caller; otherwise, callee functions will still be able to read from memory locations where its callers' stacks (and hence its callers' local variables) are located.

It is also possible to write a heap allocator that contains our prescribed safety-checks, so that the heap allocator can be proved safe by our logic approach, and statically linked to a program being developed. Caller functions can then allocate variables on the heap, and pass pointers to these

variables to their callee functions. We intend to explore the development of a heap allocator that can be proved safe in our logic approach in future.

8.1.2 Return of `struct`s From Functions

The logic approach in this dissertation does not support generating safety proofs for programs that contain functions that have a `struct` return type. This is because commodity compilers (e.g., `gcc`) compile C functions that return a `struct` to machine-code whose behavior cannot be automatically proved safe by our logic approach.

Unsafe Behavior in `struct`-returns

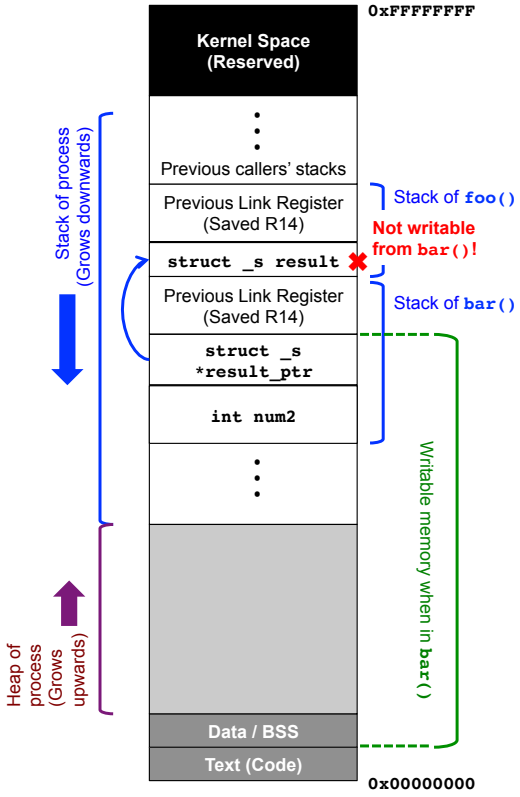


Figure 8.2. Typical `struct` return behavior from a function `bar()`.

To support `struct`-returns from a function, in general, the compiler passes an address in register `r0` to the callee function, at which the callee is expected to return the `struct`. The compiler then stores the return address on the stack of the callee function. Figure 8.2 illustrates the stacks of two functions `foo()` and `bar()`, where `foo()` calls `bar()`, and `bar()` returns a `struct`. Then, the compiler automatically allocates memory on `foo()`'s stack to receive the returned `struct`, and

passes the address of this memory in register `r0`. Next, the compiler copies this address from the register `r0` to `bar()`'s stack, as shown at the location marked "`struct _s *result_ptr`". Then, for `bar()` to return a `struct` to its caller `foo()`, the compiler copies the `struct` to be returned to the address pointed to at the location marked "`struct _s *result_ptr`" in Figure 8.2.

However, as discussed earlier, our approach to CFI does not allow instructions in a function to write to memory addresses in the stack frames of functions that are earlier in the call stack (i.e., its caller and their callers). However, the address passed from a caller to a callee for the callee to write the saved `struct` to will be on the caller's stack. Then, by the ATPCS [28], the callee will necessarily advance its own frame pointer's address past the caller's stack, so that the callee does not interfere with the caller's stack. This then renders the return address passed from the caller to the callee unwritable, based on the safety requirement as defined in our approach (§3.1), as the temporary address will be at an address that is larger than the callee-function's stack frame-pointer address.

Supporting Passing of `struct`s Between Functions

Hence, for two functions A and B, such that A calls B, for B to pass a `struct` to its caller A, function A should: (i) allocate a `struct` in a region of memory that can be provably safely written to by the function B, and (ii) pass a pointer to this `struct` to its callee function B. For the `struct` to be provably-safely writable by B, it must be at an address that is either larger than the base of B's stack, or smaller than the top of B's stack.

As discussed earlier in §8.1.1, for a caller function to pass a pointer to a callee function, e.g., for a caller to receive a `struct` from its callee, the caller function needs to pass a pointer to a global variable to its callee function. Then, without any compiler modifications (as is a goal of this dissertation, §1.3), programmers cannot make use of a C `return` statement to return its `struct`. Instead, programmers must manually "return" the `struct` back to its caller by copying the `struct` to the address of the global variable supplied to it by its caller. In future, we also envision that with a heap allocator that can be proved safe by our logic approach, functions can also pass pointers to heap-allocated variables to their callees to receive "returned" `structs`.

8.1.3 Hoisting of Safety-Checks

The proof automation in our logic approach makes use of local reasoning (§4.2.3) to avoid requiring loop invariants to summarize the behavior of loops. If a potentially-CFI-violating expression

contains expressions that change in a loop, then moving a CFI safety-check outside of a loop will require the behavior of the loop to be summarized, so that the analysis can reason about the entire loop at once. Then, summarizing the behavior of the loop will require loop invariants, whose automatic inference is an open problem in the general case.

Generating loop invariants automatically typically uses syntactic heuristics for loops written in various syntactically similar ways [147, 117], or estimations from dynamic analyses [148]. These approaches do not generalize well: for each type of loop pattern encountered, a separate approach for automating the CFI proof for the pattern will be required. Due to the lack of generalizable ways to handle CFI proofs for optimized safety-checks, we do not plan to address the automated proving of CFI safety for such optimized patterns of CFI remedial-hints in this dissertation.

In addition, in experiments with more balanced workloads (e.g., our file-based I/O utilities, §7.2), the run-time overheads of the inserted safety-checks are significantly lower than 220% even when they are inserted inside loops, suggesting that the performance overheads of the safety-checks in our enforcement approach are likely to be reasonable for realistic applications, even with safety-checks located inside of loops.

8.2 Security Analysis

Next, we analyze the impacts of our approach to CFI on the security of the target programs that we protect. First, we discuss the additional defenses that are needed, considering our threat model (§1.3), in order to achieve high-assurance of the security of our target programs (§8.2.1). Next, we analyze the list of 2011 CWE/SANS Top 25 Most Dangerous Software Errors [127], and qualitatively evaluate the software errors that our approach is able to mitigate (§8.2.2).

8.2.1 Threat Model and Assumed Defenses

In §1.3, we described the attacker model in our approach to CFI in this dissertation. Our attacker is an "external input attacker" who is only able to supply arbitrary external inputs to our target program, e.g., via files, network communication, and any other means of external communication. For this attacker model to be realized, a developer or IT engineer who is preparing and configuring a system to be deployed must ensure that security measures are in place to prevent attackers from:

- Compromising the hardware and firmware of the system.

- Compromising the OS kernel of the system.
- Compromising any other software on the system, that may in turn allow privilege escalation attacks that allow the OS kernel to be compromised.

Complementary Defenses

We discuss some of the security measures and high-assurance components that software developers and IT engineers can deploy to build a system for which they can have high-assurance of its security. While we introduce some concepts and techniques that can be used, we do not strive to be exhaustive in our list.

First, the hardware and firmware of the platform that a target program runs on must be able to function correctly, and be free of backdoors and implementation bugs (e.g., incorrectly-behaving instructions in processors). Many techniques have been developed for verifying various properties about hardware circuits (e.g., functional correctness, absence of deadlock, handshake protocols) [149]. In addition, recent work has formally verified important properties about various aspects of instruction set architectures (ISAs), such as device driver behavior [103] and instruction isolation for the ARMv7 architecture [102]. Techniques have also been developed to detect insider attacks that may circumvent microprocessor functionality [150, 151].

Second, the OS kernel must function correctly and be free of security bugs that attackers might otherwise exploit. Recent work has addressed various aspects of formally verifying that OS kernels are correct. For instance, the C implementation of the seL4 microkernel has been formally verified to be faithful to its high-level specification [113], and its subsequent compilation to machine-code for the ARM platform has also been automatically verified to be correct [152]. The Verve OS kernel and run-time system has been automatically verified to have type- and memory-safety [35] using a combination of Typed Assembly Language (TAL) [80, 81, 82] and a Hoare-style verifier. The CertiKOS project [96, 97] has built an OS kernel using a clean-slate design that has been proven to have important security properties. Dam et al. verified information-flow security for the PROSPER separation kernel [101]. In addition, some CFI techniques such as KCoFI [56] have focused on hardening OS kernels against CFI attacks.

These projects and techniques point to many possible candidates for deploying secure and correct OS kernels. Such OS kernels with strong guarantees of correctness and security provide the assurance that: (i) attackers cannot compromise the OS kernel to modify the target program and

its memory in user-space, and that (ii) the assumptions in our approach that the OS kernel correctly performs context-switching (e.g., correctly saving and restoring registers) during system call invocations do indeed hold.

Third, although our target programs are statically compiled and linked (§1.3), and hence do not have any direct dependencies on external libraries that need to be secure, they may rely on additional auxiliary software, such as device drivers and networking protocol stacks that reside and execute in the OS kernel. In addition, there may be user-level network protocols that programmers may link against for use in their programs, such as SSL. Techniques have been developed to verify safety properties of device drivers, such as in the Microsoft SLAM project, which produced the Static Driver Verifier (SDV) [153, 154]. Techniques have also been developed to verify the correctness of implementations of important network security protocols such as SSL [155].

Thus, while our approach to CFI can ensure that user-mode programs are secure against attacks from external input attackers who attempt to circumvent the CFI of programs, additional complementary security techniques are necessary to provide *defense-in-depth*, and we have provided a brief (and non-exhaustive) list of some such complementary techniques.

Residual Threats

While our approach to CFI protects our target programs from control-flow hijacking attacks, the main residual threat (apart from those that can be defended against using the complementary defenses we listed above) that our approach does not protect against is data attacks.

Our approach to CFI prevents control-flow sensitive areas of memory from being written by store instructions on the ARM architecture. However, attacks can still overwrite areas of memory that are not control-flow sensitive. These include data items in memory that are not control-flow sensitive. While attackers are not able to overflow buffers into control-flow sensitive regions of memory in our approach (e.g., overwrite the stack of a caller function), attackers can still overflow buffers into non-control-flow sensitive regions of memory.

For instance, there may be a buffer that is allocated as a global variable in the data section of the program's memory, that is adjacent to another variable in the data section. Attackers can still overflow such a buffer to overwrite adjacent variables. Likewise, attackers can also overflow heap buffers (supposing there is a heap allocator that our logic approach is able prove is safe) to overwrite adjacent variables. Attackers can then overwrite data items that may have an effect on the logic of the application, and thus indirectly modify the control-flow of the program. Such attacks

do not violate the safety property of CFI, but still constitute an attack as the logic of the program is circumvented.

8.2.2 Mitigated Attacks

Next, we review the CWE/SANS Top 25 Most Dangerous Software Errors [127], and qualitatively assess which of these errors can be mitigated by the approach to CFI in this dissertation.

ID	Software Error	Mitigated?
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Not Applicable
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Not Applicable
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Some errors directly mitigated
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Not Applicable
CWE-306	Missing Authentication for Critical Function	Not Applicable
CWE-862	Missing Authorization	Not Applicable
CWE-798	Use of Hard-coded Credentials	Not Applicable
CWE-311	Missing Encryption of Sensitive Data	Not Applicable
CWE-434	Unrestricted Upload of File with Dangerous Type	Not Applicable
CWE-807	Reliance on Untrusted Inputs in a Security Decision	Not Applicable
CWE-250	Execution with Unnecessary Privileges	Not Applicable
CWE-352	Cross-Site Request Forgery (CSRF)	Not Applicable
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Not Applicable
CWE-494	Download of Code Without Integrity Check	Not Applicable
CWE-863	Incorrect Authorization	Not Applicable
CWE-829	Inclusion of Functionality from Untrusted Control Sphere	Not Applicable
CWE-732	Incorrect Permission Assignment for Critical Resource	Not Applicable
CWE-676	Use of Potentially Dangerous Function	Some impacts mitigated
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	Not Applicable
CWE-131	Incorrect Calculation of Buffer Size	Some impacts mitigated
CWE-307	Improper Restriction of Excessive Authentication Attempts	Not Applicable
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')	Not Applicable
CWE-134	Uncontrolled Format String	Some impacts mitigated
CWE-190	Integer Overflow or Wraparound	Some impacts mitigated
CWE-759	Use of a One-Way Hash without a Salt	Not Applicable

Table 8.1. Ranked list of the CWE/SANS Top 25 Most Dangerous Software Errors, and a qualitative evaluation of which errors can be mitigated by the approach to CFI in this dissertation. CWE IDs and Software Error descriptions are from [127].

In Table 8.1, we evaluate which of the Top 25 errors our approach to CFI can mitigate. First, we note that a majority of the Top 25 errors are not related to CFI, which is a property about the execution of programs written in low-level languages such as C. For instance, many of the errors are related to SQL and OS command executions (CWE-89, 78), web applications running over HTTP (CWE-79, 434, 352, 494, 601, 829), authentication and cryptographic mechanisms (CWE-306, 862, 798, 311, 863, 732, 327, 307, 759), and interactions with the OS (CWE-807, 250, 22).

Second, our approach to CFI directly mitigates some buffer overflows (CWE-120) when they lead to control-flow hijacks, which SANS ranks as its third most dangerous software error. Other related software errors that our approach mitigates are those that can subsequently lead to buffer overflows or control-flow hijacks, namely, incorrect buffer size calculation (CWE-131), uncontrolled format strings (leading to format string attacks via use of the "`%n`" format specifier in C) (CWE-134), and integer over-/under-flows leading to wraparounds that can sometimes lead to buffer overflows (CWE-190). We say that some of the impacts of these software errors are mitigated (for CWE-131, 134, 190), as our approach prevents any potential buffer overflows that can arise from these errors, but does not mitigate any other effects, such as logic errors, that may result from errors such as incorrect buffer sizes.

While it is straightforward that our approach can directly mitigate buffer overflows (when they lead to control-flow hijacks), our approach can partially mitigate the subsequent impacts of some of the other software errors.

Incorrect buffer size calculations and integer over-/under-flows can lead to buffer overflows when the incorrect calculations involve pointer arithmetic or calculation of buffer or array indices. Then, when the safety-checks that are prescribed by our approach are in place, any statement that would result in a buffer overflow at run-time would be stopped from executing by the safety-check. Likewise, the execution of the print code that handles the "`%n`" format specifier will also be surrounded by a safety-check in our approach, and if an input is supplied that would result in a dangerous buffer overflow, the print code will be prevented from executing by the safety-check.

In our approach, for dangerous string functions such as `strcpy` (CWE-676), our enforcement approach will prescribe safety-checks for the statements that perform the actual dangerous string operations. Thus, our approach will be able to mitigate the dangerous behavior in these dangerous functions, although it would require users to implement their own safe versions of string functions provided by standard C libraries such as GNU `libc`, as we discussed in §7.2.1. We say that some of the impacts of this software error are mitigated by our approach, as there may be other dangerous

functions that result in flaws other than buffer overflows resulting in CFI violations.

Hence, our approach to CFI is able to directly mitigate the 3rd most dangerous software error in the CWE/SANS Top 25 Most Dangerous Software Errors, and it is also able to partially mitigate some of the impacts (that result in buffer overflows leading to control-flow hijacks) of 4 other errors on the list.

8.3 Machine-code vs. Source-code Views of CFI

Next, we discuss the challenges faced in providing CFI enforcement mechanisms at the source-code level of abstraction using source-code safety-checks (as prescribed by PCFIRE-C, §3.2), and proving CFI at the machine-code level of abstraction using our AUSPICE framework (§4).

The main challenge faced in the discrepancy between the levels of abstraction where we enforce CFI, as compared to where we prove CFI, is that we need to maintain a correspondence between the source-code level artifact at which CFI is being enforced, and the machine-code level at which CFI is being proved.

At the source-code level in C, potential CFI violations occur in suspect statements that make changes to memory (§3.2). More specifically, the safety-checks prescribed in our enforcement approach need to extract the memory address(es) that are written to by a suspect statement. These memory addresses that are written to are then subjected to a bounds checks in our prescribed safety-check. At the source-code level, these memory addresses manifest in multiple ways, e.g., as a pointer, as the address of a variable, as an array element, or as a member of a `struct`, amongst others.

Then, at the machine-code level, the memory addresses written to by the suspect instruction (which the suspect statement is compiled to) appears in two locations: (i) in the compiled machine-code of the PCFIRE-C-prescribed safety-check, and (ii) in the compiled machine-code of the suspect instruction. AUSPICE tries to automatically prove that the memory address bounds established by the safety-check apply to the memory addresses written to by the suspect instruction (§5.4). For this automatic proof to succeed, the memory address, as manifested at the machine-code level in (i) the safety-check, and (ii) the suspect instruction which writes to memory, must be identical. Otherwise, the bounds established by the safety-check cannot be used to reason about the memory addresses written to.

At the machine-code level, the operands of comparisons in our prescribed safety-checks, and the memory addresses written to in the suspect instructions, both manifest as values stored in a register, as ARM is a load/store architecture. In the $\mathscr{L}_{LR}$ program logic in AUSPICE, Hoare triples are constructed for basic blocks of instructions using the Compose rule in Hoare Logic (§4.2.2). Thus, the expressions of values in registers that AUSPICE reasons about are scoped to the basic block level. The basic block construction algorithm in AUSPICE (§5.1) is typical: we identify leaders as the targets of indirect jumps, and each basic block begins from a leader instruction and continues until but not including the next leader instruction.

Then, AUSPICE reasons about the effects of basic blocks "locally" (§4.2.3), i.e., it does not accumulate the effects of instructions on machine resources such as registers across multiple basic blocks, and reasons only about adjacent basic blocks. As a result, for AUSPICE's automatic safety assertion discharge to succeed, safety-checks need to be in a basic block adjacent to the suspect instruction.

For suspect statements whose memory expressions may be complex, PCFIRE-C's construction of safety-check prescriptions (§3.2) tries to ensure that its prescribed safety-checks are in an adjacent basic block. PCFIRE-C does this by suggesting to programmers to first pre-compute the memory address to write to using a pointer variable, and then writing directly to the pointer using the dereferencing ("`*`") operator in the suspect statement. This ensures that after the safety-check, there are no intervening basic blocks used to compute the memory address to write to, as the pre-computed address is already available in a temporary register in the compiled machine-code.

8.4 Summary

In this chapter, we described some of the limitations to program behavior (§8.1) that are imposed by our approach to CFI in this dissertation. These are: (i) instructions in a function are allowed to write to only addresses smaller than their current frame pointer address, which implies that functions can pass data to their callers only through global variables; (ii) functions are not allowed to return `structs` to their callers; and (iii) our prescribed safety-checks cannot be hoisted out of loops as our logic approach would not be able to support automated proof generation without manually-supplied loop invariants from programmers.

We also qualitatively evaluated the security benefits of our approach (§8.2). We described some complementary security measures that would need to be deployed alongside our approach to secure

a target program given our threat model, and we discussed some of the residual threats that our approach does not protect our target programs against, such as data attacks. We also qualitatively discussed how our approach to CFI is able to directly mitigate the 3rd most dangerous software error on CWE/SANS' list of Top 25 Most Dangerous Software Errors [127], and is able to mitigate some of the dangerous subsequent effects of 4 other errors on the list.

Finally, we discussed challenges in prescribing safety-checks for CFI and proving CFI at different levels of abstraction, at the source-code and machine-code levels respectively (§8.3). We described how the machine-code representations of the memory addresses checked in our prescribed safety-checks must correspond to those of the memory addresses written to, i.e., register values, and how in PCFIRE-C we ensure this by using temporary staging pointer variables to ensure the memory addresses checked and written to have identical machine-code representations by using the same pointer variable in the safety-check and memory-write statements.

Chapter 9

Conclusions and Future Work

This dissertation introduces an approach to Control-Flow Integrity (CFI) for embedded software that meets the challenges posed by today's open, connected, and pervasive embedded software. Today's embedded software is open and connected, exposing them to external attackers: CFI protects the control-flow of embedded software from being hijacked and modified by potentially malicious inputs from such external attackers. Our approach to CFI consists of two parts: an enforcement approach, which prescribes mechanisms for programmers to include in their C programs to enforce CFI, and a logic approach, which automatically generates safety proofs of CFI for ARM machine-code programs containing CFI safety-checks prescribed by our enforcement approach.

Our approach to CFI provides enforcement mechanisms that prevent the root-causes of CFI, so that today's pervasive embedded software that is safety-critical will not be stopped when CFI violations are detected, and our approach also provides enforcement mechanisms that are visible to programmers, so that programmers can see how the enforcement mechanisms affect their software, and customize them appropriately for their applications. In our approach, we developed the PCFIRE-C tool to prescribe check-and-branch C statements as safety-checks around suspect C statements that may cause CFI violations. Then, we leave it to programmers to insert these safety-checks, so that they can account for any changes to the behavior of their software.

Our approach to CFI also enables CFI to be formally proved automatically without expert inputs. This enables CFI to be achieved in a high-assurance manner, and for as many programmers as possible, without requiring programmers to have expert knowledge about formal methods. In our approach, we developed the AUSPICE framework, which layers a local-reasoning safety rule on top of a trustworthy semantics and a Hoare Logic for ARM machine-code instructions. Then, the AUSPICE framework provides a proof automation algorithm consisting of our Selective Compo-

sition proof tactic to enable reasoning about CFI enforced using source-code-based safety-checks, and an abstract interpretation algorithm to automatically discharge safety-proof obligations. The AUSPICE framework also includes extensions to support reasoning about CFI in user-mode ARM machine-code programs that invoke system calls in an underlying operating system.

Using 26 benchmark programs from two embedded software benchmark suites as well as our own test programs, we demonstrated that our approach is able to prescribe CFI safety-checks as well as automatically prove the CFI of machine-code programs, for programs containing a variety of C constructs and features such as loops, nested loops, function calls, arrays, and bit operations. Using 3 of our simple implementations of common Unix-based file I/O utilities, we also demonstrated that our approach is able to prescribe CFI safety-checks and prove the CFI of machine-code programs containing file-based I/O behavior. In addition, we implemented 4 programs containing various hardware inputs and outputs such as light sensors, accelerometers, LEDs, and an LCD, and including an implementation of a human fall-detection algorithm, for the Raspberry Pi single-board-computer. Then, we demonstrated that our approach is able to prescribe CFI safety-checks and prove the CFI of these programs containing hardware inputs and outputs. Finally, we demonstrated in a simple case-study that we were able to provably mitigate a known security vulnerability in the WU-FTPD FTP server, although we required safety-checks to be inserted in string functions typically provided by a standard C library. This suggests that our approach to CFI requires programmers to take a clean-slate approach to software development to ensure that any library function used by a target program must also be made safe for its CFI to be provable at the machine-code level.

We showed that the times taken for our approach to prescribe CFI safety-checks was short, and was less than 1 minute in most cases, and less than 8 minutes in all cases. We also showed that the times taken for our approach to prove CFI automatically was under 2 hours for most programs, and for a small number of large programs (more than 400 lines of C code, or more than 20 kB of machine-code), took between 18 and 72 hours, depending on the complexity of the target program.

The approach to CFI presented in this dissertation addresses the challenges faced in providing CFI for open, connected, and pervasive embedded software in the following ways:

CFI violations are currently prevented after-the-fact. Our enforcement approach prescribes safety-checks made up of check-and-branch statements that target the root-causes of CFI violations, and prevent these violations from occurring. These safety-checks carry out bounds checks on memory addresses before they are written to. Then, if writing to these memory addresses results in a CFI violation, the memory-write statements are not executed. This prevents the root-causes of

CFI violations, as compared to previous CFI techniques that detect only the end-results of a CFI violation, possibly long after the violation occurred.

Recovery actions after CFI is violated are limited. Previous CFI techniques are limited in their recovery actions as they detect CFI violations after-the-fact. Hence, previous techniques are limited to stopping programs in which CFI violations have been detected, as a CFI violation cannot be undone after it has occurred, without knowing the CFI-sensitive information that was overwritten in the attack. In our enforcement approach, since our prescribed safety-checks prevent the root-causes of CFI violations, we are able to allow programmers to perform different recovery actions. While we do not propose recovery actions in this dissertation, alternative recovery actions can be taken in the `else`-branch of our check-and-branch safety-check statements. Examples of recovery actions include ignoring the suspect statement that was prevented, and continuing operation, and signaling to other parts of the software that an error has been prevented, allowing appropriate actions to be taken at other points in the software.

CFI techniques are not transparent to the software development process. Previous CFI techniques required modified tools in the software compilation process: they required binary rewriters that modified program binaries after they were compiled, or modified compilers to add different behaviors such as shadow stacks. In our approach, our PCFIRE-C tool for prescribing safety-checks provides programmers with prescribed safety-checks that they can insert in their source-code prior to compilation. Then, programmers compile their programs with a standard, unmodified commodity compiler (we used `gcc`-4.6 in our evaluation), after which our AUSPICE framework proves CFI automatically for the resulting compiled machine-code program with the inserted safety-checks. Our approach to CFI provides safety-check prescriptions for the source-code of programs, before the software compilation begins, and our approach proves CFI for the machine-code of programs after the compilation process. Thus, our approach is transparent to the software development process by intervening before and after software compilation, making it amenable to development projects that do not allow modifications to the software development process, such as in high-assurance software projects.

We have demonstrated that our PCFIRE-C tool can prescribe CFI safety-checks for our test programs at the C source-code level for a number of test programs from the MiBench [33] and WCET benchmark [34] suites, as well as for our own implementations of file-based I/O utilities and programs on the Raspberry Pi single-board-computer performing hardware I/O operations.

Support for automated proofs is contingent on after-the-fact CFI enforcement. Previous techniques for CFI that have provided automated proofs of CFI have relied on safety-checks that are automatically-inserted using a modified software development process (e.g., binary-rewriting), and their proof automation relies on the presence of such automatically-inserted checks. As a result, their proof automation does not support the possibility of proof failures. In our approach, as safety-checks are inserted by programmers, our logic approach needs to support proofs that may fail, without running forever. In our logic approach, AUSPICE is explicitly designed to handle proof failures in its abstract interpretation algorithm, and it does so by recording the paths of undischarged safety assertions through the Control-Flow Graph (CFG) of a program. Then, when a cycle is detected in the propagation path of an undischarged safety assertion, the proof process is halted, and a proof failure is declared.

In addition, in our logic approach, we developed the Selective Composition proof tactic in AUSPICE which enables safety proofs to succeed using safety-checks provided using source-code mechanisms, as compared to previous CFI techniques, which relied on binary-rewriting inserted machine-code safety-checks. The Selective Composition proof tactic eagerly propagates logic pre-conditions forward in the Control-Flow Graph to Hoare logic theorems of successor basic blocks. This makes the pre-conditions of compound safety-checks in machine-code statements, as compiled from our prescribed source-code safety-checks, available for proof discharge in our abstract interpretation algorithm in AUSPICE.

The automated generation of CFI safety proofs in our logic approach, by the AUSPICE framework that we developed, has been evaluated on 26 benchmark programs from the MiBench [33] and WCET benchmark [34] suites, and we have demonstrated that AUSPICE was able to automatically generate safety proofs for our test programs containing various C features and constructs.

Support for realistic features in embedded applications has not been demonstrated. Previous techniques for automated CFI proofs in machine-code programs have focused on machine-code programs that run "bare-metal" on a processor without an operating system. Thus, previous techniques have not supported user-mode programs containing system calls. The main challenge is that the Cambridge ARM model, which AUSPICE in our logic approach is based on, does not support reasoning about the user-mode effects of a system call invocation. In our approach, in AUSPICE, we support proofs for system call invocations in user-mode programs by axiomatizing the user-mode-visible effects of system call invocations, assuming that the operating system kernel correctly services system calls according to the specified behavior in its interface. We then modified the

proof automation algorithm in AUSPICE to support system calls by using a symbolic and opaque representation of system call effects at the intra-procedural level, and concretizing these effects during our inter-procedural analysis where the concrete arguments to each system call invocation are available from the call-site of the system call invocation.

Our safety-check prescription and safety proof automation for programs containing system call invocations has been demonstrated on programs implementing simple Unix file I/O utilities (`cat`, `wc`, `grep`), and on programs providing hardware inputs and outputs on the Raspberry Pi single-board-computer, including a human fall-detector. Our approach was able to prescribe safety-checks, and subsequently automatically prove the safety of these test programs containing various file and hardware I/O behaviors.

9.1 Open Questions and Future Work

The work presented in this dissertation introduces a new point in the design space of approaches to tackle the problem of ensuring Control-Flow Integrity (CFI) in software, that is amenable to today's open, connected and pervasive embedded software. Our approach to CFI prioritizes the customizability of recovery actions for CFI violations, and the automation of formal CFI proofs, over the run-time overheads of providing CFI enforcement.

As an initial exploration of this point in the design space of focusing on customizable recovery actions and automatically provable CFI, our work has focused on providing CFI for function return-pointers. As such, one open question is how our preventative and provable approach to CFI can be extended to provide CFI enforcement for explicit function pointers.

Second, our work has explored how individual instances of embedded software running on a single (possibly embedded) computer can be secured against potentially malicious external inputs by providing CFI enforcement. A second open question following our work is how the overall security of multiple communicating instances of embedded software can be compositionally built up from individual instances of embedded software that are each individually formally proven secure against CFI violations.

Third, in evaluating our approach to CFI in this dissertation, we constructed a number of C programs to evaluate and demonstrate the capabilities of our PCFIRE-C and AUSPICE tools. In writing these C programs that featured file and hardware I/O behavior and system call invocations, we wrote a number of functions typically implemented by a standard C library such as GNU `libc` that in-

corporated safety-checks prescribed by PCFIRE-C and could be automatically proved to be safe by AUSPICE. An open question arising from our evaluation exercise is whether additional engineering support can be provided to programmers in the form of a safe standard C library that incorporates PCFIRE-C-prescribed safety-checks, that can be automatically proved safe by AUSPICE.

9.1.1 Provable CFI for Explicit Function Pointers

Most of the previous techniques for CFI [15, 25, 55] have focused on protecting the CFI of indirect jumps made using explicit function pointers, while using shadow stacks to protect function return pointers. Then, for indirect jumps, these techniques provide CFI at varying levels of granularity by picking the number of equivalence classes of indirect jump targets that are tracked, and checking that indirect jumps are made only to targets that belong to the same equivalence class. Then, the Control-Flow Graph (CFG) of the program is computed to determine the equivalence classes of indirect jump sites and targets.

In our approach, in the absence of indirect jumps made through explicit function pointers, we were able to compute a precise inter-procedural CFG for our target programs. As a result, we were able to achieve a context-sensitive inter-procedural analysis in the AUSPICE framework in our approach. However, with the use of explicit function pointers, in the general case, each indirect jump site can be a jump to any function in the program. This would cause a state-space explosion for a precise inter-procedural analysis.

One possible approach for supporting provable CFI in the presence of explicit function pointers is to reformulate our currently precise formal definition of CFI (in §4.2) to admit a less precise definition that is context-insensitive at the inter-procedural level. However, this reduces the strength of the formal proofs of CFI generated.

Another possible approach is to constrain the CFG computed in the presence of indirect jumps made using explicit function pointers. Similarly to how PCFIRE-C prescribes source-code safety-checks in our current approach to constrain the behavior of dangerous memory-write expressions, we can also make use of source-code safety-checks at indirect jump sites to limit the possible values of explicit function pointers that an indirect jump site can jump to. This is because programmers are likely to know in advance which functions are plausible jump targets from an indirect jump site. This would then allow the logic approach to compute a more precise CFG, and minimize the state-space explosion caused by an unconstrained number of possible indirect jump targets. In addition,

our approach would also need an enumeration of all memory locations that contain explicit function pointers, to add safety assertions to assert that these locations will not be overwritten at run-time.

In addition, supporting explicit function pointers will allow us to provide support for machine-code compiled from C++ programs, which make use of explicit function pointers to implement virtual function calls. In the case of virtual function calls in C++, the compiler is also aware of all possible jump targets for each virtual function call, making it possible for us to use such compiler information to compute a precise CFG for our logic approach.

9.1.2 Provable CFI as a Building Block for Distributed Properties

In this dissertation, we have focused on protecting the execution of embedded software running on a single (embedded) computer from CFI violations. However, with the growth in connectivity and the openness of today's embedded software, embedded software is likely to communicate with each other. Thus, there is a need for secure distributed embedded software.

Most approaches for ensuring the security and reliability of distributed software has taken a top-down view, for instance, by designing secure and reliable protocols for communication among multiple participants in a distributed system. However, such approaches are reliant on the software on each individual node first being reliable. Otherwise, when security violations such as CFI violations occur, the execution of the software can be hijacked and its behavior circumvented, resulting in arbitrary Byzantine faults.

As our approach to CFI can ensure that the behavior of software running on individual embedded nodes in a distributed system cannot be circumvented by CFI violations, we can use this property to build up to system-level security for a distributed system in a bottom-up manner. We envision that security properties about distributed systems can be expressed in a bottom-up manner, beginning from secure individual nodes. Then, the proofs of CFI that are automatically generated by our approach can be used either directly in a larger system-level proof about the distributed system (i.e., in the same proof environment), or they can be used indirectly as evidence of the security of each individual node, and reasoning about the software separately at the distributed system level.

9.1.3 Provable CFI for Large-scale Software Projects

Safe Libraries

In developing our case-studies to evaluate our approach to CFI, we implemented safe versions of a number of functions typically implemented by a standard C library such as GNU `libc`. We believe that our approach to CFI, which uses source-code safety-checks, can be extended to most functions provided by a typical standard C library. As compared to other work that has developed safe C libraries, our approach will be able to produce safe C libraries for which safety proofs can be automatically generated.

One challenge in producing such a provably safe C library for which safety proofs can be generated automatically, is whether this safety can be achieved without changing the interfaces (i.e., function prototypes and signatures) of current functions provided by typical standard C libraries, and if these interfaces need to be changed, then how can we minimize such changes.

Modular Preventative CFI

Another challenge is that our approach to provable CFI is currently not modular: our logic approach requires all instructions that can possible be executed to be statically compiled and linked in the target program. Another open question is whether our logic approach can support proofs for modularly compiled programs, perhaps with a cooperating dynamic link-loader.

Some CFI techniques have tackled the challenges of separate compilation [53] and modularity [55], and an open question is whether these techniques can be adapted to support the preventative CFI that was presented in this dissertation. One challenge is that these CFI techniques have provided coarse-grained CFI, as compared to the CFI provided in our approach, which uses precise interprocedural CFGs (due to our current exclusion of explicit function pointers).

Appendices

Appendix A

Buffer Overflow Example

In this appendix, we present code listings for WU-FTPD's "mapping `chdir`" buffer-overflow vulnerability (CVE-1999-0878) in Zitser et al.'s corpus of vulnerable code fragments containing buffer-overflows, as described in our case study on applying the CFI approach in this dissertation in §7.5.

Listings A.1 and A.2 list the original source-code of the buggy C code fragment containing the buffer-overflow vulnerabilities in WU-FTPD's "mapping `chdir`" vulnerability.

Listing A.3 lists Zitser's fixed version of the buffer-overflow vulnerability in the buggy C code fragment in Listing A.1. We provide Zitser's fixed version for comparison with the safety-checks prescribed by our PCFIRE-C. Zitser's buffer-overflow fixes are inserted at Lines 80, 86, 93, 94, 128, 132, 181, 210, and 238.

Listings A.4 and A.5 list the source-code of the C code fragment after PCFIRE-C's prescribed safety-checks have been added to the source-code in Listing A.1. PCFIRE-C-prescribed safety-checks were inserted in lines 158, 208, 214, and 229 in Listing A.4, and in lines 31, 60, and 98 in Listing A.5.

A.1 Buggy Version

Listing A.1. `mapped-path-bad.c` file in Zitser's WU-FTPD "mapping `chdir`" bug.

```
1  /*
2  MIT Copyright Notice
3
4  Copyright 2003 M.I.T.
5
6  Permission is hereby granted, without written agreement or royalty fee, to use,
7  copy, modify, and distribute this software and its documentation for any
8  purpose, provided that the above copyright notice and the following three
9  paragraphs appear in all copies of this software.
10
11 IN NO EVENT SHALL M.I.T. BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL,
12 INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
13 AND ITS DOCUMENTATION, EVEN IF M.I.T. HAS BEEN ADVISED OF THE POSSIBILITY OF
```

```
14 SUCH DAMANGE.
15
16 M.I.T. SPECIFICALLY DISCLAIMS ANY WARRANTIES INCLUDING, BUT NOT LIMITED TO
17 THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
18 AND NON-INFRINGEMENT.
19
20 THE SOFTWARE IS PROVIDED ON AN "AS-IS" BASIS AND M.I.T. HAS NO OBLIGATION TO
21 PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
22
23 $Author: tleek $
24 $Date: 2004/01/05 17:27:52 $
25 $Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-bad.c,v 1.1.1.1 2004/01/05
       17:27:52 tleek Exp $
26 */
27 /*
28 WU-FTPD Copyright Notice
29
30
31   Copyright (c) 1999,2000 WU-FTPD Development Group.
32   All rights reserved.
33
34   Portions Copyright (c) 1980, 1985, 1988, 1989, 1990, 1991, 1993, 1994
35     The Regents of the University of California.
36   Portions Copyright (c) 1993, 1994 Washington University in Saint Louis.
37   Portions Copyright (c) 1996, 1998 Berkeley Software Design, Inc.
38   Portions Copyright (c) 1989 Massachusetts Institute of Technology.
39   Portions Copyright (c) 1998 Sendmail, Inc.
40   Portions Copyright (c) 1983, 1995, 1996, 1997 Eric P. Allman.
41   Portions Copyright (c) 1997 by Stan Barber.
42   Portions Copyright (c) 1997 by Kent Landfield.
43   Portions Copyright (c) 1991, 1992, 1993, 1994, 1995, 1996, 1997
44     Free Software Foundation, Inc.
45
46   Use and distribution of this software and its source code are governed
47   by the terms and conditions of the WU-FTPD Software License ("LICENSE").
48
49   If you did not receive a copy of the license, it may be obtained online
50   at http://www.wu-ftpd.org/license.html.
51
52
53 $Author: tleek $
54 $Date: 2004/01/05 17:27:52 $
55 $Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-bad.c,v 1.1.1.1 2004/01/05
       17:27:52 tleek Exp $
56 */
57 /*
58 <source>
59 */
60 //#define SYSCALL
61
62 #ifdef SYSCALL
63 #include <ctype.h>
64 #include <stdlib.h>
65 /*#include <pwd.h> */ /* Using custom made pwd() */
66 #include <string.h>
67 #include <unistd.h>
68 #include <sys/stat.h>
69 #include <fcntl.h>
70 #include <assert.h>
71 #include <stdio.h>
72 #else
73 #define NULL ((void *) 0)
74 #endif
75 #include "my-include.h"
76 #include <sys/types.h>
77
78 /* Dummy chdir function to silence syscalls
79  * pwd is localized, so chdir effectively does nothing except
80  * check that an actual directory was created by the Makefile
```

```
 */
 #ifndef SYSCALL
int chdir(const char *path) {
   return 0;
}
 #endif

#ifdef MAPPING_CHDIR
/* Keep track of the path the user has chdir'd into and respond with
 * that to pwd commands. This is to avoid having the absolue disk
 * path returned, which I want to avoid.
 */
char mapped_path[ MAXPATHLEN ] = "/";

char *
#ifdef __STDC__
mapping_getwd(char *path)
#else
    mapping_getwd( path )
    char *path;
#endif
{

#ifdef SYSCALL
  printf("Copying %d chars into buffer path[] whose size = %d\n", strlen(mapped_path) + 1,
      MAXPATHLEN + 1);
#endif

  /* BAD */
  strcpy( path, mapped_path ); /* copies mapped_path to path without doing a size check */
  return path;
}

/* Make these globals rather than local to mapping_chdir to avoid stack overflow */
char pathspace[ MAXPATHLEN ]; /* This buffer can get overflowed too */
char old_mapped_path[ MAXPATHLEN ];

void
#ifdef __STDC__
/* appends /dir to mapped_path if mapped_path != /, else appends simply dir */
do_elem(char *dir)
#else
    do_elem( dir )
    char *dir;
#endif
{
  /* . */
  if( dir[0] == '.' && dir[1] == '\0' ){
       /* ignore it */
       return;
     }

     /* .. */
     if( dir[0] == '.' && dir[1] == '.' && dir[2] == '\0' ){
            char *last;
            /* lop the last directory off the path */
            if (( last = strrchr( mapped_path, '/'))){
              /* If start of pathname leave the / */
              if( last == mapped_path )
                last++;
              *last = '\0';
            }
            return;
     }

     /* append the dir part with a / unless at root */
     if( !(mapped_path[0] == '/' && mapped_path[1] == '\0') )
       /* BAD */
       strcat( mapped_path, "/" ); /* no bounds checking is done */
```

```
149       /* We do not check to see if there is room in mapped_path for dir */
150       /* BAD */
151       strcat( mapped_path, dir ); /* no bounds checking is done */
152 }
153
154 int
155 #ifdef __STDC__
156 mapping_chdir(char *orig_path)
157 #else
158 mapping_chdir( orig_path )
159       char *orig_path;
160 #endif
161 {
162       int ret;
163       char *sl, *path;
164
165 #ifdef SYSCALL
166       printf("Entering mapping_chdir with orig_path = %s\n", orig_path);
167 #endif
168
169       strcpy( old_mapped_path, mapped_path ); /* old_mapped_path is initially / */
170       path = &pathspace[0];
171
172       /* BAD */
173       strcpy( path, orig_path ); /* suppose path = orig_path = /x/xx/xxx/xxxx/... */
174 #ifdef SYSCALL
175       printf("Copying orig_path to path....max strlen(path) = %d. strlen(path) = %d\n",
              MAXPATHLEN - 1, strlen(path));
176       if (strlen(path) >= MAXPATHLEN){
177         printf ("ALERT:pathspace[MAXPATHLEN] has been overflowed!\n");
178       }
179 #endif
180
181         /* set the start of the mapped_path to / */
182         if( path[0] == '/' ){
183           mapped_path[0] = '/';
184           mapped_path[1] = '\0';
185           path++;
186         }
187
188 #ifdef SYSCALL
189       printf("so far mapped_path = %s\n", mapped_path);
190 #endif
191
192       while( (sl = strchr( path, '/' )) ){
193              char *dir;
194              dir = path;
195              *sl = '\0';
196              path = sl + 1;
197              if( *dir )
198                do_elem( dir ); /* appends directory names to mapped_path */
199              if( *path == '\0' )
200                break;
201       }
202       if( *path )
203         {
204 #ifdef SYSCALL
205           printf("path = %s.. calling do_elem\n", path);
206 #endif
207           do_elem( path ); /* we're in root and path is of the form aaaaa ... mapped_path
                  becomes /aaaa.. */
208         }
209 #ifdef SYSCALL
210       printf("mapped_path = %s\n", mapped_path);
211 #endif
212 #ifdef SYSCALL
213       if (strlen(mapped_path) >= MAXPATHLEN){
214         printf("ALERT: mapped_path[MAXPATHLEN] has been overflowed!\n");
```

```
215       }
216 #endif
217
218
219       if( (ret = chdir( mapped_path )) < 0 ){ /* change to the specified path */
220 #ifdef SYSCALL
221         printf("couldn't chdir to %s !\n", mapped_path);
222 #endif
223         strcpy( mapped_path, old_mapped_path ); /* change mapped_path back to original, i.e
              root */
224 #ifdef SYSCALL
225         printf("mapped_path changed to %s\n", mapped_path);
226 #endif
227       }
228
229       return ret;
230 }
231
232
233 /* From now on use the mapping version */
234
235 #define getwd(d) mapping_getwd(d)
236 #define getcwd(d,u) mapping_getwd(d)
237
238 #endif /* MAPPING_CHDIR */
239
240 /* Define pwd */
241
242 void
243 #ifdef __STDC__
244 pwd(void)
245 #else
246 pwd()
247 #endif
248 {
249   int canary = 7; /* used to see if path[] gets overflowed */
250   char path[MAXPATHLEN + 1]; /* Path to return to client */
251
252 #ifndef MAPPING_CHDIR
253 #ifdef HAVE_GETCWD
254   extern char *getcwd();
255 #else
256 #ifdef __STDC__
257   extern char *getwd(char *);
258 #else
259   extern char *getwd();
260 #endif
261 #endif
262 #endif /* MAPPING_CHDIR */
263
264 #ifdef HAVE_GETCWD
265   if (getcwd(path,MAXPATHLEN) == (char *) NULL) /* mz: call to mapping_getwd might overflow
         path */
266 #else
267     if (getwd(path) == (char *) NULL) /* mz: call to mapping_getwd might overflow path buf
          */
268 #endif
269       {
270 #ifdef SYSCALL
271         printf("Couldn't get current directory!\n");
272 #endif
273       }
274     else{
275 #ifdef SYSCALL
276       printf("Current directory = %s\n", path);
277       printf("max strlen(path) is %d, strlen(path) = %d\n", MAXPATHLEN-1, strlen(path));
278       printf("Canary should be 7. Canary = %d\n", canary);
279       if (canary != 7)
280         printf("ALERT: path[MAXPATHLEN + 1] has been overflowed!\n");
```

```
#endif
    }
}


int main(int argc, char **argv){

#ifdef SYSCALL
  char orig_path[MAXPATHLEN + 20];
  FILE *f;

  assert (argc == 2);
  f = fopen(argv[1], "r");
  assert(f != NULL);

  fgets(orig_path, MAXPATHLEN + 20, f); /* get path name */
  fclose(f);

#ifdef SYSCALL
  printf("orig_path = %s\n", orig_path);
#endif
#else
  char orig_path[MAXPATHLEN + 20] = {0x2f,
0x74,0x6d,0x70,0x2f,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,
0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x00};
#endif

  mapping_chdir(orig_path); /* this overflows mapped_path[] and pathspace[] */
  pwd(); /* get current working directory.. this calls getwcd = mapping_getwd*/
                  /* mapping_getwd overflows path[] */


  return 0;

}
/*
</source>
*/
```

Listing A.2. `string.c` file in Zitser's WU-FTPD "mapping `chdir`" bug.

```
#include <string.h>
#include <stddef.h>
#include <sys/types.h>

//typedef char reg_char

// simple implementation
void* _memcpy(void *dest, const void *source, size_t num) {
    int i = 0;
    // casting pointers
    char *dest8 = (char *)dest;
    char *source8 = (char *)source;
    for (i = 0; i < num; i++) {
        dest8[i] = source8[i];
    }
    return dest;
}


char *_strchr(register const char *s, int c)
{
       do {
              if (*s == ((char)c)) {
                     return (char *) s; /* silence the warning */
              }
       } while (*s++);

       return NULL;
}
```

```
30
31 char *
32 _strcpy(char *s1, const char *s2)
33 {
34     char *s = s1;
35     while ((*s++ = *s2++) != 0)
36         ;
37     return (s1);
38 }
39
40 #undef strcat
41
42 /* Append SRC on the end of DEST. */
43 char *
44 _strcat (dest, src)
45      char *dest;
46      const char *src;
47 {
48   char *s1 = dest;
49   const char *s2 = src;
50   char c;
51
52   /* Find the end of the string. */
53   do
54     c = *s1++;
55   while (c != '\0');
56
57   /* Make S1 point before the next character, so we can increment
58      it while memory is read (wins on pipelined cpus). */
59   s1 -= 2;
60
61   do
62     {
63       c = *s2++;
64       *++s1 = c;
65     }
66   while (c != '\0');
67
68   return dest;
69 }
70
71 #undef strrchr
72
73 /* Find the last occurrence of C in S. */
74 char *
75 _strrchr (const char *s, int c)
76 {
77   register const char *found, *p;
78
79   c = (unsigned char) c;
80
81   /* Since strchr is fast, we use it rather than the obvious loop. */
82
83   if (c == '\0')
84     return strchr (s, '\0');
85
86   found = NULL;
87   while ((p = strchr (s, c)) != NULL)
88     {
89       found = p;
90       s = p + 1;
91     }
92
93   return (char *) found;
94 }
95
96
97 #undef strlen
98 /* Return the length of the null-terminated string STR. Scan for
```

```
   the null terminator quickly by testing four bytes at a time. */
size_t
_strlen (str)
    const char *str;
{
  const char *char_ptr;
  const unsigned long int *longword_ptr;
  unsigned long int longword, himagic, lomagic;

  /* Handle the first few characters by reading one character at a time.
     Do this until CHAR_PTR is aligned on a longword boundary. */
  for (char_ptr = str; ((unsigned long int) char_ptr
                        & (sizeof (longword) - 1)) != 0;
       ++char_ptr)
    if (*char_ptr == '\0')
      return char_ptr - str;

  /* All these elucidatory comments refer to 4-byte longwords,
     but the theory applies equally well to 8-byte longwords. */

  longword_ptr = (unsigned long int *) char_ptr;

  /* Bits 31, 24, 16, and 8 of this number are zero. Call these bits
     the "holes." Note that there is a hole just to the left of
     each byte, with an extra at the end:
     bits: 01111110 11111110 11111110 11111111
     bytes: AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD
     The 1-bits make sure that carries propagate to the next 0-bit.
     The 0-bits provide holes for carries to fall into. */
  himagic = 0x80808080L;
  lomagic = 0x01010101L;
  if (sizeof (longword) > 4)
    {
      /* 64-bit version of the magic. */
      /* Do the shift in two steps to avoid a warning if long has 32 bits. */
      himagic = ((himagic << 16) << 16) | himagic;
      lomagic = ((lomagic << 16) << 16) | lomagic;
    }
  if (sizeof (longword) > 8)
    abort ();

  /* Instead of the traditional loop which tests each character,
     we will test a longword at a time. The tricky part is testing
     if *any of the four* bytes in the longword in question are zero. */
  for (;;)
    {
      longword = *longword_ptr++;

      if (((longword - lomagic) & ~longword & himagic) != 0)
        {
          /* Which of the bytes was the zero? If none of them were, it was
             a misfire; continue the search. */

          const char *cp = (const char *) (longword_ptr - 1);

          if (cp[0] == 0)
            return cp - str;
          if (cp[1] == 0)
            return cp - str + 1;
          if (cp[2] == 0)
            return cp - str + 2;
          if (cp[3] == 0)
            return cp - str + 3;
          if (sizeof (longword) > 4)
            {
              if (cp[4] == 0)
                return cp - str + 4;
              if (cp[5] == 0)
```

```
167               return cp - str + 5;
168             if (cp[6] == 0)
169               return cp - str + 6;
170             if (cp[7] == 0)
171               return cp - str + 7;
172           }
173       }
174   }
175 }
```

A.2 Fixed Version: Zitser's Fix

Listing A.3. Zitser's fixed version of the `mapped-path-bad.c` file in Zitser's WU-FTPD "mapping `chdir`" bug.

```
1  /*
2  MIT Copyright Notice
3
4  Copyright 2003 M.I.T.
5
6  Permission is hereby granted, without written agreement or royalty fee, to use,
7  copy, modify, and distribute this software and its documentation for any
8  purpose, provided that the above copyright notice and the following three
9  paragraphs appear in all copies of this software.
10
11 IN NO EVENT SHALL M.I.T. BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL,
12 INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
13 AND ITS DOCUMENTATION, EVEN IF M.I.T. HAS BEEN ADVISED OF THE POSSIBILITY OF
14 SUCH DAMANGE.
15
16 M.I.T. SPECIFICALLY DISCLAIMS ANY WARRANTIES INCLUDING, BUT NOT LIMITED TO
17 THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
18 AND NON-INFRINGEMENT.
19
20 THE SOFTWARE IS PROVIDED ON AN "AS-IS" BASIS AND M.I.T. HAS NO OBLIGATION TO
21 PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
22
23 $Author: tleek $
24 $Date: 2004/02/05 15:21:24 $
25 $Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-ok.c,v 1.2 2004/02/05
       15:21:24 tleek Exp $
26 */
27 /*
28 WU-FTPD Copyright Notice
29
30
31   Copyright (c) 1999,2000 WU-FTPD Development Group.
32   All rights reserved.
33
34   Portions Copyright (c) 1980, 1985, 1988, 1989, 1990, 1991, 1993, 1994
35     The Regents of the University of California.
36   Portions Copyright (c) 1993, 1994 Washington University in Saint Louis.
37   Portions Copyright (c) 1996, 1998 Berkeley Software Design, Inc.
38   Portions Copyright (c) 1989 Massachusetts Institute of Technology.
39   Portions Copyright (c) 1998 Sendmail, Inc.
40   Portions Copyright (c) 1983, 1995, 1996, 1997 Eric P. Allman.
41   Portions Copyright (c) 1997 by Stan Barber.
42   Portions Copyright (c) 1997 by Kent Landfield.
43   Portions Copyright (c) 1991, 1992, 1993, 1994, 1995, 1996, 1997
44     Free Software Foundation, Inc.
45
46   Use and distribution of this software and its source code are governed
47   by the terms and conditions of the WU-FTPD Software License ("LICENSE").
48
49   If you did not receive a copy of the license, it may be obtained online
50   at http://www.wu-ftpd.org/license.html.
51
52
53 $Author: tleek $
54 $Date: 2004/02/05 15:21:24 $
```

```
$Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-ok.c,v 1.2 2004/02/05
    15:21:24 tleek Exp $
*/
/*
<source>
*/
#include <sys/types.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/*#include <pwd.h>*/ /* Using custom made pwd() */
#include <string.h>
#include <unistd.h>
#include "my-include.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>

#ifdef MAPPING_CHDIR
/* Keep track of the path the user has chdir'd into and respond with
 * that to pwd commands. This is to avoid having the absolue disk
 * path returned, which I want to avoid.
 */
char mapped_path[ MAXPATHLEN ] = "/";


char *mapping_getcwd(char *path, size_t size) /* NEW.. a replacement for mapping_getwd */
{

  printf("Copying at most %d chars into buffer path[] whose size = %d\n", size, MAXPATHLEN
      + 1);

  /* OK */
  strncpy(path, mapped_path, size);
  path[size-1] = '\0';
  return path;
}


/* Make these globals rather than local to mapping_chdir to avoid stack overflow */
char pathspace[ MAXPATHLEN ];
char old_mapped_path[ MAXPATHLEN ];


void
#ifdef __STDC__
do_elem(char *dir)
#else
do_elem( dir )
     char *dir;
#endif
{
     /* . */
     if( dir[0] == '.' && dir[1] == '\0' ){
            /* ignore it */
            return;
     }

     /* .. */
     if( dir[0] == '.' && dir[1] == '.' && dir[2] == '\0' ){
            char *last;
            /* lop the last directory off the path */
            if(( last = strrchr( mapped_path, '/') )){
                   /* If start of pathname leave the / */
                   if( last == mapped_path )
                          last++;
                   *last = '\0';
            }
            return;
```

```
122       }
123
124       /* append the dir part with a leading / unless at root */
125       if( !(mapped_path[0] == '/' && mapped_path[1] == '\0') )
126         if (strlen(mapped_path) < sizeof(mapped_path) - 1) /* NEW */
127           /*OK*/
128           strcat(mapped_path, "/");
129
130       if (sizeof(mapped_path) - strlen(mapped_path) > 1) /* NEW */
131         /*OK*/
132         strncat(mapped_path, dir, sizeof(mapped_path) - strlen(mapped_path) - 1); /* NEW */
133 }
134
135
136 int
137 #ifdef __STDC__
138 mapping_chdir(char *orig_path)
139 #else
140 mapping_chdir( orig_path )
141       char *orig_path;
142 #endif
143 {
144       int ret;
145       char *sl, *path;
146
147       strcpy( old_mapped_path, mapped_path );
148       path = &pathspace[0];
149
150       /*OK*/
151       printf ("strlen(path) = %d \t path=%s\n", strlen (path), path);
152       printf ("strlen(orig_path) = %d \t orig_path=%s\n", strlen (orig_path), orig_path);
153
154       if (strlen (path) >0)
155         strcpy( path, orig_path );
156
157       /* / at start of path, set the start of the mapped_path to / */
158       if( path[0] == '/' ){
159              mapped_path[0] = '/';
160              mapped_path[1] = '\0';
161              path++;
162       }
163
164       while( (sl = strchr( path, '/' )) ){
165              char *dir;
166              dir = path;
167              *sl = '\0';
168              path = sl + 1;
169              if( *dir )
170                     do_elem( dir );
171              if( *path == '\0' )
172                     break;
173       }
174
175       if( *path ){
176         printf("path = %s.. calling do_elem\n", path);
177         do_elem( path );
178       }
179       printf("mapped_path = %s\n", mapped_path);
180
181       if (strlen(mapped_path) >= MAXPATHLEN){
182         printf("ALERT: mapped_path[MAXPATHLEN] has been overflowed!\n");
183       }
184
185       if( (ret = chdir( mapped_path )) < 0 ){
186         printf("couldn't chdir to %s !\n", mapped_path);
187         strcpy(mapped_path, old_mapped_path );
188         printf("mapped_path changed to %s\n", mapped_path);
189       }
```

```
190
191       return ret;
192 }
193
194
195 #define getwd(d) mapping_getwd(d)
196 #define getcwd(d,u) mapping_getcwd((d), (u)) /* NEW */
197
198 #endif /* MAPPING_CHDIR */
199
200
201 /* Define pwd */
202
203 void
204 #ifdef __STDC__
205 pwd(void)
206 #else
207 pwd()
208 #endif
209 {
210   int canary = 7;
211   char path[MAXPATHLEN + 1]; /* Path to return to client */
212
213 #ifndef MAPPING_CHDIR
214 #ifdef HAVE_GETCWD
215   extern char *getcwd();
216 #else
217 #ifdef __STDC__
218   extern char *getwd(char *);
219 #else
220   extern char *getwd();
221 #endif
222 #endif
223 #endif /* MAPPING_CHDIR */
224
225 #ifdef HAVE_GETCWD
226   if (getcwd(path,MAXPATHLEN) == (char *) NULL) /* call is made to mapping_getcwd */
227     {
228       printf("wu-ftpd: Illegal path supplied!\n");
229     }
230 #else
231     if (getwd(path) == (char *) NULL)
232 #endif
233       {
234       printf("path = %s\n", path);
235       printf("Current directory = %s\n", path);
236       printf("max len of path is %d, strlen(path) = %d\n", MAXPATHLEN, strlen(path));
237       printf("Canary should be 7. Canary = %d\n", canary);
238       if (canary != 7)
239         printf("ALERT: path[MAXPATHLEN + 1] has been overflowed!\n");
240       }
241 }
242
243
244   int main(int argc, char **argv){
245
246   char orig_path[MAXPATHLEN + 20];
247   FILE *f;
248
249   f = fopen("pathfile", "r");
250   fgets(orig_path, MAXPATHLEN + 20, f); /* get path name */
251   fclose(f);
252
253   printf("orig_path = %s\n", orig_path);
254
255   mapping_chdir(orig_path); /* this can overflow mapped_path[] and pathspace[] */
256   pwd(); /* get current working directory.. this calls getwcd = mapping_getwd*/
257                   /* mapping_getwd may overflow path[] */
258
```

```
259
260   return 0;
261
262 }
263 /*
264 </source>
265 */
```

A.3 Fixed Version: Our Fix

Listing A.4. `mapped-path-bad.c` file in Zitser's WU-FTPD "mapping `chdir`" bug, with PCFIRE-C-prescribed safety-checks inserted.

```
1  /*
2  MIT Copyright Notice
3
4  Copyright 2003 M.I.T.
5
6  Permission is hereby granted, without written agreement or royalty fee, to use,
7  copy, modify, and distribute this software and its documentation for any
8  purpose, provided that the above copyright notice and the following three
9  paragraphs appear in all copies of this software.
10
11 IN NO EVENT SHALL M.I.T. BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL,
12 INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
13 AND ITS DOCUMENTATION, EVEN IF M.I.T. HAS BEEN ADVISED OF THE POSSIBILITY OF
14 SUCH DAMANGE.
15
16 M.I.T. SPECIFICALLY DISCLAIMS ANY WARRANTIES INCLUDING, BUT NOT LIMITED TO
17 THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
18 AND NON-INFRINGEMENT.
19
20 THE SOFTWARE IS PROVIDED ON AN "AS-IS" BASIS AND M.I.T. HAS NO OBLIGATION TO
21 PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
22
23 $Author: tleek $
24 $Date: 2004/01/05 17:27:52 $
25 $Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-bad.c,v 1.1.1.1 2004/01/05
       17:27:52 tleek Exp $
26 */
27 /*
28 WU-FTPD Copyright Notice
29
30
31   Copyright (c) 1999,2000 WU-FTPD Development Group.
32   All rights reserved.
33
34   Portions Copyright (c) 1980, 1985, 1988, 1989, 1990, 1991, 1993, 1994
35     The Regents of the University of California.
36   Portions Copyright (c) 1993, 1994 Washington University in Saint Louis.
37   Portions Copyright (c) 1996, 1998 Berkeley Software Design, Inc.
38   Portions Copyright (c) 1989 Massachusetts Institute of Technology.
39   Portions Copyright (c) 1998 Sendmail, Inc.
40   Portions Copyright (c) 1983, 1995, 1996, 1997 Eric P. Allman.
41   Portions Copyright (c) 1997 by Stan Barber.
42   Portions Copyright (c) 1997 by Kent Landfield.
43   Portions Copyright (c) 1991, 1992, 1993, 1994, 1995, 1996, 1997
44     Free Software Foundation, Inc.
45
46   Use and distribution of this software and its source code are governed
47   by the terms and conditions of the WU-FTPD Software License ("LICENSE").
48
49   If you did not receive a copy of the license, it may be obtained online
50   at http://www.wu-ftpd.org/license.html.
51
52
53 $Author: tleek $
54 $Date: 2004/01/05 17:27:52 $
```

```
55 $Header: /mnt/leo2/cvs/sabo/hist-040105/wu-ftpd/f1/mapped-path-bad.c,v 1.1.1.1 2004/01/05
       17:27:52 tleek Exp $
56 */
57 /*
58 <source>
59 */
60 //#define SYSCALL
61
62 #ifdef SYSCALL
63 #include <ctype.h>
64 #include <stdlib.h>
65 /*#include <pwd.h> */ /* Using custom made pwd() */
66 #include <string.h>
67 #include <unistd.h>
68 #include <sys/stat.h>
69 #include <fcntl.h>
70 #include <assert.h>
71 #include <stdio.h>
72 #else
73 #define NULL ((void *) 0)
74 #endif
75 #include "my-include.h"
76 #include <sys/types.h>
77
78 #define MIN_SAFE_MEM 0x9000
79 #define WORD_SIZE 4
80 #define ABS_MAX_SAFE_MEM 0xBF000000
81 #define MAX_SAFE_MEM ((unsigned int) (ABS_MAX_SAFE_MEM - WORD_SIZE))
82
83 #define GET_STACK_POINTER(dest_var)  \
84   asm ( "mov r5,r13"    \
85         : "=r" (dest_var)   \
86         : /* no inputs */   \
87         : /* no clobber */)
88 #define GET_FRAME_POINTER(dest_var)  \
89   asm ( "mov r4,r11"    \
90         : "=r" (dest_var)   \
91         : /* no inputs */   \
92         : /* no clobber */)
93
94 /* Dummy chdir function to silence syscalls
95  * pwd is localized, so chdir effectively does nothing except
96  * check that an actual directory was created by the Makefile
97  */
98  #ifndef SYSCALL
99 int chdir(const char *path) {
100    return 0;
101 }
102  #endif
103
104 #ifdef MAPPING_CHDIR
105 /* Keep track of the path the user has chdir'd into and respond with
106  * that to pwd commands. This is to avoid having the absolue disk
107  * path returned, which I want to avoid.
108  */
109 char mapped_path[ MAXPATHLEN ] = "/";
110
111 char *
112 #ifdef __STDC__
113 mapping_getwd(char *path)
114 #else
115     mapping_getwd( path )
116     char *path;
117 #endif
118 {
119
120 #ifdef SYSCALL
121   printf("Copying %d chars into buffer path[] whose size = %d\n", strlen(mapped_path) + 1,
       MAXPATHLEN + 1);
```

```
122 #endif
123
124   /* BAD */
125   strcpy( path, mapped_path ); /* copies mapped_path to path without doing a size check */
126   return path;
127 }
128
129 /* Make these globals rather than local to mapping_chdir to avoid stack overflow */
130 char pathspace[ MAXPATHLEN ]; /* This buffer can get overflowed too */
131 char old_mapped_path[ MAXPATHLEN ];
132
133 void
134 #ifdef __STDC__
135 /* appends /dir to mapped_path if mapped_path != /, else appends simply dir */
136 do_elem(char *dir)
137 #else
138     do_elem( dir )
139     char *dir;
140 #endif
141 {
142   /* . */
143   register unsigned int r11_val asm ("r4");
144
145   if( dir[0] == '.' && dir[1] == '\0' ){
146         /* ignore it */
147         return;
148       }
149
150       /* .. */
151       if( dir[0] == '.' && dir[1] == '.' && dir[2] == '\0' ){
152              char *last;
153              /* lop the last directory off the path */
154              if (( last = strrchr( mapped_path, '/' ))){
155                /* If start of pathname leave the / */
156                if( last == mapped_path )
157                  last++;
158                GET_FRAME_POINTER(r11_val);
159                if ((((unsigned int) last) >= MIN_SAFE_MEM) && (((unsigned int) last) <=
                       MAX_SAFE_MEM) &&
160                    (((unsigned int) last) <= (r11_val-(3*WORD_SIZE))) && (r11_val >= (3*
                           WORD_SIZE))) {
161                  *last = '\0';
162                }
163              }
164              return;
165       }
166
167       /* append the dir part with a / unless at root */
168       if( !(mapped_path[0] == '/' && mapped_path[1] == '\0') )
169         /* BAD */
170         strcat( mapped_path, "/" ); /* no bounds checking is done */
171       /* We do not check to see if there is room in mapped_path for dir */
172       /* BAD */
173       strcat( mapped_path, dir ); /* no bounds checking is done */
174 }
175
176 int
177 #ifdef __STDC__
178 mapping_chdir(char *orig_path)
179 #else
180 mapping_chdir( orig_path )
181       char *orig_path;
182 #endif
183 {
184       int ret;
185       char *sl, *path;
186       register unsigned int r11_val asm ("r4");
187
```

```
188 #ifdef SYSCALL
189       printf("Entering mapping_chdir with orig_path = %s\n", orig_path);
190 #endif
191
192       strcpy( old_mapped_path, mapped_path ); /* old_mapped_path is initially / */
193       path = &pathspace[0];
194
195       /* BAD */
196       strcpy( path, orig_path ); /* suppose path = orig_path = /x/xx/xxx/xxxx/... */
197 #ifdef SYSCALL
198       printf("Copying orig_path to path....max strlen(path) = %d. strlen(path) = %d\n",
              MAXPATHLEN - 1, strlen(path));
199       if (strlen(path) >= MAXPATHLEN){
200         printf ("ALERT:pathspace[MAXPATHLEN] has been overflowed!\n");
201       }
202 #endif
203
204         /* set the start of the mapped_path to / */
205         if( path[0] == '/' ){
206           char *c;
207           c = &(mapped_path[0]);
208           GET_FRAME_POINTER(r11_val);
209           if ((((unsigned int) (c)) >= MIN_SAFE_MEM) && (((unsigned int) (c)) <=
                  MAX_SAFE_MEM) &&
210               (((unsigned int) (c)) <= (r11_val-(4*WORD_SIZE))) && (r11_val >= (4*WORD_SIZE))
                      ) {
211             *c = '/';
212           }
213           c = &(mapped_path[1]);
214           GET_FRAME_POINTER(r11_val);
215           if ((((unsigned int) (c)) >= MIN_SAFE_MEM) && (((unsigned int) (c)) <=
                  MAX_SAFE_MEM) &&
216               (((unsigned int) (c)) <= (r11_val-(4*WORD_SIZE))) && (r11_val >= (4*WORD_SIZE))
                      ) {
217             *c = '\0';
218           }
219           path++;
220         }
221
222 #ifdef SYSCALL
223       printf("so far mapped_path = %s\n", mapped_path);
224 #endif
225
226       while( (sl = strchr( path, '/' )) ){
227              char *dir;
228              dir = path;
229              GET_FRAME_POINTER(r11_val);
230              if ((((unsigned int) sl) >= MIN_SAFE_MEM) && (((unsigned int) sl) <=
                     MAX_SAFE_MEM) &&
231                  (((unsigned int) sl) <= (r11_val-(4*WORD_SIZE))) && (r11_val >= (4*
                         WORD_SIZE))) {
232                *sl = '\0';
233              }
234              path = sl + 1;
235              if( *dir )
236                do_elem( dir ); /* appends directory names to mapped_path */
237              if( *path == '\0' )
238                break;
239       }
240       if( *path )
241         {
242 #ifdef SYSCALL
243           printf("path = %s.. calling do_elem\n", path);
244 #endif
245           do_elem( path ); /* we're in root and path is of the form aaaaa ... mapped_path
                  becomes /aaaa.. */
246         }
```

```
247 #ifdef SYSCALL
248       printf("mapped_path = %s\n", mapped_path);
249 #endif
250 #ifdef SYSCALL
251       if (strlen(mapped_path) >= MAXPATHLEN){
252         printf("ALERT: mapped_path[MAXPATHLEN] has been overflowed!\n");
253       }
254 #endif
255
256
257       if( (ret = chdir( mapped_path )) < 0 ){ /* change to the specified path */
258 #ifdef SYSCALL
259         printf("couldn't chdir to %s !\n", mapped_path);
260 #endif
261         strcpy( mapped_path, old_mapped_path ); /* change mapped_path back to original, i.e
              root */
262 #ifdef SYSCALL
263         printf("mapped_path changed to %s\n", mapped_path);
264 #endif
265       }
266
267       return ret;
268 }
269
270
271 /* From now on use the mapping version */
272
273 #define getwd(d) mapping_getwd(d)
274 #define getcwd(d,u) mapping_getwd(d)
275
276 #endif /* MAPPING_CHDIR */
277
278
279
280 /* Define pwd */
281
282 void
283 #ifdef __STDC__
284 pwd(void)
285 #else
286 pwd()
287 #endif
288 {
289   int canary = 7; /* used to see if path[] gets overflowed */
290   char path[MAXPATHLEN + 1]; /* Path to return to client */
291
292 #ifndef MAPPING_CHDIR
293 #ifdef HAVE_GETCWD
294   extern char *getcwd();
295 #else
296 #ifdef __STDC__
297   extern char *getwd(char *);
298 #else
299   extern char *getwd();
300 #endif
301 #endif
302 #endif /* MAPPING_CHDIR */
303
304 #ifdef HAVE_GETCWD
305   if (getcwd(path,MAXPATHLEN) == (char *) NULL) /* mz: call to mapping_getwd might overflow
         path */
306 #else
307     if (getwd(path) == (char *) NULL) /* mz: call to mapping_getwd might overflow path buf
          */
308 #endif
309       {
310 #ifdef SYSCALL
311         printf("Couldn't get current directory!\n");
312 #endif
```

```
313       }
314     else{
315 #ifdef SYSCALL
316       printf("Current directory = %s\n", path);
317       printf("max strlen(path) is %d, strlen(path) = %d\n", MAXPATHLEN-1, strlen(path));
318       printf("Canary should be 7. Canary = %d\n", canary);
319       if (canary != 7)
320         printf("ALERT: path[MAXPATHLEN + 1] has been overflowed!\n");
321 #endif
322     }
323 }
324
325
326 int main(int argc, char **argv){
327
328 #ifdef SYSCALL
329   char orig_path[MAXPATHLEN + 20];
330   FILE *f;
331
332   assert (argc == 2);
333   f = fopen(argv[1], "r");
334   assert(f != NULL);
335
336   fgets(orig_path, MAXPATHLEN + 20, f); /* get path name */
337   fclose(f);
338
339 #ifdef SYSCALL
340   printf("orig_path = %s\n", orig_path);
341 #endif
342 #else
343   char orig_path[MAXPATHLEN + 20] = {0x2f,
344 0x74,0x6d,0x70,0x2f,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,
345 0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x00};
346 #endif
347
348   mapping_chdir(orig_path); /* this overflows mapped_path[] and pathspace[] */
349   pwd(); /* get current working directory.. this calls getwcd = mapping_getwd*/
350                   /* mapping_getwd overflows path[] */
351
352
353   return 0;
354
355 }
356 /*
357 </source>
358 */
```

Listing A.5. `string.c` file in Zitser's WU-FTPD "mapping `chdir`" bug, with PCFIRE-C-prescribed safety-checks inserted.

```
1  #include <string.h>
2  #include <stddef.h>
3  #include <sys/types.h>
4
5  #define MIN_SAFE_MEM 0x9000
6  #define WORD_SIZE 4
7  #define ABS_MAX_SAFE_MEM 0xBF000000
8  #define MAX_SAFE_MEM ((unsigned int) (ABS_MAX_SAFE_MEM - WORD_SIZE))
9
10 #define GET_STACK_POINTER(dest_var)  \
11   asm ( "mov r5,r13"    \
12         : "=r" (dest_var)   \
13         : /* no inputs */   \
14         : /* no clobber */)
15 #define GET_FRAME_POINTER(dest_var)  \
16   asm ( "mov r4,r11"    \
17         : "=r" (dest_var)   \
18         : /* no inputs */   \
19         : /* no clobber */)
```

```
20
21 //typedef char reg_char
22
23 // simple implementation
24 void* memcpy(void *dest, const void *source, size_t num) {
25     int i = 0;
26     register unsigned int r11_val asm ("r4");
27     // casting pointers
28     char *dest8 = (char *)dest;
29     char *source8 = (char *)source;
30     for (i = 0; i < num; i++) {
31       GET_FRAME_POINTER(r11_val);
32       if ((((unsigned int) (dest8 + i)) >= MIN_SAFE_MEM) && (((unsigned int) (dest8 + i)) <=
              MAX_SAFE_MEM) &&
33          (((unsigned int) (dest8 + i)) <= (r11_val-(2*WORD_SIZE))) && (r11_val >= (2*
                 WORD_SIZE))) {
34         dest8[i] = source8[i];
35       }
36     }
37     return dest;
38 }
39
40
41 char *strchr(register const char *s, int c)
42 {
43        do {
44               if (*s == ((char)c)) {
45                      return (char *) s; /* silence the warning */
46               }
47        } while (*s++);
48
49        return NULL;
50 }
51
52 char *
53 strcpy(char *s1, const char *s2)
54 {
55   register unsigned int r11_val asm ("r4");
56   int res;
57     char *s = s1;
58     do {
59       res = 0;
60       GET_FRAME_POINTER(r11_val);
61       if ((((unsigned int) s) >= MIN_SAFE_MEM) && (((unsigned int) s) <= MAX_SAFE_MEM) &&
62          (((unsigned int) s) <= (r11_val-(3*WORD_SIZE))) && (r11_val >= (3*WORD_SIZE))) {
63         res = (*s++ = *s2++);
64       }
65     } while (res != 0);
66     // while ((*s++ = *s2++) != 0)
67     // ;
68     return (s1);
69 }
70
71 #undef strcat
72
73 /* Append SRC on the end of DEST. */
74 char *
75 strcat (dest, src)
76      char *dest;
77      const char *src;
78 {
79   register unsigned int r11_val asm ("r4");
80   char *s1 = dest;
81   const char *s2 = src;
82   char c;
83
84   /* Find the end of the string. */
85   do
```

```
86      c = *s1++;
87    while (c != '\0');
88
89    /* Make S1 point before the next character, so we can increment
90       it while memory is read (wins on pipelined cpus). */
91    s1 -= 2;
92
93    do
94      {
95        char *tmp_dst;
96        c = *s2++;
97        tmp_dst = ++s1;
98        GET_FRAME_POINTER(r11_val);
99        if ((((unsigned int) (tmp_dst)) >= MIN_SAFE_MEM) && (((unsigned int) (tmp_dst)) <=
              MAX_SAFE_MEM) &&
100           (((unsigned int) (tmp_dst)) <= (r11_val-(3*WORD_SIZE))) && (r11_val >= (3*
                WORD_SIZE))) {
101         *tmp_dst = c;
102       }
103     }
104   while (c != '\0');
105
106   return dest;
107 }
108
109 #undef strrchr
110
111 /* Find the last occurrence of C in S. */
112 char *
113 strrchr (const char *s, int c)
114 {
115   register const char *found, *p;
116
117   c = (unsigned char) c;
118
119   /* Since strchr is fast, we use it rather than the obvious loop. */
120
121   if (c == '\0')
122     return strchr (s, '\0');
123
124   found = NULL;
125   while ((p = strchr (s, c)) != NULL)
126     {
127       found = p;
128       s = p + 1;
129     }
130
131   return (char *) found;
132 }
133
134
135 #undef strlen
136 /* Return the length of the null-terminated string STR. Scan for
137    the null terminator quickly by testing four bytes at a time. */
138 size_t
139 strlen (str)
140      const char *str;
141 {
142   const char *char_ptr;
143   const unsigned long int *longword_ptr;
144   unsigned long int longword, himagic, lomagic;
145
146   /* Handle the first few characters by reading one character at a time.
147      Do this until CHAR_PTR is aligned on a longword boundary. */
148   for (char_ptr = str; ((unsigned long int) char_ptr
149                         & (sizeof (longword) - 1)) != 0;
150        ++char_ptr)
151     if (*char_ptr == '\0')
152       return char_ptr - str;
```

```
153
154   /* All these elucidatory comments refer to 4-byte longwords,
155      but the theory applies equally well to 8-byte longwords. */
156
157   longword_ptr = (unsigned long int *) char_ptr;
158
159   /* Bits 31, 24, 16, and 8 of this number are zero. Call these bits
160      the "holes." Note that there is a hole just to the left of
161      each byte, with an extra at the end:
162      bits: 01111110 11111110 11111110 11111111
163      bytes: AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD
164      The 1-bits make sure that carries propagate to the next 0-bit.
165      The 0-bits provide holes for carries to fall into. */
166   himagic = 0x80808080L;
167   lomagic = 0x01010101L;
168   if (sizeof (longword) > 4)
169     {
170       /* 64-bit version of the magic. */
171       /* Do the shift in two steps to avoid a warning if long has 32 bits. */
172       himagic = ((himagic << 16) << 16) | himagic;
173       lomagic = ((lomagic << 16) << 16) | lomagic;
174     }
175   if (sizeof (longword) > 8)
176     abort ();
177
178   /* Instead of the traditional loop which tests each character,
179      we will test a longword at a time. The tricky part is testing
180      if *any of the four* bytes in the longword in question are zero. */
181   for (;;)
182     {
183       longword = *longword_ptr++;
184
185       if (((longword - lomagic) & ~longword & himagic) != 0)
186         {
187           /* Which of the bytes was the zero? If none of them were, it was
188              a misfire; continue the search. */
189
190           const char *cp = (const char *) (longword_ptr - 1);
191
192           if (cp[0] == 0)
193             return cp - str;
194           if (cp[1] == 0)
195             return cp - str + 1;
196           if (cp[2] == 0)
197             return cp - str + 2;
198           if (cp[3] == 0)
199             return cp - str + 3;
200           if (sizeof (longword) > 4)
201             {
202               if (cp[4] == 0)
203                 return cp - str + 4;
204               if (cp[5] == 0)
205                 return cp - str + 5;
206               if (cp[6] == 0)
207                 return cp - str + 6;
208               if (cp[7] == 0)
209                 return cp - str + 7;
210             }
211         }
212     }
213 }
```

Bibliography

[1] A. Wood, “The Internet-of-Things is revolutionising our lives, but standards are a must,” March 2015, http://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must. 1

[2] “Embedded Software,” *Advances in Computers*, vol. 56, 2002. 1

[3] I. Gartner, “Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015,” 2015, http://www.gartner.com/newsroom/id/3165317. 1

[4] Intel, Inc., “Internet of Things Applications Across Industries,” 2016, http://www.intel.com/content/www/us/en/internet-of-things/industry-solutions.html. 1

[5] C. Miller and C. Valasek, “Remote Exploitation of an Unaltered Passenger Vehicle,” http://bit.ly/1Xk71rn. 1, 9

[6] A. Greenberg and K. Zetter, “How the Internet of Things Got Hacked,” 2015, https://www.wired.com/2015/12/2015-the-year-the-internet-of-things-got-hacked/. 1

[7] Symantec Security Response, “IoT devices being increasingly used for DDoS attacks,” 2016, http://www.symantec.com/connect/blogs/iot-devices-being-increasingly-used-ddos-attacks. 1

[8] D. Jacoby, “IoT: How I hacked my home,” 2014, https://securelist.com/analysis/publications/66207/iot-how-i-hacked-my-home/. 1

[9] J. Knight, “Safety Critical Systems: Challenges and Directions,” in *ICSE*, 2002. 2

[10] K. Zhao and L. Ge, “A Survey on the Internet of Things Security,” 2013. 2

[11] M. Rahman, B. Carbunar, and U. Topkara, "SensCrypt: A Secure Protocol for Managing Low Power Fitness Trackers," in *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2014. 2

[12] A. Murugesan, O. Sokolsky, S. Rayadurgam, M. Whalen, M. Heimdahl, and I. Lee, "Linking Abstract Analysis to Concrete Design: A Hierarchical Approach to Verify Medical CPS Safety," in *ICCPS*, Apr 2014. 2

[13] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam, "Safety-critical Medical Device Development Using the UPP2SF Model Translation Tool," *ACM TECS*, vol. 13, no. 4s, Apr. 2014. 2

[14] F. Allen, "Control-Flow Analysis," in *SIGPLAN Notices*, 1970. 4, 5

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *ACM CCS*, 2005. 4, 5, 7, 8, 9, 14, 18, 24, 25, 26, 27, 28, 40, 42, 126, 166

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software Guards for System Address Spaces," in *OSDI*, 2006. 4, 7, 8, 9, 17, 18, 26, 27, 28, 33, 40, 126

[17] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *USENIX Security*, 2006. 4, 9, 18, 26, 27, 28, 34, 40

[18] L. Zhao, G. Li, B. D. Sutter, and J. Regehr, "ARMor: Fully Verified Software Fault Isolation," in *EMSOFT*, 2011. 4, 7, 8, 9, 16, 17, 18, 26, 27, 28, 29, 33, 40, 124, 126, 139

[19] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A Practical System for Verifying Concurrent C," in *TPHOLs*, 2009. 5

[20] B. Jacobs, J. Smans, and F. Piessens, "VeriFast: Imperative Programs as Proofs," in *Conference on Verified Software: Theories, Tools and Experiments (VSTTE): VS-Tools Workshop*, 2010. 5

[21] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *POPL*, 2006. 5, 18, 37, 38

[22] U.S. F.D.A., "MAUDE Adverse Event Report: BAXTER HEALTHCARE PTE. LTD." Aug 2007, http://1.usa.gov/25NWCKC. 5, 6

[23] S. Checkoway et al., "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security*, 2011. 6

[24] "Gartner says the Internet of Things installed base will grow to 26 billion units by 2020," Dec 2013, http://gtnr.it/1h780GZ. 6

[25] C. Zhang et al., "Practical Control Flow Integrity & Randomization for Binary Executables," in *IEEE Security & Privacy*, 2013. 7, 8, 18, 26, 27, 28, 40, 126, 166

[26] B. Zeng, G. Tan, and G. Morrisett, "Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing," in *ACM CCS*, 2011. 7, 8, 18, 26, 27, 28, 40

[27] Radio Technical Commission for Aeronautics (RTCA), "DO-178C: Software Considerations in Airborne Systems and Equipment Certification," 2012. 8, 20

[28] "The ARM-THUMB Procedure Call Standard," 2000, http://bit.ly/1NbOQhT. 12, 46, 52, 152

[29] "Linux Programmer's Manual: Syscalls," http://bit.ly/1VChJMY. 12, 107

[30] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security*, 2015. 14, 28, 38, 41

[31] M. Myreen and M. Gordon, "Hoare Logic for Realistically Modeled Machine Code," in *TACAS*, 2007. 15, 19, 33, 34, 63, 64, 69, 75, 78, 80, 94, 95, 97, 100, 123

[32] M. Myreen, A. Fox, and M. Gordon, "Hoare Logic for ARM Machine Code," in *Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN)*, 2007. 15, 19, 33, 34, 63, 64, 65, 67, 78, 80, 123, 139

[33] Guthaus, M. et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE WWC Workshop*, 2001. 18, 122, 124, 125, 126, 127, 128, 146, 163, 164

[34] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks – Past, Present and Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010. 18, 122, 127, 128, 146, 163, 164

[35] J. Yang and C. Hawblitzel, "Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System," in *PLDI*, 2010. 18, 154

[36] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury, "Building Embedded Systems with Embedded DSLs," in *ACM SIGPLAN International Conference on Functional Programming*, 2014. 18, 32, 33

[37] G. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," in *POPL*, 2002. 18, 31

[38] K. Slind and M. Norrish, "A Brief Overview of HOL4," in *TPHOLs*, 2008. 19, 33, 63, 64

[39] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, vol. 2, 2000, pp. 119–129 vol.2. 22

[40] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. 22, 26, 30

[41] Aleph One, "Smashing The Stack For Fun And Profit," 1996, http://www.phrack.org/issues/49/14.html#article. 22

[42] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security Privacy*, vol. 2, no. 4, pp. 20–27, July 2004. 22

[43] "Execute never bits," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html. 23

[44] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, 1998. 23, 28, 29, 143

[45] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Security & Privacy*, 2013. 23

[46] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," in *SOSP*, 1993. 24, 33, 126

[47] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "A Theory of Secure Control Flow," in *International Conference on Formal Engineering Methods (ICFEM)*, 2005. 29, 40

[48] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Security & Privacy*, 2008. 26, 27, 40

[49] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Osaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *IEEE Security & Privacy*, 2009. 26, 27, 28, 40

[50] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating Code-Reuse Attacks with Control-Flow Locking," in *ACSAC*, 2011. 26, 27, 40

[51] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A. Sadeghi, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones," in *NDSS*, 2012. 26, 27, 40

[52] J. Pewny and T. Holz, "Control-Flow Restrictor: Compiler-based CFI for iOS," in *ACSAC*, 2013. 26, 27, 40

[53] B. Niu and G. Tan, "Monitor Integrity Protection with Space Efficiency and Separate Compilation," in *ACM CCS*, 2013. 26, 27, 40, 168

[54] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX Security*, 2013. 26, 27, 40

[55] B. Niu and G. Tan, "Modular Control-Flow Integrity," in *PLDI*, 2014. 26, 27, 28, 40, 166, 168

[56] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *IEEE Security & Privacy*, 2014. 26, 27, 29, 40, 154

[57] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *OSDI*, 2014. 26, 27, 29, 40

[58] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *USENIX Security*, 2014. 26, 27, 40

[59] A. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, "CCFI: Cryptographically Enforced Control Flow Integrity," in *ACM CCS*, 2015. 26, 27, 40

[60] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *NDSS*, 2015. 26, 27, 30, 40

[61] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *ACM CCS*, 2015. 26, 27, 40

[62] V. van der Veen, E. Gotkas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level," in *IEEE Security & Privacy*, 2016. 26, 27, 40

[63] U. Erlingsson and F. Schneider, "IRM Enforcement of Java Stack Inspection," in *IEEE Security & Privacy*, 2000. 26

[64] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *IEEE Security & Privacy*, 2015. 27

[65] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution Via Program Shepherding," in *USENIX Security*, 2002. 29

[66] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *USENIX ATC*, 2008. 29

[67] K. Scott and J. Davidson, "Safe Virtual Execution Using Software Dynamic Translation," in *ACSAC*, 2002. 29

[68] V. van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *ACM CCS*, 2015. 29

[69] R. de Clercq et al., "SOFIA: Software and Control Flow Integrity Architecture," in *DATE*, 2016. 29

[70] I. Fratric, "ROPGuard: Runtime prevention of return-oriented programming attacks." 30

[71] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. Deng, "ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks," in *NDSS*, 2014. 30, 38

[72] V. Pappas, M. Polychronakis, and A. Keromytis, "Transparent ROP Exploit Mitigation using Indirect Branch Tracing," in *USENIX Security*, 2013. 30, 38

[73] S. Kowshik, D. Dhurjati, and V. Adve, "Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems," in *CASES*, 2002. 30

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

[75] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory Safety Without Runtime Checks or Garbage Collection," in *LCTES*, 2003. 30

[76] D. Dhurjati, S. Kowshik, and V. Adve, "Enforcing Alias Analysis for Weakly Typed Languages," in *PLDI*, 2006. 30

[77] F. Pfenning, "C0: Specification and Verification in Introductory Computer Science," 2010, http://c0.typesafety.net/. 31

[78] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *PLDI*, 2009. 31, 32

[79] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI*, 2006. 31

[80] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From System F to Typed Assembly Language," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, pp. 527–568, May 1999. 32, 154

[81] ——, "From System F to Typed Assembly Language," in *POPL*, 1998. 32, 154

[82] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "TALx86: A Realistic Typed Assembly Language," in *Workshop on Compiler Support for System Software (WCSSS)*, 1999. 32, 154

[83] G. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *OSDI*, Oct 1996. 32, 35

[84] L. Zhao, G. Li, and J. Regehr, "A Practical Logic Framework for Verifying Safety Properties of Executables," in *LOLA*, 2011. 33, 34

[85] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan, "RockSalt: Better, Faster, Stronger SFI for the x86," in *PLDI*, 2012. 34

[86] "ACL2," http://www.cs.utexas.edu/users/moore/acl2/. 34, 35

[87] "The Coq Proof Assistant," https://coq.inria.fr/. 34

[88] A. Chlipala, "Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic," in *PLDI*, 2011. 34

[89] M. Myreen, M. Gordon, and K. Slind, "Machine-code verification for multiple architectures: An application of decompilation into logic," in *FMCAD*, 2008. 34, 35

[90] G. Tan and A. Appel, "A Compositional Logic for Control Flow," in *VMCAI*, 2006. 34

[91] S. Goel et al., "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls," in *FMCAD*, 2014. 35

[92] Y. K. Tan, M. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "A New Verified Compiler Backend for CakeML," in *ACM SIGPLAN International Conference on Functional Programming*, 2016. 35

[93] M. Myreen, K. Slind, and M. Gordon, "Decompilation into Logic — Improved," in *FMCAD*, 2012. 35

[94] D. Yu, N. Hamid, and Z. Shao, "Building Certified Libraries for PCC: Dynamic Storage Allocation," in *ESOP*, 2003. 35

[95] Z. Ni and Z. Shao, "Certified Assembly Programming with Embedded Code Pointers," in *POPL*, 2006. 35

[96] Z. Shao, "Certified Software," *Commun. ACM*, vol. 53, no. 12, Dec. 2010. 35, 37, 154

[97] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo, "CertiKOS: A Certified Kernel for Secure Cloud Computing," in *ACM Asia-Pacific Workshop on Systems (APSys)*, 2011. 35, 37, 154

[98] D. Costanzo, Z. Shao, and R. Gu, "End-to-End Verification of Information-Flow Security for C and Assembly Programs," in *PLDI*, 2016. 35

[99] X. Feng, Z. Shao, Y. Dong, and Y. Guo, "Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads," in *PLDI*, 2008. 35

[100] H. Chen, X. Wu, Z. Shao, J. Jockerman, and R. Gu, "Toward Compositional Verification of Interruptible OS Kernels and Device Drivers," in *PLDI*, 2016. 35

[101] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel," in *ACM CCS*, 2013. 36, 37, 154

[102] N. Khakpour, O. Schwarz, and M. Dam, "Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties," in *Certified Programs and Proofs (CPP)*, 2013. 36, 37, 154

[103] O. Schwarz and M. Dam, "Formal Verification of Secure User Mode Device Execution with DMA," in *International Haifa Verification Conference (HVC)*, 2014. 36, 37, 154

[104] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz, "BAP: A Binary Analysis Platform," in *CAV*, 2011. 36

[105] A. Platzer and J. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems (system description)," in *IJCAR*, 2008. 36

[106] A. Platzer, "Differential dynamic logic for hybrid systems," *Journal of Automated Reasoning*, vol. 41, 2008. 36, 105

[107] ——, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer-Verlag, New York, Inc., 2010. 36

[108] S. Mitsch and A. Platzer, "ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models," in *International Conference on Runtime Verification (RV)*, 2014. 36, 100

[109] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *IEEE Security & Privacy*, 2013. 37

[110] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, 2004. 37

[111] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor," in *USENIX Security*, 2016. 37

[112] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, May 2015. 37

[113] G. Klein et al., "seL4: Formal verification of an OS kernel," in *SOSP*, Oct 2009. 37, 154

[114] "Isabelle," https://isabelle.in.tum.de. 37

[115] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," in *CAV*, 2010. 37

[116] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999. 37

[117] Z. Xu, B. Miller, and T. Reps, "Safety Checking of Machine Code," in *PLDI*, 2000. 37, 153

[118] S. Blazy, V. Laporte, and D. Pichardie, "An Abstract Memory Functor for Verified C Static Analyzers," in *ACM SIGPLAN International Conference on Functional Programming*, 2016. 38

[119] A. Appel, "Verified Software Toolchain," in *ESOP*, 2011. 38

[120] J. H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A formally-verified C static analyzer," in *POPL*, 2015. 38

[121] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *USENIX Security*, 2014. 38, 41

[122] E. Gotkas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out Of Control: Overcoming Control-Flow Integrity," in *IEEE Security & Privacy*, 2014. 38, 41

[123] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *USENIX Security*, 2014. 38, 41

[124] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A. Sadeghi, "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks," in *ACM CCS*, 2015. 39, 41

[125] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity." 39, 41

[126] J. Tan, H. Tay, U. Drolia, R. Gandhi, and P. Narasimhan, "PCFIRE: Towards Provable Preventative Control-Flow Integrity for Embedded Software," in *EMSOFT*, 2016. 42, 62, 63, 121, 122

[127] "CWE/SANS Top 25 Most Dangerous Software Errors," 2011, http://cwe.mitre.org/top25. 43, 153, 156, 160

[128] J. Tan, H. Tay, R. Gandhi, and P. Narasimhan, "AUSPICE: Automatic Safety Property Verification for Unmodified Executables," in *VSTTE*, 2015. 46, 63, 78, 80, 82, 99, 121, 123

[129] "clang: a C language family frontend for LLVM," http://clang.llvm.org/. 52, 53

[130] S. Harbison and G. S. Jr., *C: A Reference Manual*, 5th ed. Pearson, 3 2002. 56

[131] A. Fox, "Formal specification and verification of ARM6," in *TPHOLs*, 2003. 63, 64, 78, 80, 123

[132] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, Oct. 1969. 63, 66, 75, 94

[133] J. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE LICS*, 2002. 64, 65

[134] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, 1977. 83, 89

[135] B. Beckert and A. Platzer, "Dynamic Logic with Non-rigid Functions: A Basis for Object-oriented Program Verification," in *IJCAR*, 2006. 97, 99

[136] J. Tan, H. Tay, R. Gandhi, and P. Narasimhan, "AUSPICE-R: Automatic Safety-Property Proofs for Realistic Features in Machine Code," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2016. 101

[137] "ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition," 2014. 102, 103

[138] "Application Binary Interface for the ARM Architecture," http://bit.ly/22OaMai. 103

[139] "Using the GNU Compiler Collection," 2003, https://gcc.gnu.org/onlinedocs/4.6.3/. 117

[140] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, 1st ed. Addison-Wesley, 1988. 118

[141] "Bionic," http://bit.ly/1V0cJl3. 125, 127, 128, 146

[142] "As Gadgets Shrink, ARM Still Reigns As Processor King," Sep 2013, http://onforb.es/19LIzgd. 136

[143] G. Henderson, "WiringPi: GPIO Interface library for the Raspberry Pi," http://wiringpi.com/. 136, 146

[144] N. Jia, "Detecting Human Falls with a 3-Axis Digital Accelerometer," 2009, http://bit.ly/23fXhFE. 137

[145] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *SIGSOFT/FSE*, 2004. 141

[146] M. Zitser, "Securing Software: An Evaluation of Static Source Code Analyzers," Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, Tech. Rep., 2003, M.S. Thesis. 141, 142, 147

[147] P. Godefroid and D. Luchaup, "Automatic Partial Loop Summarization in Dynamic Test Generation," 2011. 153

[148] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," in *ICSE*, 1999. 153

[149] A. Hu, "Formal Hardware Verification with BDDs: An Introduction," in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM*, 1997. 154

[150] A. Waksman and S. Sethumadhavan, "Tamper Evident Microprocessors," in *IEEE Security & Privacy*, 2010. 154

[151] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for Hardware Trust," in *Design Automation Conference (DAC)*, 2013. 154

[152] T. Sewell, M. Myreen, and G. Klein, "Translation Validation for a Verified OS Kernel," in *PLDI*, 2013. 154

[153] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," 2006. 155

[154] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft," in *Integrated Formal Methods*, 2004. 155

[155] S. Chaki and A. Datta, "ASPIER: An Automated Framework for Verifying Security Protocol Implementations," in *IEEE CSF*, 2009. 155