Formal Verification of Memory Preservation for x86-64 Assembly via Proof Generation

Joshua Alexander Bockenek

Preliminary Exam

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Freek Verbeek
Patrick R. Schaumont
Michael S. Hsiao
Changhee Jung

September 2, 2019
Blacksburg, Virginia

Keywords: Formal Verification, x86-64 Assembly, Interactive Theorem Proving, Proof Generation, Memory Preservation

Formal Verification of Memory Preservation for x86-64 Assembly via Proof Generation

Joshua Alexander Bockenek

(ABSTRACT)

Formal characterization of the memory used by a program is an important basis for security analyses and compositional verification. Proving that that program only modifies memory within specified regions, the property of *memory preservation*, is an important aspect of that. However, accurately proving memory preservation requires operating on the assembly level due to the semantic gap between high-level languages and the code that processors actually execute. This is unfortunate, as verifying programs on the assembly level is difficult. Automated methods, such as model checking, would not be able to handle many interesting functions due to the undecidability of memory preservation. Fully-interactive methods do not scale well either. The solution is to combine proof generation with interactive theorem proving in a *semi-automated manner*: let some untrusted tool extract as much information as it can from the functions under test and then generate all the necessary proofs to be completed in a theorem prover.

The first contribution of this dissertation is a control-flow-driven verification approach with mostly manual invariant specification at automatically-selected cutpoints. The memory regions and any additional preconditions must also be determined manually. This methodology was applied to 63 functions from the HermitCore unikernel library, including one recursive one, covering 2379 assembly instructions.

The second contribution of this dissertation is a syntax-driven verification approach with fully-automated invariant and memory region generation. It produces formal memory usage certificates that can be verified in Isabelle/HOL with minimal effort, the main manual work being weakening any loop invariants. This was successfully applied to 251 functions from the Xen hypervisor project, covering a total of 12 252 assembly instructions.

Acknowledgments

This work was supported in part by *the Office of Naval Research* (ONR) under grant N00014-17-1-2297 and *the Naval Sea Systems Command* (NAVSEA)/*the Naval Engineering Education Consortium* (NEEC) under grant N00174-16-C-0018. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of ONR or NAVSEA/NEEC.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

ABI application binary interface

AES Advanced Encryption Standard

ASL ARM Specification Language

BAP the Binary Analysis Platform

CAN Controller Area Network

CFG control flow graph

CFI control-flow integrity

CISC complex instruction set computer

CPU central processing unit

DiL decompilation into logic

FDL functional description language

FMUC formal memory usage certificate

GCC the GNU Compiler Collection

HOL higher-order logic

ISA instruction set architecture

ITP interactive theorem proving

LFP least fixed point

MRR memory region relation

MSB most significant bit

NAVSEA the Naval Sea Systems Command

NEEC the Naval Engineering Education Consortium

Nqthm the Boyer-Moore theorem prover

ONR the Office of Naval Research

OS operating system

QEMU Quick Emulator

RISC reduced instruction set computer

RTOS real-time operating system

SCF syntactic control flow

SIMD single instruction, multiple data

SLOC source lines of code

SMT satisfiability modulo theories

SPADE the Southampton Program Analysis Development Environment

STP Simple Theorem Prover

TCB trusted computing base

VC verification condition

VCG verification condition generator

VM virtual machine

Nomenclature

A_{SP}	Type of assignments
β	Type of basic blocks
$\bigcirc$	Denotes an arbitrary binary operator
$\bot$	Used here to represent an empty value, such as the result of calling a partial function with a value it does not have an actual result for or in general the `None` value of an optional type
$\bullet$	Indicates bitstring concatenation
$\equiv$	Indicates term equivalence; the term on the left may be replaced by the term on the right
$\bot_{\mathrm{E}}$	Indicates exceptional state
E_{SP}	Type of expressions
$\gg$	Performs unsigned right shift when used with word values
$\ll$	Performs left shift when used with word values
$\mathbb{B}$	Type of boolean values, True and False
$\mathbb{W}$	Type of 64-bit words
$\mathbb{N}$	Type of natural numbers
Φ	Type of branching conditions
ϕ	Denotes a generated invariant
$\prod$	Product of a sequence of terms; multiplication equivalent of $\sum$
$\subseteq$	Indicates subset relation

$\langle h, l \rangle w$ Indicates taking bits in word w from bit l to bit h using 0-indexing

SP Type of state parts (regions, flags, and registers)

A Type of instructions

L Type of instruction addresses in a program; a 64-bit word

S Type representing program state; an Isabelle record

Chapter 1

Introduction

Proving that a non-trivial program has no bugs is not an easy task. As technology continues to improve, software will continue to increase in complexity. Providing methods to ease the work of reasoning over programs is a necessity in the modern world. This is particularly important for programs that are intended for high-reliability applications, such as avionics, medical equipment, or other safety-critical systems. Formal verification allows reasoning over programs with a high degree of assurance.

This dissertation introduces the property of *memory preservation* as well as methodologies for formally verifying this property over real-world programs. Memory preservation expresses that the memory a program writes to is bounded by prespecified regions.

Memory preservation cannot be proven fully automatically as it is an undecidable property, mandating an *interactive* but *scalable* approach. It also must be proven at the assembly level as it relies on concrete memory layout.

Formal verification of software has been an active research field for decades. This dissertation aims to provide a formal verification method that is specifically tailored to memory preservation. This allows more automation and scalability. For example, we have shown that we can formally verify approximately 12 000 lines of assembly code obtained by decompiling binaries of the XEN hypervisor with minimal user interaction. To the best of our knowledge, there exists no current state-of-the-art method that specifically aims at formal verification of memory preservation.

1.1 Motivation

As a basic property, memory preservation has potential applications to security analyses, compositional reasoning, and even concurrency. These potential applications are described in more detail below.

1.1.1 Security

Unbounded memory usage can lead to vulnerabilities such as buffer overflows and data leakage. One example of such a vulnerability would be 2014's Heartbleed [63]. Heartbleed was caused by a lack of bounds checking on a string array requested as output as part of a "heartbeat" message. This, combined with a custom memory manager that also had no security protections against out-of-bounds memory accesses, lead to potential leakage of sensitive data such as passwords and encryption keys. Memory preservation could serve as a foundation for formal security analyses that could be used to expose vulnerabilities involving malicious writes.

Another important property that memory preservation could help with is *control-flow integrity* (CFI) [1]. CFI ensures that software execution follows a predetermined *control flow graph* (CFG) using static analysis and runtime checks. At a minimum, this requires proving that a program cannot overwrite its stack pointer or that a called function does not overwrite local variables of its caller. In other words, it must be proven that the memory writes of a program are confined to prespecified regions, which is exactly what memory preservation states. This can aid in avoiding return-oriented programming attacks without excessive runtime overhead.

The property of *noninterference* is also a useful one for security. On a high level, it states that a group of users using a certain set of commands *does not interfere* with another group of users if the the first group's actions have no effect on what the second group of users can see [59, 108]. On a functional level, that could be interpreted as a statement that a non-interfering function does not modify any memory that is accessed by the function not being interfered with. Memory preservation is specifically about showing that all memory outside of specific regions is not modified by the function or functions associated with those regions, so proving that the region sets for two functions are disjoint would essentially prove noninterference for those two functions.[1]

1.1.2 Composition

Scalability in verification is only feasible with composition; proofs of functional correctness or some other property over a large suite of software require decomposing that suite into manageable chunks. Separation logic provides a *frame rule* that supports such decomposition [75, 94, 104]. In words, the frame rule states that, if a program or program fragment can be confined to a certain part of a state, properties of that program or program fragment carry over when used as part of a larger system involving that state. Memory preservation allows for discharging the most involved part of the frame rule, at least in terms of individual assembly functions. That is, it shows that the memory preservation of those functions is constrained to specific regions in memory. This could then serve as a basis for a larger proof effort over multi-function assembly programs.

[1] A weaker property would be showing that one of the functions does not write to any of the memory regions read by the other, but that would actually be harder as we do not currently differentiate between regions that are read and written.

1.1.3 Concurrency

Reasoning over concurrent programs is complicated due to the potential interactions between threads. While there are ways of handling such interactions in a structured manner via kernel- or library-provided inter-process communication, one method commonly used for the sake of efficiency is *shared memory*. Shared memory, in the context of this work, refers to threads or processes sharing either a full memory space or portions of one (via memory mapping) that can be written to and read from freely by any thread or process with access to it. Usage of shared memory can result in *unintended* interactions between threads. Memory preservation could be adapted to show the absence of such interactions by proving that multiple threads only write to specifically-allowed regions of shared memory. Doing so would, of course, require a proper model of concurrency, which is out of scope of this dissertation.

1.2 Challenges

A fully-automated verification effort to prove memory preservation would be ideal, but it is not a feasible approach [97]. As per Rice's theorem [105], memory preservation is ultimately an undecidable property. One alternative would be to use *interactive theorem proving* (ITP), but that does not scale well either due to the amount of intricate user interaction involved. This dissertation proposes a *semi-automated* approach. It uses an ITP environment, but with code generation to generate as much of the proof code as possible.

1.3 Assembly-Level Verification

Properties that reason over the concrete memory used by a program, such as memory preservation, cannot be satisfactorily expressed on the source-code level. This is because even programs in a relatively low-level language like C have abstractions on memory for local variables and function calls. How and where that memory is allocated may be compiler, *application binary interface* (ABI), and *instruction set architecture* (ISA)-specific. It can even depend on what compiler options are in use, including the level of optimization. While one way of resolving that issue would be to choose a specific compiler and provide a formal analysis of how it arranges memory (or write a compiler to do so), that method places restrictions on the build process. Targeting assembly or machine code directly, as done in this dissertation, allows bypassing the build process, which also opens the door for verification of legacy code.

Example 1.1. As a further illustration, consider formulating a property that a function cannot overwrite its own return address. Doing so would require knowledge of the layout of the stack, including the values of the stack and frame pointers, thus making it an *assembly-level* property.

As a side benefit, targeting assembly means that there is no need to trust all the steps between

writing source code and obtaining a binary from it. Doing so reduces the *trusted computing base* (TCB) without needing to use a compiler that has been formally proven to maintain the semantics of the source code in the binaries it produces.

1.3.1 Challenges

The biggest challenge in assembly-level verification is the semantic gap between compiled and source code. Higher-level languages hide details of their implementation behind layers of abstraction, which makes it easier to reason about them on that level but makes it harder to formally show equivalence with the semantics of to lower abstraction levels. Meanwhile, assembly languages are close to direct interfaces with their corresponding ISAs, having minimal differences in semantics but not being easy to reason about directly.

As an example of the semantic gap, assembly code generally lacks the structured control flow found in languages on a higher level of extraction. Instead, all control flow on the assembly level is performed using conditional or unconditional branches, either to a predetermined location or to a calculated label.

A further example would be source code containing division operations being compiled to run on a processor that does not provide hardware division. Many *central processing unit*s (CPUs) for embedded systems lack support for hardware division as efficient division algorithms require a lot of circuitry. For such processors, runtime division must be calculated using an algorithm implemented in assembly rather than via a specific instruction.

Even the basic concept of numeric types is minimal on the assembly level, much less more abstract data types like lists or trees. While most ISAs do have different instructions for signed versus unsigned integer arithmetic, as well as distinct instructions for floating-point operations, individual values in memory have no type. They are merely lists of bytes starting at some address, and even the number of bytes and the address to read from or write to can be variable. A user could go as far as supplying the result of a floating-point computation as the address operand of an instruction that loads or stores memory. Historically, there have been computers that associated type information with memory locations in hardware [49, 50, 121], but we do not have that luxury on typical modern systems.

An additional issue with assembly, and the one most significant for this dissertation, lies in the simplicity of the user-exposed memory model. The vast majority of high-level, structured languages with scoping prevent function calls from accessing the local variables of other calls without significant effort or explicit notation, but the same is not true for assembly. An assembly instruction that operates on memory can refer to any address within range of its address operands even if it is not supposed to. Most modern ISAs do provide some form of memory protection, but those generally rely on runtime detection of invalid accesses and are often not fine-grained enough for reasoning about individual stack frames or local variables. Any verification effort that wishes to reason about low-level memory properties must provide its own abstractions and assumptions on layout.

1.3.2 Current Approaches to Assembly Verification

In 2014, Goel [57] and Goel et al. [58] produced formal semantics for most user-mode x86-64 instructions as well as for commonly-used system calls. That work allows mechanized reasoning over compiled programs in the ACL2 theorem prover [71].

Soon after, Tan et al. [120] introduced a logic framework called AUSPICE for automated verification of safety properties on the assembly level. AUSPICE took six hours to execute on 533 instructions, but was applicable to unmodified code. Our methodology in Chapter 6 is also applicable to unmodified code, as long as that code is assembly.

More recently, Baumann et al. [5] provided an ARMv8-based hypervisor that was formally verified on the machine code level to ensure isolation of guest *operating system*s (OSes). That work was based on an earlier one for an ARMv7 separation kernel, PROSPER [36, 37].

Additionally, earlier this year, Fromherz et al. [54] embedded a subset of the x86-64 ISA in the functional, verification-oriented language F* [46]. This was done in order to perform a proof of correctness over the commonly-used cryptographic routine AES-GCM. Their usage of a *verification condition generator* (VCG) is similar to ours in Chapter 6, but ours did not need to be separately formally verified as we implemented it with proven-true Hoare rules.

1.4 Contributions

This dissertation presents two formal approaches to per-function verification of the assembly-level property we call memory preservation: *control-flow-driven* verification and *syntax-driven* verification. Both approaches use some form of control-flow analysis over functions in x86-64 assembly to generate incomplete proofs. Those proofs are then loaded into the interactive theorem prover Isabelle/HOL and completed there. The proof strategies for both approaches involve *symbolic execution* of the underlying assembly code [72], albeit in different ways.

The main differences between the two approaches lie in their degrees of automation, the strengths of their invariants, and how they perform symbolic execution. The first approach, control-flow-driven verification, requires significantly more user user input but has the potential for much stronger invariants. Meanwhile, the second approach, syntax-driven verification, has a significantly higher level of proof automation via the generation of *formal memory usage certificate*s (FMUCs) but is not as suited for stronger invariant production. Symbolic execution is also more efficient in the control-flow-driven approach as it more closely follows the structure of the function's CFG. In contrast, the syntax-driven approach must deal with operating on a restricted set of control flow constructs, which can result in extra symbolic execution.

1.4.1 Control-Flow-Driven Verification

This methodology for verification of memory preservation relies on treating function bodies as CFGs with basic blocks as the nodes, much as compilers do when performing their analyses. In order to reason about the CFGs, they are annotated with predicates on state at specific locations, between which the program will be symbolically executed. While it is possible to reason about full functional correctness with this methodology, doing so takes a significant amount of effort due to the very low level of abstraction assembly provides, even with proven-correct formal simplification rules in Isabelle. Because of this, we focused on the aforementioned property of memory preservation.

In our model, memory usage is formulated as a set of *regions* that start at some address and have a specific size in bytes. We do not currently differentiate between regions for writes and regions for reads, though doing so is a possibility in the future. Proving memory preservation requires performing symbolic execution on the underlying assembly instructions and showing that no regions beyond those needed to complete the proof are modified.

In order to reason about that memory usage so we can prove memory preservation in a theorem prover, the structure of the proof must be extracted from the assembly programs. For that purpose, our code generation tool for this work produces the skeleton of a proof based on the control flow of the analyzed programs. This is achieved using off-the-shelf tools.

That proof skeleon specifies where the program should be annotated and provides some initial conditions based on register values. It also provides the proof steps to properly perform symbolic execution and starts the user off with a basic set of regions determined from variables in the stack frame. The two steps remaining, however, are up to the user. Those steps are formulating any remaining memory regions to successfully complete symbolic execution and fleshing out the annotations on state so that the symbolic execution of later blocks can continue from that of earlier ones.

The control-flow-driven methodology was applied to 63 functions extracted from the HermitCore [78] unikernel library [82], covering 760 *source lines of code* (SLOC) or over 2379 assembly instructions. Of those functions, 18 had loops and 33 had subcalls. Optimized variants were also verified for 12 of the functions involved, resulting in 75 functions verified. There was even one function that featured recursion, which turned out to be the most challenging function to handle. Other than the recursive function, the most challenging ones to handle were the ones with loops. Formulating annotations that must hold for all loop iterations is not easy when a significant amount of memory operations are performed.

The closest related work to this, that of Matthews et al. [85], resulted in the verification of only 20 functions, with 631 assembly-level instructions in total. That is only 26.67 % of the functions, or under 26.5 % of the instructions, that we verified here. On top of that, the ISAs they worked with are not as low-level as the x86-64 ISA. While they verified functional correctness instead of a weaker property like memory preservation, they also specifically reduced the complexity of the most complicated set of functions they verified by using a simple `xor` cipher instead of a proper block cipher.

1.4.2 Syntax-Driven Verification

Taking our experiences from the control-flow-driven verification work into account, we chose a slightly different path for the second verification work presented in this dissertation. This approach focuses on relating symbolically-executed basic blocks with a syntactic representation of program control flow. It also involves significantly more information generation than the previously-discussed approach.

Abstracting away from the concrete control flow to a more structured syntax increases the capacity for automation as it allows for the development of a set of *Hoare rules* over the syntactic control flow [65]. By developing and using a set of such formal rules, we were able to restrict symbolic execution to the level of individual basic blocks and then use those rules to do the rest of the work. This greatly simplified our proof strategies for proving memory preservation.

The change in methodology alone would not have been enough, however. As stated, we also generate much more information. That additional information consists of the full set of memory regions for each basic block, the corresponding *memory region relations* (MRRs), and the block's preconditions and postconditions. Having that information generated for them greatly reduces the work an end user must put in compared to our initial approach.

Unlike the previous work, this one was applied to assembly obtained by running `objdump` on three *unmodified* binaries resulting from the Xen Project hypervisor build process [29]. Of the 352 functions present in those binaries, 251 or 71 % were verified. Ultimately, over 12 252 optimized instructions were covered with only 1047 manual lines of proof required. That is an approximate ratio of one manual line of proof for every 12 instructions handled, or an average of 16 manual lines of proof for every loop handled, of which there were 65.

To the best of my knowledge, this is the first work to achieve that degree of coverage for optimized x86-64 binaries produced by production code. While Tan et al. [120] produced a fully-automated methodology for binary analysis, it was much slower than our approach here, meaning they would take longer to cover the same amount of functions even though they had more automation. Under normal circumstances, this approach can complete the proofs for two functions with a total of 97 assembly instructions in less than ten minutes. That is 9.7 insts/min compared to 1.48 insts/min for AUSPICE, 6.55 times as fast. We did have some functions that took an overly long period of time due to the suboptimality of syntactic control flow with respect to minimizing symbolic execution, but those were atypical.

1.5 Organization of Dissertation

Following this introduction in Chapter 2 is a review of tools and work related to the field of assembly-level verification and software correctness in general. Domain-specific information necessary to understand the work and terminology can then be found in Chapter 3. For an in-depth exploration of the basis for the symbolic execution engines and formal memory

reasoning used by the contributions of this work, see Chapter 4. After that, the control-flow-driven approach to verification of memory preservation mentioned above is presented in Chapter 5 while the syntax-driven approach is presented in Chapter 6. Finally, my dissertation wraps up in Chapter 7, which includes a discussion of possible post-preliminary exam work.

Chapter 2

Related Work

Verification of assembly has been an active field of research for decades. This chapter covers some of that history.

Up first in Section 2.1 are some previous formal verification efforts that target assembly. Following that is work on a lower level, verification of the hardware that runs machine code, in Section 2.2. After that is work in which assembly verification played a role in a larger verification context, Section 2.3, and then verified compilation and static analysis tools are discussed in Sections 2.4 and 2.5. Table 2.1 provides an overview of the assembly, hardware, and integrated projects. It also includes the works presented in this dissertation.

2.1 Assembly-Level Verification

Clutterbuck and Carré [31] performed formal verification of assembly programs using SPACE-8080, a verifiable, analyzable subset of the Intel 8080 ISA. Their work used *the Southampton Program Analysis Development Environment* (SPADE) [25], a set of software tools for "the efficient development, analysis, and formal verification of high-integrity software". SPADE provides a *functional description language* (FDL) for modeling programs in order to analyze and formally verify them using a VCG, proof checker, and symbolic interpreter. They used automatic translation to model their SPACE-8080 program in FDL, and their verification methodology used the same kind of annotated control-flow analysis as our control-flow-driven approach presented in Chapter 5 does, with additional assertions on state to avoid errors and stronger conditions in order to prove functinoa correctness. They also provided *rewrite rules* that described the semantics of the formal models in SPADE. Unlike the work detailed in Chapter 5, however, they only covered a single 33-instruction function with the verification methodology they presented.

Another usage of SPADE for a more in-depth verification of assembly was in the correctness proof of fuel control code for a Rolls-Royce jet engine [95]. Once again, it involved formulating a verification-friendly model of the Z8002 ISA in SPADE, the development of a Prolog translator from Z8002 assembly to FDL, and the formalization of written specifications into

proper pre- and postconditions in SPADE. SPADE's proof checker was then used to validate the correctness of the translated control code. While they assumedly covered many more instructions than the previous SPADE work did, the authors did not go into detail on the amount of work that was actually done. The only number given was that the specifications for about 10 % of the code modules under test were clarified and one module received a code fix to improve its performance.

Similarly, Yu and Boyer [20, 135] presented operational semantics and mechanized reasoning for approximately 80 % of the instructions of the MC68020 microprocessor, over 85. They implemented those semantics and mechanized their approach in *the Boyer-Moore theorem prover* (Nqthm) [19], a precursor to the theorem prover ACL2 [71]. They then applied their mechanized reasoning to check functional correctness for a binary search, quicksort, a standard C string library, and others. These early efforts required significant interaction, as Yu and Boyer required over 19 000 lines of manually written proof to verify approximately 900 assembly instructions. Compare this to the 1047 lines of manual proof required to prove memory preservation over 12 252 assembly instructions. Admittedly, they were verifying stronger properties, which would greatly increase the amount of work required for verification, but even then it's a significant difference.

Following that, Matthews et al. [85] targeted a simple machine model called TINY as well as the M5 operational model of Java virtual machine bitcode. Their approach for functional correctness, implemented in ACL2, utilized symbolic execution of operational semantics over code annotated with manually written invariants in order to generate *verification condition*s (VCs) and then discharge them. They even supported compositionality by verifying subcalls individually. Both of the assembly-style languages they tested with feature a stack for handling scratch variables rather than a register file as x86, ARM, and most other mainstream ISAs do. The case studies they verified were an implementation of the Fibonacci sequence, a factorial function, and functions for CBC-mode encryption and decryption. In total, they covered 631 assembly instructions, less than that handled by either of the methodologies presented in this dissertation. Of course, they were targeting a stronger property than either of those works, but they also did not perform any significant work to automate their approach. All in all, however, this work is the closest to the control-flow-driven approach presented in Chapter 5 of any of the works presented in this chapter, as they even implemented a version in Isabelle. Their Isabelle version did not support compositionality, however.

Additionally, Goel et al. presented an approach for modeling and verifying non-deterministic programs on the binary level [57, 58]. As with Matthews et al. [85], their work was implemented in ACL2. In addition to formulating the semantics of most user-mode x86 instructions, they provided semantics for common system calls. System call semantics increase the spread of programs that can be fully verified. Their work was applied to multiple small case studies, including a word count program and two kernel-mode memory copying examples.

Ultimately, the main difference between the above-mentioned existing approaches and the methodologies presented in this dissertation lies in the degree of automation. As stated previously, ITP over semantics of assembly instructions does not scale under normal circumstances. This is again due to the amount of intricate user interaction required.

Fully automated approaches to formal verification, however, do not necessarily scale either. The recent automated approach AUSPICE provided by Tan et al. [120] takes about 6 hours to run on a 533-instruction string search algorithm. This is despite the fact that, similar to our approaches, they were targeting weaker safety properties rather than going for functional correctness. As another similarity to our approach in Chapter 6, they too used a full set of Hoare rules in their analysis.

Though it is not a verification methodology by itself, there is also *decompilation into logic* (DiL) [89, 90]. Developed by Myreen et al. in the HOL4 theorem prover [118], DiL uses operational semantics of machine code to lift programs into a functional form. That functional form can then be used in a Hoare logic framework for program analysis [88]. It formally covers the gap between machine code and a *higher-order logic* (HOL) model and allows for verification of properties in a theorem prover that utilizes that model. DiL has been used for both ARM and x86 ISA machine models and applied to various large examples, including benchmarks such as a garbage collector as well as the Skein hash function. It has even been used as a component in a binary-level verification methodology over the seL4 microkernel [113].

Also, Feng et al. [47, 48] presented stack abstractions for modular verification of assembly code in the Coq theorem prover [30]. Their work allows for integration of various proof-carrying code systems [91]. As with the work presented in Chapter 6, it utilizes a Hoare-style framework for its verification. The authors applied their work to multiple example functions, such as two factorial implementations as well as `setjmp` and `longjmp`. In contrast to the approach presented in Chapter 6, though not that in Chapter 5, manual annotations are required to provide information regarding invariants and memory layout.

Schlich [110] worked on the development of a model checker for analysis of microcontroller assembly, [mc]square. Implemented in Java and supporting several microcontroller ISAs, it uses multiple methods of state space reduction in order to avoid state space explosion as much as possible. While it was applied to multiple case studies, some of those case studies were only to analyze the effectiveness of the various abstraction techniques used for state space reduction.

The most relevant case study to this dissertation was its application to software compiled for an automotive microcontroller [111]. The three programs they focused on for their case study were designed to record speed measurements from sensors on four wheels, calculate the actual speed, and then transmit it over a *Controller Area Network* (CAN) bus.[1] Some of the programs required simplification to be checkable, so for consistency they applied, or at least attempted to apply, the same simplifications to all three programs. They did what they could to remove sends over the CAN bus and tried to focus on the speed signal from just one wheel rather than all four. Ultimately, they were able to reason about all three programs and prove both functional and non-functional properties of those programs. On average they covered around 700 lines of C or 2666 assembly instructions. In terms of the works presented in this dissertation, the case study on HermitCore in Section 5.5 did involve isolating the functions under test before compiling them and covered less instructions. By contrast, the

[1] A CAN bus is a standard bus for electronic communications in automotive applications.

analyzed Xen functions in Section 6.4 were handled without any modification whatsoever to the Xen build process and covered even more instructions.

Brauer et al. [21] intially performed static analysis of stack bounds for the Atmel ATmega16 and Intel MCS-51 microcontrollers in order to verify a lack of stack overflows. Their work was applied to eight programs, four compiled for each microcontroller. The functions compiled for the ATmega16 ISA had previously been used to evaluate the effectiveness of [MC]SQUARE. They then embedded their static analysis in [MC]SQUARE as a means to improve the accuracy of dead variable reduction, which [MC]SQUARE uses for state space reduction. While this stack safety approach is similar in focus to the memory preservation works I present here, the scope is much smaller. Their focus was specifically on stack memory rather than memory in general.

Earlier this year, Fromherz et al. [54] embedded a subset of the x86-64 ISA in the functional, verification-oriented language F* [46]. This was done in order to prove commonly-used crytographic routines that mix C with assembly for performance reasons are secure from information leakage. The cryptographic routines they applied their work to were Poly1305-*Advanced Encryption Standard* (AES) [6] and AES-Galois/Counter Mode [43]. Their aim was to use F*'s dependent type system to run a verified VCG during type checking, with the generated VCs then being supplied to a *satisfiability modulo theories* (SMT) solver. The conditions on state to generate the VCs were expressed using Vale, a language for assembly verification [17]. This was done in the style of *proof by reflection* [7]. The VCG itself, QuickCode, was formally proven sound in F* as well. They measured performance of their verified algorithms, as one of them could be transpiled to C. They also measured the performance of various versions of their VCG, and found that optimizing the SMT queries did improve performance significantly.

Unlike our works in Chapters 5 and 6, the authors of this paper had to do extra reasoning to ensure the C and assembly code models were interoperable. As we operated directly on full assembly, we did not have to worry about that kind of interfacing. Both this work and the work presented in Chapter 6 used an explicit VCG, but ours was proven correct by its Isabelle/HOL definitions; we did not have to perform any additional work to ensure correctness of the methodology. Of course, as we implemented ours in an interactive theorem prover, we could guide the VC generation and discharging as needed.

2.2 Hardware Verification

On a level lower than the assembly code level, or even the machine code level, is the work done by hardware designers and testers to verify the products that run those codes. Quite often this is done with model checkers.

For example, ARM recently released several of their ISAs in a machine-parsable, executable format called *ARM Specification Language* (ASL) [124], a result that took five years to develop. While not directly verifiable, the documents were automatically translated into a verifiable form for use with a verification tool called ISA-Formal [103]. ISA-Formal is intended

to verify processor pipeline control and has been successfully used for that purpose on multiple versions of the ARM ISA. At its core, it uses bounded model checking for instruction sequence exploration. They accomplished this by developing a translator from ASL to the subset of System-Verilog that can be handled by commercial Verilog model checkers.

Not all of the components they worked with could be handled by the model checkers, however; they restricted its usage to ISA analysis. Alternative verification techniques were used for the components that the model checkers could not handle, such as the floating-point units and memory system. These alternative techniques involved raising the abstraction level and/or reducing the explored state space, as their goal was not necessarily to detect all bugs in those specific units. Instead, they wanted to check the correctness of the logic connecting them to other processor elements, which they were successful at.

More recently, the Sail ISA specification language, which supports automatic generation of emulation code and of proof definitions for Isabelle, HOL4, and in some cases Coq, has been used to provide rigorous semantic models of various *reduced instruction set computer* (RISC) ISAs [2, 42]. The ISAs in question are ARMv8.3-A, CHERI-MIPS, and RISC-V. It was also used to produce a proof of correctness for a model of ARMv8-A address translation in Isabelle.

2.3 Integrated Assembly-Level Verification Efforts

A major verification effort based on decompilation into logic was the verification of the seL4 kernel [73, 74]. The seL4 project provides a microkernel written in formally proven correct C code. The tool AutoCorres is used for C code verification [61]. Sewell, Myreen, and Klein [113] verified a *refinement relation* between the C source code and corresponding non-optimized and `O2`-optimized ARM binaries. The major differences with respect to our work is that our methodology targets existing production code, instead of code written with verification in mind. For example, the seL4 source code does not allow taking the addresses of stack variables (such as in Fig. 6.7a): their approach requires a static separation of stack and heap instead.

Shi et al. [114] formally verified a *real-time operating system* (RTOS) for automotive use called ORIENTAIS. Part of their approach involved source-level verification using a combination of Hoare logic and abstract communicating sequential processes model analysis [66]. Binary verification was done by lifting the RTOS binary to xBIL, a related hardware verification language [115]. They translated requirements from the OSEK automotive industry standard to source code annotations. Ultimately, they proved properties such as deadlock-freedom, memory access safety, and bounded response time in the presence of interrupts [116]. A similarity with our work was the usage of Hoare logic, while the difference is that we performed verification solely on the assembly level and with a more complex ISA. We ultimately handled over 14 631 x86-64 instructions compared to their 60. While they did handle 8000 lines of C as well, that is still a higher-level language than x86-64 assembly.

Targeting a similar case study as Chapter 6, Dam et al. [36] and Dam, Guanciale, and Nemati

[37] formally verified a tiny ARMv7 *separation kernel*, PROSPER, at the assembly level. Separation kernels are similar to hypervisors, providing isolation for individual components of a system and ensuring only those components that are allowed to communicate do [107]. Their methodology integrated HOL4 with *the Binary Analysis Platform* (BAP) [22]. BAP utilizes a custom intermediate language that provides an architecture-agnostic representation of machine instructions and their side effects. First, the formal model of the ARM ISA provided by Fox and Myreen [53] was used in an HOL4 tool to translate the ARM binary into BAP's intermediate language. Following that, the SMT solver *Simple Theorem Prover* (STP) [55] was used to determine the targets of indirect branches and to perform weakest-precondition computation with Hoare triples to verify the user contracts. While the approach was generally automated, user input was still required to describe the contracts the separation kernel was verified against. An extension to the work is found in the HAPSOC project by Baumann et al. [5], who did a similar proof for the ARMv8-A model provided by Fox [52].

Finally, Bevier et al. [11] presented a systems approach to software verification that targeted correctness all the way down to the hardware level. All of their work was implemented in Nqthm. Hunt [68] developed a general-purpose, 32-bit microprocessor, FM8502, and proved that its gate-level specification was an implementation of its formal ISA. Bevier [8–10] designed a small OS kernel, Kit, and proved that it implemented "a fixed number of conceptually distributed communicating processes" along with a set of typical kernel services and some security properties. He did not prove that it could run on an FM8502, however; it was executed on a more abstract model instead. Young [134] designed and proved the correctness of a code generator, a major compiler component, for a subset of the Gypsy 2.05 programming language [60]. That code generator's output was the verified, high-level assembly language Piton [87]. Moore [86] then proved the correctness of that language's FM8502 implementation.

2.4 Verified Compilation

In contrast to directly verifying machine or assembly code, one can verify source code and then use *verified compilation*. Verified compilation establishes that the semantics of the output of the verified compiler is equivalent to the semantics of the input, so a program that has previously been verified correct is verified to still be correct after compilation.

The CompCert project is one such verified compiler, used by the seL4 project to reduce its TCB [73]. It is written in a subset of C called Clight [13, 79], though it itself is able to handle most components of the C99 standard. The main difference between C and Clight is that Clight is *pure*; none of its operations have side effects. It also provides only one loop structure, an infinite loop that must be broken out of to exit. Clight has been mechanized in the Coq theorem prover with established big-step operational semantics, making it a useful subset of C for program verification work in Coq.

The full proof of *semantic preservation*, as it is called in the CompCert documentation, is based off of proofs of semantic preservation for each step in CompCert's compilation process,

of which there are twenty. On top of that, it has eleven different intermediate languages for those steps, all of which had to be proven semantically equivalent.

Another example of verified compilation is CakeML [77]. It utilizes a (substantial) subset of Standard ML modeled with big-step operational semantics in HOL. Its main compiler frontend is designed to take ML-like HOL functions and translate them to a CakeML abstract syntax tree, which is then translated into machine code using a verified backend. The compiler itself is bootstrapped, meaning it can compile itself in HOL. It also provides support for using Hoare logic to perform post-hoc verification using a version of the CFML verification framework [26, 27, 62].

More recently, Chen et al. [28] produced a compositional framework for the development of certified, *interruptible* OS kernels that use device drivers. This was previously a challenge due to the non-local and asynchronous behavior of hardware interrupts. Their approach uses a general certified device model with multiple instantiations and provides an effective model of device interrupts. The verification was done in Coq with the kernel written in Clight. Once verification was complete, the source code was compiled using a modified version of CompCert to ensure the semantics were maintained. They showed the value of their work by taking a preexisting, non-interruptible, mostly-verified kernel, mCertiKOS [32], and extending it to work in their framework along with a couple of device drivers. In order to deal with devices running in parallel with the CPU, device interaction, drivers that are written in a mix of assembly and C, and properly integrating the correctness proofs for individual components of the system under test, they did the following. First, they designed their system architecture such that each device driver is given its own logical CPU, running independently from the core of the kernel. Then they designed an abstract model of interrupts based on existing hardware implementations. The correctness of the system as a whole was shown by starting from a base machine model and proving a refinement relation with each layer of abstraction placed on top of it. As they started off with a mostly-verified kernel, it is likely they would have had more difficulty if they had started with a kernel not explicitly designed to be verified. While we did isolate functions for verification in Chapter 5, we did not do so for Chapter 6.

2.5 Non-Formal Static Analysis

Non-formal static analysis of binary code for the detection of bugs has also been an active research field for decades [22, 76, 128]. This section covers some of the projects in that field from the past fifteen years.

The BitBlaze project [12, 119] provides a tool called Vine which lifts x86 instructions to its own intermediate language for assembly in order to perform analyses on a higher level. That language is fully-featured and can itself be compiled back down to assembly if so desired. In terms of analyses, Vine can construct CFGs for the lifted programs, perform compiler-style dataflow analysis, generate dependency graphs, and slice programs [123, 130]. Though Vine itself is not formally verified, it does support interfacing with the aforementioned SMT solver STP as well as CVC Lite [3] and CVC3 [4]. This allows for formal verification.

Meanwhile, the tool Infer [24], now developed at Facebook, provides in-depth static analysis of LLVM code to detect bugs in programs written in a variety of languages. It utilizes separation logic [104] and bi-abduction to perform its analyses in an automated fashion. It is designed to be integrated into compiler toolchains in order to provide immediate feedback even in continuous integration scenarios. For all the languages it handles, it checks for null pointer accesses and other null-related issues as well as checking for language-specific concurrency issues. For C-style languages with a lower level of abstraction, it also looks for memory leaks, performs linting for violations of coding conventions, and checks for calls to mobile device application programming interfaces that are not available for the target device OS. In Android code and Java in general, it ensures annotations can be reached, looks out for mutability issues, and checks for resource leaks.

The static analysis tool FindBugs, written for Java code, takes a bit of a different approach from those other two [67]. Rather than performing control flow or dataflow analyses, it searches Java bytecode for common (bad practice) code idioms in order to detect likely bugs. Much like Infer, some of the common errors it highlights include null pointer dereferences, objects that compare equal not having equal hash codes, and inconsistent synchronization. It even provides a bug database that can be used to keep track of its warnings throughout multiple iterations of development.

A somewhat older tool, Splint [45] detects buffer overflows and similar potential security flaws in C code. Splint relies on annotated preconditions to derive postconditions based on the syntactic structure of the code. While their methodology is very similar to the formal technique of Hoare logic, described later on in this dissertation (Section 3.1.2), they used heuristics for loop analysis rather than proper invariants and thus could potentially miss bugs. The annotations Splint uses are memory-focused, such as allowing users to specify that certain variables should never be null or providing an emulation of `unique_ptr` functionality. It also does constraint analysis and issues warnings when it encounters an expression it cannot determine will not result in a bug, as well as checking for format string vulnerabilities.

The main difference between these static analysis tools and formal verification is that these tools are generally highly suited to finding bugs but are not able to prove their absence. This is due to a reliance on efficient but incomplete techniques, such as depth-bounded searches.

2.6 Summary

This section covered some of the work related to that presented in this dissertation. Previous assembly- and hardware-level formal verification efforts, verification efforts containing assembly or binary analysis components, verified compilation, and non-formal static analysis tools were all discussed.

Notably, while multiple assembly-level verification efforts presented in this chapter achieved more coverage than the over 2379 instructions achieved by the work in Chapter 5, none

Table 2.1: Overview of related assembly verification and other work

Work	Target	Approach	Applications	Verified code
Clutterbuck,Carré	SPACE-8080	ITP+VCG	Example func	33 insts
O'Neill et al.	Z8002	ITP+VCG	Jet engine code	
Yu & Boyer	MC68020	ITP	String funcs	863 insts
Matthews et al.	Tiny/JVM	ITP+VCG	CBC enc/dec	631 insts
Goel et al.	x86-64	ITP	word-count	186 insts
Tan et al.	ARMv7	ATP	String search	983 insts
Myreen et al.	ARM/x86	DiL	seL4	9500 SLOC
Feng et al.	MIPS-like	ITP	Example funcs	
Schlich et al.	ATmega16	MC	Auto funcs	Around 8k insts
Brauer et al.	ATmega16 Intel MCS-51	SA+MC	Example progs	2630 SLOC 935 SLOC
Fromherz et al.	C/x86-64	ATP+VCG	AES funcs	
Chapter 5	x86-64	ITP+VCG	HermitCore	2379+ insts
Chapter 6	x86-64	CG,ITP,VCG	Xen	12 252 insts
Reid et al.	ASL	MC+others	ARM ISAs	2 209 191+ SLOC
Sewell et al.	C	TV+DiL	seL4	9500 SLOC
Shi et al.	C/ARM9	ATP+MC	ORIENTAIS	8k SLOC, 60 insts
Dam et al.	ARMv7	ATP+UC	PROSPER	3000 insts
Baumann et al.	ARMv8-A	ATP+UC	HAPSOC	8000 SLOC
Bevier et al.	PDP-11-like	ITP+TV	Full system	3k+ SLOC/insts

VCG = Verification Condition Generation
SLOC = Source Lines of Code
UC = User Contracts
TV = Translation Validation
SA = Static Analysis
DiL = Decompilation into Logic
ATP = Automated Theorem Proving
CG = Certificate Generation
MC = Model Checking
ASL = ARM Specification Language

appear to have achieved the 12 252 verified instructions covered in Chapter 6 except the work produced by ARM employees.

Chapter 3

Background

This part of my dissertation provides domain-specific information necessary to understand the work presented in it. It is grouped into the two main categories of formal methods (Section 3.1) and assembly language (Section 3.2).

3.1 Formal Methods

To quote Butler [23],

> "Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.

This dissertation comes under the *verification* component of that.

3.1.1 Symbolic Execution

At its most basic, *symbolic execution* refers to executing a program with a set of symbolic inputs rather than concrete values [72]. Based on the semantics of the program, the execution may end up taking multiple paths; it could potentially be an infinite number if there are loops involved.

In this work, the individual steps of symbolic execution are implemented as *rewrite rules* over the state that derive their representation from Applying those rules in sequence to each step or instruction of a program allows aggregation of the individual state changes involved in the execution.

3.1.2 Hoare Logic

A form of *axiomatic semantics*, Hoare logic [65, 88] describes the behavior of a program with a *Hoare triple*.

Definition 3.1 (Basic Hoare triple)**.** Given predicates on state P and Q and a program C, $\{P\}C\{Q\}$ asserts that executing C on a state where P holds will result in a state where Q holds (as long as C terminates, that is).

In a more formal form and using $C(\sigma)$ to represent the result of executing program C from initial state σ, we have:

$$\{P\}C\{Q\} \equiv P(\sigma) \wedge C(\sigma) \neq \bot_{\text{NT}} \longrightarrow Q(C(\sigma)).$$

To prove that triple, deductive reasoning with *Hoare rules* formally derived from the structure of whatever programming language is in use would be used to syntactically decompose the program into its constituent behaviors. If any loops were to be present, the loop rule would require an additional condition that would hold on every iteration, a *loop invariant*, for each loop.

As Hoare logic was heavily inspired by Floyd's flowcharts, it is sometimes referred to as *Floyd-Hoare logic*.

Verification Condition Generation

In the context of this dissertation, a VCG is the tool that applies Hoare rules (for Chapter 6) or performs symbolic execution (for Chapter 5) in order to obtain a set of proof subgoals that must be proven true for the full proof to succeed. These goals are the VCs.

3.1.3 Theorem Proving

Isabelle and HOL

The theorem prover utilized in this work was Isabelle 2018 [93]. It is a generic proof assistant with a flexible, extensible syntactic framework. Isabelle also utilizes a powerful proof language known as intelligible semi-automated reasoning [131] Its most-commonly-used logic is HOL; when used with that logic, it is referred to as Isabelle/HOL.

HOL-Word We made heavy use of Isabelle/HOL's Word library [40] for the work presented in this dissertation. That library provides a limited-precision integer type, `'a word`, where `'a` is the number of bits in the integer. Various operations are provided for manipulation of and arithmetic involving formal words, including bit indexing, bit shifting, setting specific bits, and signed and unsigned arithmetic. Operators for inequality are also included, as well as operations for converting between word sizes.

Eisbach This simple but powerful language for declaring custom proof methods [84] is used in the verification methodologies for both works presented in this dissertation, though Chapter 6 used more of its features. Internally, it relies heavily on the standard method

Table 3.1: Method expressions

Syntax	Name	Behavior
`a, b`	Sequential composition	Apply `a`, then `b`
`a; b`	Structural composition	Apply `a` to the first subgoal, then apply `b` to just the goals produced by `a`
`a | b`	Fallthrough	Try applying `a` and then apply `b` if `a` fails
`a?`	Attempt	Try to apply `a`, leaving the goal alone if it fails
`a+`	Repeat at least once	Will repeat until `a` fails or no subgoals remain
`a[n]`	Subgoal restriction	Apply `a` to the first n subgoals, defaulting to 1

expression syntax [132], described in Table 3.1. The precedence of the syntactic elements involved, from low to high, is `| ; , [] + ?`. Parentheses can be used to modify the precedence, as with regular mathematical expressions.

Creating a standalone method is as simple as:

```
method a =
  b, c
```

Then `a` can be used in a proof the same as any other method. To make a method that can have theorems supplied to it, you use the `uses` keyword:

```
method a uses t =
  (simp add: t), c
```

The method could now be called like `apply (a t: thm1 thm2)` and it would supply the two theorems to `simp`. You can also create methods that take terms as input:

```
method a for L :: 'a list =
  b[where l=L]
```

And then call it like `apply (a <[1, 2]>)`.

3.2 Assembly Language

No modern high-level programming language is ever executed directly on hardware. Instead, they are *compiled*, either before execution (ahead-of-time compilation) or at runtime (just-in-time compilation), to low-level *machine code*. That machine code is specified by the ISA of the CPU in use. One step above machine code is assembly language, which is a textual representation of the binary code to execute, with each instruction and its operands being represented by human-readable mnemonics. There are usually some additional abstractions,

3.2.1 The x86-64 Instruction Set Architecture

An ISA is the specification of the visible behavior of a processor. They have long been published as human-readable documents [18, 69], though ARM recently released several of their ISAs in a machine-parsable, executable format [124].

An ISA can specify many things about the processor they describe. The data types supported by the processor are included in that, such as integers and floating-point values of specific sizes. The processor state is another major one, including how much main memory and registers are available; maybe even details of the cache(s) if it's an architecture where caching (on certain levels) is a manual process, such as is traditional for graphics processing units. The semantics of the architecture are another, such as the memory consistency model (complicated enough for x86-64 that formalizing it was a challenge [98, 99, 112]) and the addressing modes. The actual set of instructions supported by the ISA and how it communicates with external devices are also important.

Historically, there have been two main types of ISAs: *complex instruction set computer*s (CISCs) and RISCs [70]. CISCs came first, featuring complex ISAs with multiple addressing modes and many variable-length instructions with in-depth behavior. RISCs were a response to the growing complexity of CISC designs, providing a set of simple instructions that, other than loads and stores for memory access, only operate on a large register file. The main push for RISCs was actually a desire to reduce the complexity of implementations. RISC instructions being generally simpler than their CISC equivalents means less circuitry is required to implement them, which reduces die size/chip surface area and allows for increases in clock speed. This also allows the instructions themselves to complete faster, with most RISC instructions finishing in one clock cycle compared to the many many-cycle instructions of CISCs.

Currently, RISCs are used more often for systems that require low-power operation while CISCs are used more for high-performance applications. In modern times, however, differentiating between the two types of architectures due to power or performance concerns is no longer as relevant as it used to be. For current designs, the differences in performance and power have more to do with implementation than ISA [14].

In the end, this dissertation used the x86-64 ISA as it is a widely-used architecture that has had formal semantics derived for most of its instructions in previous works [64, 106]. It is the 64-bit, mostly-backwards-compatible successor to the x86 ISA, also known as x64, x86_64, AMD64 (for AMD chips), and Intel 64 (for Intel chips). The x86 family of processors originated in 1978 with the 16-bit Intel 8086, whose design is still reflected in that of modern x86 ISAs. Many of the instructions are the same, though extended over the years to work with larger operands. There are also many new instructions that have been added over the years.

The rest of this section describes features and properties of x86-64 that are relevant to this dissertation.

State

For the purposes of this dissertation, the important components of x86-64 state are the memory model and the registers available for use.

Memory Addressing in x86-64 takes the form of Eq. (3.1).

$$a = \left\{ \begin{array}{l} \texttt{fs}: \\ \texttt{gs}: \end{array} \begin{bmatrix} \vdots \\ \texttt{GPR} \\ \vdots \end{bmatrix} + \left(\begin{bmatrix} \vdots \\ \texttt{GPR} \\ \vdots \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \right) \right\} + \text{displacement} \tag{3.1}$$

$$\texttt{rip}$$

In this ISA, while addresses are all 64 bits long, only 52 of those bits are actually usable for physical addressing. Additionally, while it provides *virtual memory* so that every process gets its own address space, hiding the details of its memory management and paging from user-mode programs, that virtual memory only gets 48 bits for addressing; the rest must be sign-extended from the *most significant bit* (MSB) of that 48-bit value or else the address will be rejected.

While the actual instructions have no problems accepting 64-bit address operands, the issue lies in how paging is implemented and the usage of high address bits as storage for metadata. Allowing a full 64-bit address space would require additional storage space for the information that is currently stored in those bits. It is not impossible that it will happen eventually, however.

For now, the works in this dissertation make a simplifying assumption that all 64 bits are available for addressing. Properly modeling the 48-bit restriction would require additional word arithmetic, though would not be infeasible. We also do not deal with memory consistency, as the symbolic execution engine in Chapter 4 does not implement out-of-order execution or any similar optimizations.

Registers The standard registers in x86-64 (Table 3.2) are 64 bits in size, but their lower bits can be referenced alone if need be. If you write to one of the available 32-bit registers, it will zero out the upper 32 bits, but if you write to a 16- or 8-bit register, the rest of the contents will be preserved.

Flags

Stored in the `rflags` register on x86-64, these bits (Table 3.3) indicate, and in some cases control, processor state. `CF`, `OF`, `PF`, `SF`, and `ZF` are all used for specific conditional branches.

Table 3.2: The x86-64 registers (excluding SIMD)

Purpose	64 bit	32 bit	16 bit	8 bit high	8 bit low
General Purpose[a]	`rax`	`eax`	`ax`	`ah`	`al`
General Purpose[b]	`rbx`	`ebx`	`bx`	`bh`	`bl`
General Purpose[c]	`rcx`	`ecx`	`cx`	`ch`	`cl`
General Purpose[d]	`rdx`	`edx`	`dx`	`dh`	`dl`
General Purpose[e]	`rdi`	`edi`	`di`		`dil`
General Purpose[f]	`rsi`	`esi`	`si`		`sil`
General Purpose	`r8`	`r8d`	`r8w`		`r8b`
General Purpose	`r9`	`r9d`	`r9w`		`r9b`
General Purpose	`r10`	`r10d`	`r10w`		`r10b`
General Purpose	`r11`	`r11d`	`r11w`		`r11b`
General Purpose	`r12`	`r12d`	`r12w`		`r12b`
General Purpose	`r13`	`r13d`	`r13w`		`r13b`
General Purpose	`r14`	`r14d`	`r14w`		`r14b`
General Purpose	`r15`	`r15d`	`r15w`		`r15b`
Base Pointer	`rbp`	`ebp`	`bp`		`spl`
Stack Pointer	`rsp`	`esp`	`sp`		`bpl`
Program Counter/Inst. Pointer	`rip`	`eip`	`ip`		
Code Segment (= 0)			`cs`		
Data Segment (= 0)			`ds`		
Extra Segment (str ops, = 0)			`es`		
Variable-Purpose Segment			`fs`		
Variable-Purpose Segment			`gs`		
Stack Segment (= 0)			`ss`		
Processor Status	`rflags`	`eflags`	`flags`		

[a] Formerly the accumulator (temporary variable for math); was also used for I/O port access and interrupt calls

[b] Formerly base pointer for memory access; also got some interrupt return values

[c] Formerly loop counter; was also used for shifts and got some interrupt return values

[d] Formerly data; was used for I/O port access, arithmetic, some interrupt calls

[e] Formerly destination index for string and memory array operations, was also used for far pointer addressing with `es`

[f] Formerly source index for string and memory array operations

Table 3.3: Flags for x86-64

Bit	Label	Description
0	`CF`	Carry flag*
1		Reserved, always 1 in `eflags`
2	`PF`	Parity flag†
3		Reserved
4	`AF`	Adjust flag‡
5		Reserved
6	`ZF`	Zero flag; indicates result was 0
7	`SF`	Sign flag; indicates MSB of result was 1
8	`TF`	Trap flag; permits operation of a processor in single-step mode
9	`IF`	Interrupt enable flag§
10	`DF`	Direction flag; controls the direction of string processing
11	`OF`	Overflow flag; indicates result overflowed
12-13	`IOPL`	I/O privilege level; shows I/O privilege level of current program or task
14	`NT`	Nested task flag‖
15		Reserved, always 1 on 8086 and 186, always 0 on later models
16	`RF`	Resume flag; enables turning off certain exceptions while debugging code
17	`VM`	Virtual 8086 mode flag¶
18	`AC`	Alignment check (486SX+ only)
19	`VIF`	Virtual interrupt flag (Pentium+)
20	`VIP`	Virtual interrupt pending (Pentium+)
21	`ID`	Able to use `cpuid` instruction (Pentium+)

* Indicates arithmetic carry/borrow has been generated out of arithmetic logic unit's MSB

† Set if the number of set bits in the least significant byte is even

‡ Indicates when an arithmetic carry or borrow has been generated out of the four least significant bits (lower nibble). Primarily used to support binary-coded decimal arithmetic.

§ Determines if the CPU will handle maskable hardware interrupts

‖ In protected mode, indicates one system task has invoked another via `call` rather than `jmp`.

¶ In protected mode, allows the execution of real mode applications that are incapable of running directly in protected mode (a form of hardware virtualization).

Instructions

This section covers some of the important x86-64 instructions.

Function and Stack

call Pushes the address of the next instruction onto the stack, decrementing **rsp**, and then jumps to its operand.

enter Modifies stack for entry to procedure for high level language. Takes two operands: the amount of storage to be allocated on the stack and the nesting level of the procedure.

push Pushes its operand onto the stack, decrementing **rsp**.

pop Pops the current value off the stack, incrementing **rsp**, and stores the popped value in its operand.

leave Releases local stack storage created by the previous **enter** instruction.

ret Pops the current value off the stack, incrementing **rsp**, and jumps to it.

Jumps

jo Jump if overflow (OF)

jno Jump if not overflow ($\neg$OF)

js Jump if sign (SF)

jns Jump if not sign ($\neg$SF)

je Jump if equal (ZF)

jz Jump if zero

jne Jump if not equal ($\neg$ZF)

jnz Jump if not zero

jb Jump if below (CF)

jnae Jump if not above or equal

jc Jump if carry

jnb Jump if not below ($\neg$CF)

jae Jump if above or equal

jnc Jump if not carry

jbe Jump if below or equal (CF $\vee$ ZF)

jna Jump if not above

ja Jump if above ($\neg$CF $\wedge$ $\neg$ZF)

jnbe Jump if not below or equal

jl Jump if less (SF $\neq$ OF)

jnge Jump if not greater or equal

jge Jump if greater or equal (SF = OF)

jnl Jump if not less

jle Jump if less or equal (ZF $\vee$ SF $\neq$ OF)

jng Jump if not greater

jg Jump if greater ($\neg$ZF $\wedge$ SF = OF)

jnle Jump if not less or equal

jp Jump if parity (PF)

jpe Jump if parity even

jnp Jump if not parity ($\neg$PF)

jpo Jump if parity odd

jcxz Jump if **cx** register is 0

jecxz Jump if **ecx** register is 0

***Single instruction, multiple data* (SIMD)** These instructions, which operate on multiple 32- or 64-bit chunks, allow for a degree of vectorization (up to eight 64-bit values or 16 32-bit values at once on modern versions of x86-64 with AVX-512). When AVX-512 is supported, there are 32 512-bit **zmmN** registers available, each of which can also be accessed as a 256-bit **ymmN** or a 128-bit **xmmN** register. Those functions from the case studies that use SIMD instructions only go up to **xmm**. In some cases, they are just used for internal moving of several values at once, but several of the functions rely on SIMD instructions to support variable-length argument lists.

3.2.2 The System V AMD64 Application Binary Interface

While an ISA specifies the features of a processor and what it can do, there is plenty more that must be standardized for easy interoperability of software. An ABI specifies the interfaces for different components in a software system, though not always formally. Those interfaces include the binary format of *object files*, the format of program libraries, how system calls are made, and the standard *calling convention*, how function calls are structured on the machine code level.

As the programs used in Chapters 5 and 6 were compiled and run on Linux, this dissertation focuses on the System V AMD64 ABI [133], which is the standard for most Unix-based systems. The relevant aspects are the fact that the System V ABI uses the Executable and Linkable Format format for binaries, though we do not care about that once we are dealing with assembly, and the calling convention. The important aspects of the calling convention for this ABI are the following:

The first six integer or pointer arguments are stored in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, in that order. `r10` is used as a static chain pointer when there are nested functions. Certain floating-point arguments will be stored in `xmm0-7`. Any additional arguments will be passed along on the stack.

Integral values up to 64 bits are stored in `rax` while those up to 128 bits are split between `rax` and `rdx`. Floating-point return values are stored in `xmm0` and `xmm1`.

Registers `rbx`, `rbp`, and `r12-r15` are *callee-saved*, meaning they must be pushed and popped locally to preserve their values between function calls. All other registers must be saved by the caller.

3.2.3 GCC

The compiler used to build the case studies in Chapters 5 and 6 is important, as it uses the segment registers `fs` and `gs` for its own purposes. `fs` keeps track of the original value of the *stack canary* used to detect stack smashing [35] while `gs` is used to access thread-local storage. For this dissertation, only `fs` is relevant, as none of the case studies or examples in Chapters 5 and 6 used thread-local storage.

3.2.4 Basic Blocks

Much of the memory work in this document relates to the concept of *basic blocks*. A basic block is defined here as a sequence of assembly instructions whose behavior can be described using only state transitions and branches. An additional restriction is that they have no internal loops; they will always terminate, either due to successful completion or early failure. This definition differs slightly from the definition used by compilers such as LLVM. The compiler definition is stricter, as it requires that each block must terminate with a control flow instruction (or fall through, on the assembly level) and must not contain any other control flow instructions [80, 81].

3.2.5 Tail Call Optimization

Tail call optimization, implemented in most major C and C++ compilers, is a technique for converting recursive functions into loop-based ones that only use a set number of stack frames. Unfortunately, that can only be applied to functions that fit the requirements for

proper tail calls [100], as imperative languages like C were not designed with such features in mind. The technique also supports optimizing non-recursive tail calls, however, and that has a lower bar.

3.3 Summary

This chapter gave an introduction to formal methods, including formal semantics, symbolic execution, Floyd verification, Hoare logic, model checking, and theorem provers. It also described some basic aspects of the x86-64 ISA and System V AMD64 ABI as well as some general details on assembly.

Chapter 4

Symbolic Execution

This chapter covers the methodology used in Chapters 5 and 6 to formally determine the state changes caused by individual basic blocks. This methodology relies on a formal big-step semantics of the x86-64 ISA provided by Roessle, Verbeek, and Ravindran [106], described in Section 4.1. We then extended those semantics with additional rewrite rules to increase efficiency and properly reason about memory. Those rules are documented in Section 4.2. The rules involving reading and writing from memory form the basis for the memory preservation methodologies in Chapters 5 and 6. They essentially generate memory region VCs that must be discharged in order to prove memory preservation.

My main contribution to this chapter was working on additional proven-correct simplification rules for word arithmetic as well as more presimplification rules for various instructions and their variants (Section 4.2).

Example 4.1 (Aggregation)**.** Consider the following two instructions:

```
1 xor ax, ax
2 add al, 1
```

These instructions write to the 64-bit register `rax`, introduced in Section 3.2.1. Registers `ax` and `al` respectively refer to the low 16 and 8 bits of that register. Symbolic execution produces the following assignment: $\texttt{rax} := \langle 63, 16\rangle\texttt{rax} \bullet 1_{16}$. Here $\langle 63, 16\rangle$ denotes taking the higher 48 bits and $\bullet$ denotes concatenation, with 1_{16} being the number one zero-extended to 16 bits. The `xor` instruction sets the lower 16 bits of the register to zero while `add` increments the lower byte by one. Both instructions keep the higher 48 bits intact. The aggregate result is overwriting the lower 16 bits of the register with the 16-bit representation of the number one.

Note that if this had used `eax` instead, the upper 32 bits of `rax` would have been zeroed out as well due to the semantics of operations on 32-bit registers in x86-64.

4.1 Machine Model

In order to perform symbolic execution, you must first have some sort of *machine model*. The machine model used in this dissertation for the work in Isabelle/HOL is an extension of the work of Roessle, Verbeek, and Ravindran [106]. They embedded in Isabelle/HOL a bitvector-based, big-step semantics machine-learned from a modern version of the x86-64 ISA. That semantics included instruction set extensions such as the Streaming SIMD Extensions family to increase the possible programs the semantics could execute. To improve reliability of their work, it was tested against an actual, live x86-64 machine to prove semantic equivalence. The semantics they used was an extension of that provided by Heule et al. [64], who did the initial application of machine learning to derive semantics from a physical machine. This produced highly reliable semantics: they formally compared a subset of their automatically-generated semantics to manually written rules based on the Intel reference manuals and found that in the few cases where they differed, the Intel manuals were wrong. Note that this model does not include concurrency.

The model is structured as follows. It has some symbolic *state* defined as an Isabelle record that stores registers, flags, and 64-bit byte-addressable memory. The memory holds both instructions and data, as in the standard von Neumann model. Each instruction is executed by a *step* function, defined to suit the nature of the symbolic execution engine in use. The works presented in this dissertation in Chapters 5 and 6 each use their own, slightly different symbolic execution engine, though the ultimate behavior is executing a sequence of instructions one by one, modifying the state each time.

The instructions themselves are loaded from the machine model by mapping from the deeply-embedded instruction representation extracted within or supplied to the step function to the bitvector formulas provided by Roessle, Verbeek, and Ravindran [106]. If no such formula exists for the current instruction, a manually-implemented variant is used. There are several sets of instructions that are guaranteed to only have manual implementations due to limitations of the machine learning setup, with the major ones being jumps, `call`, `push`, `pop`, `enter`, `leave`, and `ret`.

4.1.1 Memory Model

Reads and writes of the machine model's memory space take a specific form. They operate on *memory regions*. A memory region $[a, s]$ is defined to have type $\mathbb{W} \times \mathbb{N}$; that is, its starting address a is a 64-bit word and its size in bytes s is a natural number.

Reading a region of memory from some state σ uses the notation $\sigma : *[a, s]$. In Isabelle, this operation internally reads the list of s bytes starting from the given address a in the appropriate order and converts it to a word. If it is clear from context which state is meant, the state will be omitted. Meanwhile, writing to memory uses the notation $x := e$, which has type $A_{\text{SP}} = (SP, E_{\text{SP}})$; these *assignments* denote writing an expression e to some location x that is a *state part*, SP; it can be a region, register, or flag. Flags can only take boolean expressions while the result for a register must be a 64-bit word. The behavior for regions in

Isabelle is to internally decompose the expression to write into its component bytes and then write those into memory in the appropriate order. The expressions themselves are of type E_{SP}, representing expressions over state parts. These expressions consist of common bit-vector operations including taking subsets of bits, bitstring concatenation, logical operators, casting, and floating-point, signed, and unsigned arithmetic.

In this dissertation, modifications to state are represented as sets of assignments, $\mathcal{P}(A_{\text{SP}})$, formulated as $\alpha = \{x_0 := e_0, x_1 := e_1, \dots\}$. These assignments are all independent; their initial conditions are based off of whatever state is present before application of the assignments, and thus they can be applied in any order. To order writes, use the notation $\alpha(x := e)$, indicating that assignment $x := e$ is applied after the set of assignments α. Notation $\sigma(x := e)$ or $\sigma\alpha$ indicates applying that assignment or set of assignments to the supplied state.

4.1.2 Restrictions of the Model

As the x86-64 ISA is a little-endian architecture, all operations on memory presented in this dissertation are designed with that in mind.

Example 4.2. Given the state $\sigma = \{[a, 2] := \texttt{0xEEFF}\}$, the read $\sigma : *[a, 1]$ would produce `0xFF`.

Support for big-endian architectures would require changing how reads and writes are performed, as both the formal Isabelle and informal Haskell models assume little-endianness in their implementation. Some ISAs are even *bi-endian*, allowing both big- and little-endian memory operations. These include modern versions of ARM, PowerPC, SPARC, and MIPS. Supporting bi-endianness would require additional complexity in memory handling.

Additionally, the usage of a shared data space for instructions and data, though very common, does involve some issues for verification. The model does not currently provide any memory protection schemes, such as those used in modern hardware, and there is nothing to prevent a write from overwriting the program itself. For that reason, the works presented in this dissertation must assume that the loaded assembly is never modified.

4.2 Rewrite Rules

The basic rules supplied by the formal machine model are not well-suited to verification; they are often very low-level bitvector/bitstring operations. While Roessle, Verbeek, and Ravindran [106] provided a large set of simplification rules to abstract away from the underlying representation, those rules did not cover all situations encountered in this dissertation, requiring the additions of more such rules during the process of verification. In particular, the decomposition of writes into bytes and recomposition of reads from bytes is hidden from the user under most circumstances, allowing better abstraction such as that depicted in Example 4.1.

Additionally, to increase performance, every instruction variant with learned semantics detected in an analyzed function was given a *presimplified* lemma. Most of those lemmas were obtained from [126]. They provide immediate abstractions of the low-level instruction representations that rely on the aforementioned simplification rules. Using these lemmas improves performance when performing symbolic execution as they greatly reduce the number of simplification rules that must be applied.

4.2.1 Memory Aliasing

This section provides an insight into the issue of *memory aliasing*. For example, consider the assignment $[a_1, s_1] \coloneqq v_1$ applied to the set of assignments $A = \{[a_0, s_0] \coloneqq v_0\}$. The result of that operation depends on whether the two regions $[a_0, s_0]$ and $[a_1, s_1]$ *overlap*, are *separate*, or have an *enclosure* relation. If they are separate, then the resultant minimal assignment set is $A' = \{[a_0, s_0] \coloneqq v_0, [a_1, s_1] \coloneqq v_1\}$. If they instead overlap, then the situation is more complicated. For example, in the case where $a_0 = a_1$ and $s_0 = s_1$, the resultant minimal assignment set would be $A' = \{[a_0, s_0] \coloneqq v_1\}$. Other forms of overlap or enclosure, such as writing two bytes to a four byte region or to regions that are not aligned, require even more complicated reasoning.

The actual definitions of those relations are as follows.

Definition 4.3 (Separation)**.** Two regions $r_0 = [a_0, s_0]$ and $r_1 = [a_1, s_1]$ are *separate*, notation $r_0 \bowtie r_1$, if and only if the following is true:

$$s_0 = 0 \vee s_1 = 0 \vee a_0 + s_0 \leq a_1 \vee a_1 + s_1 \leq a_0.$$

This means that, if at least one of the regions has zero size or the lower bound of one of the regions is equal to or greater than the upper bound of the other, those two regions are separate. If those regions are not separate, they *overlap*.

Definition 4.4 (Enclosure)**.** Region r_0 is *enclosed* by r_1, notation $r_0 \sqsubseteq r_1$, if and only if:

$$a_0 \geq a_1 \wedge a_0 + s_0 \leq a_1 + s_1.$$

This means that, if the lower bound of the first region is the same as or greater than the lower bound of the second region and the upper bound of the first region is either the same as or less than the upper bound of the second region, the first region is enclosed by the second.

4.2.2 Rewrite Rules for Memory

An additional problem is when a region that overlaps with at least one other region that has been modified is written to. To combine those writes, the regions must be *merged*.

Definition 4.5 (Merging)**.** The *merge*[1] of two symbolic assignments $r_0 = [a_0, s_0] \coloneqq v_0$ and $r_1 = [a_1, s_1] \coloneqq v_1$, where the write to r_0 occurs before the write to r_1, is defined as

$$r = [a, s] \coloneqq b_0 \bullet b_1 \bullet b_2, \tag{4.1}$$

where:

$$\begin{aligned}
a &= \min(a_0, a_1) \\
i_0 &= a_1 - a_0 \\
i_1 &= a_0 + s_0 - (a_1 + s_1) \\
s &= s_1 + \max(i_0, 0) + \max(i_1, 0) \\
b_0 &= \text{if } i_1 > 0 \text{ then } \langle 8s_0 - 1, 8s_0 - 8i_1 \rangle v_0 \text{ else } 0_0 \\
b_1 &= \langle 8s_1 - 1, 0 \rangle v_1 \\
b_2 &= \text{if } i_0 > 0 \text{ then } \langle 8i_0 - 1, 0 \rangle v_0 \text{ else } 0_0
\end{aligned}$$

As the merged region must encompass both original regions, its address a is the minimum of a_0 and a_1. The value stored in the merged region consists of three parts: whatever portion of v_0, if any, is below a_1; v_1 as a bitstring; and the part of v_0 above $a_1 + s_1$ (the upper bound of r_1), if there are any bits in r_0 above that address. For sets of assignments such as those mentioned above, merge is used as an infix operator, with order being important (the second assignment overwrites [parts of] the first, as shown above). Example 4.6 demonstrates a more concrete usage of merging.

Writing to Memory

The formal rewrite rule for writing to a new region into memory is structured as in Eq. (4.2). The underlined terms are the *reducible expressions*, or redexes. They are the subterms not in *normal form*, the ones that may be rewritten again after application of the rewrite rule.

$$\sigma(r_0 \coloneqq v_0)(r_1 \coloneqq v_1) \equiv \begin{cases} \underline{\sigma(r_1 \coloneqq v_1)}(r_0 \coloneqq v_0) & \text{if } r_0 \bowtie r_1 \\ \underline{\sigma((r_0 \coloneqq v_0) \text{ merge } (r_1 \coloneqq v_1))} & \text{otherwise} \end{cases} \tag{4.2}$$

The proof of correctness for the above rule is based on two lemmas. First, writing separate blocks is commutative. Second, the merge function is correct: the produced region is the result of two sequential and overlapping memory writes.

Reading from Memory

Reading from memory in the process of symbolic execution also requires analysis of separation and merging. Consider reading from the region $[a, s]$ given a set of assignments α, using Algorithm 4.1 as our guide. If an assignment to the exact region $[a, s]$ exists in the current

[1] This merge operates on the bit level, but technically the original Isabelle version uses byte lists; also, the Haskell version merges the left region onto the right, not the right onto the left as the Isabelle version does.

Algorithm 4.1 Symbolically reading from memory

Require: A set of assignments $\alpha : A_{\text{SP}}$ and symbolic region $[a, s]$
Ensure: A symbolic value and possibly-updated α
  function READMEM($\alpha, [a, s]$)
    if $\exists v \cdot ([a, s] \coloneqq v) \in \alpha$ **then**
      return (α, v)
    else
      $ovl \leftarrow \{([a', s'] \coloneqq v) \in \alpha \mid [a', s'] \not\bowtie [a, s]\}$
      $sep \leftarrow \{([a', s'] \coloneqq v) \in \alpha \mid [a', s'] \bowtie [a, s]\}$
      $[a_l, s_l], [a_r, s_r] \leftarrow$ the left- and rightmost regions in $\{[a, s]\} \cup ovl$
      $r \leftarrow [a_l, a_r - a_l + s_r]$
      $[a', s'] \coloneqq v' \leftarrow (r \coloneqq *r)$ merge ... merge ovl_1 merge ovl_0
      $\alpha' \leftarrow \{[a', s'] \coloneqq v'\} \cup sep$
      $a'' \leftarrow 8(a - a') - 1$
      return $(\alpha', \langle s + a'', a'' \rangle v')$
    end if
  end function

set of assignments, then the value assigned to that region, v, is returned. Otherwise, the algorithm must consider the set of assignments for all possibly overlapping and necessarily separated regions. One single assignment that accounts for all overlapping regions must be developed. To do this, the leftmost and rightmost overlapping regions are considered. These regions are defined as the regions that start at the smallest address a_l and end at the greatest upper bound $a_r + s_r$, respectively. The new region r has address a_l and size $a_r - a_l + s_r$. All of the overlapping regions are then merged into one single assignment based on r, starting with the trivial assignment $r \coloneqq *r$. This assignment does nothing but set up the merging, as it writes the value read from region r back to that same region. After merging, the current set of assignments is updated to be the merged region and assignment combined with all separate assignments. The final value read from memory is extracted from the merged assignment.

The correctness of the READMEM algorithm is derived from the correctness of its component operations.

Example 4.6 (Reading, writing, and merging)**.** Consider the following x86-64 assembly block:

```
1 a0: mov  word ptr [rsp-0x8], 0xEEFF
2 a1: mov dword ptr [rsp-0x4], 0xAABBCCDD
3 a2: mov  ax,                  word ptr [rsp-0x7]
4 a3: mov edi,                 dword ptr [rsp-0x6]
```

The instructions at addresses `a0` and `a1` write to two separate regions in memory, $r_0 = [\texttt{rsp} - 8, 2]$ and $r_1 = [\texttt{rsp} - 4, 4]$. Following the writes, the instruction at `a2` reads from region $[\texttt{rsp} - 7, 2]$, which is merged with r_0 to obtain $r_2 = [\texttt{rsp} - 8, 3]$. Reading from region

$[\texttt{rsp} - 6, 4]$ results in a merge with r_2 and r_1, producing region $[\texttt{rsp} - 8, 8]$. The aggregated assignment is then

$$[\texttt{rsp} - 8, 8] \coloneqq \texttt{0xAABBCCDD} \bullet \langle 31, 16\rangle * [\texttt{rsp} - 8, 8] \bullet \texttt{0xEEFF}.$$

Assuming an intial condition of $\texttt{rsp} = rsp_0$, the set M of memory regions required for the given block of assembly is ultimately

$$M = \{[rsp_0 - 8, 2], [rsp_0 - 4, 4], [rsp_0 - 7, 2], [rsp_0 - 6, 4], [rsp_0 - 8, 8]\}.$$

Reasoning over Memory Regions

Reads and writes both need to reason over separation and enclosure, so providing a means for users to easily specify those relations via assumptions over memory layout increases efficiency. This section covers formulating those assumptions and the necessary groundwork for automatic inference using them.

As stated in Section 4.1, the memory model in use is a simple, flat function from 64-bit words to bytes. As instructions and data are both stored in the same memory space, assumptions on their separation would be ideal. The function $\bigotimes$ is used to formulate such assumptions. It takes as input a set of regions annotated with unique IDs. These IDs allow reasoning over (in)equality of regions; without them, it would be impossible to determine whether two regions of the same size are equal if their addresses are non-trivial expressions.

Definition 4.7. Let M be a set of pairs of unique IDs and regions. M is *separated* if and only if all of its regions are separated:

$$\bigotimes M \equiv \forall (i_0, r_0), (i_1, r_1) \in M \cdot \text{if } i_0 = i_1 \text{ then } r_0 = r_1 \text{ else } r_0 \bowtie r_1 \tag{4.3}$$

This function over memory region sets compares all possible combinations of ID-region pairs in the supplied set, returning true only if each region has a unique ID and is separate from every other region in the set.

Originally, set M was intended to contain large regions, such as the whole stack frame. As the rewrite rules are focused on smaller regions, such as per-variable regions. rules that infer properties over smaller regions from larger ones are needed.

$$r \sqsubseteq r \tag{4.4a}$$
$$r_0 \bowtie r_1 = r_1 \bowtie r_0 \tag{4.4b}$$
$$r_0 \sqsubseteq r_2 \wedge r_1 \sqsubseteq r_3 \wedge r_2 \bowtie r_3 \longrightarrow r_0 \bowtie r_1 \tag{4.4c}$$
$$r_0 \sqsubseteq r_1 \wedge r_1 \sqsubseteq r_0 \longrightarrow r_0 = r_1 \tag{4.4d}$$
$$r_0 \sqsubseteq r_1 \wedge r_1 \sqsubseteq r_2 \longrightarrow r_0 \sqsubseteq r_2 \tag{4.4e}$$
$$r_0 \bowtie r_1 \wedge \operatorname{snd} r_0 \neq 0 \wedge \operatorname{snd} r_1 \neq 0 \longrightarrow r_0 \not\sqsubseteq r_1 \tag{4.4f}$$
$$\bigotimes(M) \wedge (i_0, r_0), (i_1, r_1) \in M \wedge i_0 \neq i_1 \longrightarrow r_0 \bowtie r_1 \tag{4.4g}$$

Equation (4.4) shows the inference rules for properties over memory regions. These rules are able to infer the properties of separation and non-enclosure for smaller regions based on assumptions over larger ones. However, they *cannot* infer enclosure.

Often, the only way to prove enclosure is to unfold its definition. This introduces two inequalities over words, as shown in Definition 4.4. Such inequalities can be solved using the Isabelle/HOL tool `unat_arith`, which is an arithmetic equation solver for bitvectors [39, 40]. That tool is augmented with several heuristics and auxiliary lemmas to facilitate enclosure proofs. However, such proofs are time-consuming and can significantly clutter the proof effort.

The initial solution to this issue, which is used in Chapter 5, relies on *parent regions*. A parent region is a member of set M and is thus a region annotated with an ID. Parent relationships are manually established to avoid having to do any unfolding. Local variables would have the stack frame as their parent region while global constants would have some data section as their parent. The following notation is used to link a memory region r_0 to a parent region r_1 with ID i: $\text{parent}(r_0, i, r_1)$. Given that information, the proof of enclosure is done automatically, and only once. The established enclosure properties are then used for inference as per the rules in Eq. (4.4).

As a concrete example, consider a two-byte array starting at address 10 and having ID 5. The region for this array would be $[10, 2]$, with ID formulation $(5, [10, 2])$. If we take the two bytes of the array as child regions, the region relations would be $\text{parent}([10, 1], 5, [10, 2])$ and $\text{parent}([11, 1], 5, [10, 2])$.

There is also an alternative to using parent regions: giving each small region its own ID. This avoids having to provide explicit parent relationships except for those cases where reads or writes of different size from or to the same region occur. Chapter 6 takes that approach.

Overflow

As a note, many of the formal rewrite rules regarding memory usage have an internal requirement that the supplied memory regions not overflow. That is, for any memory region r, its address plus its size must be less than 2^{64}. This is represented as $\text{NOBLOCKOVERFLOW}(r)$ and may be required as an explicit assumption in some cases. With the appropriate manual or generated region relations, however, it should not normally be necessary.

4.3 Summary

This chapter introduced symbolic execution, a way of aggregating the state changes for individual instruction semantics. Symbolic execution is generally implemented as a set of rewrite rules based off of some machine model. Within that model are rewrite and simplification rules for reading and writing memory, required for abstract, region-based memory reasoning. Separation and enclosure are the two main relations needed for such

reasoning. In some cases, reasoning about enclosure can be very time-consuming, and thus a set of assumptions and associated rewrite rules are provided that allow for user-provided memory layouts, which greatly increases productivity.

Chapter 5

Control-Flow-Driven Verification

The memory usage analysis approach presented in this chapter provides a Floyd-style methodology featuring annotations on specific instructions. It is designed for the proving of memory preservation over individual assembly functions. That property ensures that only documented regions of memory are modified; anything outside of those regions remains unchanged.

This control-flow-driven verification can be applied to functions with loops and subcalls, including directly-recursive calls. It can be used to prove the absence of common memory-related issues, such as buffer and stack overflows. In cases where overflow may occur, the methodology helps extract the assumptions required to prevent that. For efficiency, it selects the annotation locations, called *cutpoints*, such that every path through the program is symbolically executed only once.

An overview of the methodology's steps can be seen in Section 5.1, with formal definitions of the needed constructs presented in Section 5.2. Following that, Section 5.3 describes how the method uses composition on the function level and within function bodies. Two examples providing a brief demonstration of the methodology can be found in Section 5.4, while a real-world application to the HermitCore unikernel library [78] is presented in Section 5.5. Our observations regarding usage of the methodology on that case study are given in Section 5.6.

My three main contributions to the memory preservation work presented here were: 1. the development of a tool for producing proof skeletons for memory preservation, interfacing with the analysis tool `angr` (Section 5.1); 2. the development of structured proof strategies to flesh out and verify those skeleton proofs, along with the development of guidelines for invariants that provide for function-level composition (Section 5.3); and 3. the application of the methodology to functions in Sections 5.4.1 and 5.5.

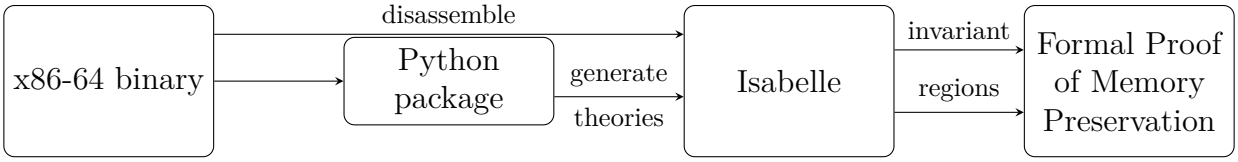


Figure 5.1: Overview of control-flow-driven memory preservation verification

5.1 Overview of Methodology

The first step in the process of analysis for a function is disassembly of an x86-64 binary containing it. This is done using a modified version of the `reassembly` analysis [129] of the binary analysis tool `angr` [117, 128]. That modified version was provided by Roessle, Verbeek, and Ravindran [106] for generating assembly usable with their Isabelle parser. By building on `angr`, the work of abstracting from binary to CFG is handled with minimal user input.

To achieve minimal symbolic execution, cutpoints are automatically selected by a Python package that relies on `angr`'s `CFGEmulated` control flow analysis. The cutpoints are described in Section 5.2.3. Basic starting predicates for those preconditions and postconditions as well as the cutpoints are generated, but the bulk of the information must be added manually. Larger-scale scalability is achieved by using function-level compositionality. Even recursive functions are supported, albeit with difficulty.

The process that selects cutpoints also generates skeleton memory preservation theories for every function analyzed. The theory files can then be opened in the theorem prover Isabelle and the assembly loaded using the parser of Roessle, Verbeek, and Ravindran [106]. Once that is done, a user can flesh out the invariants (Section 5.2.3) and add the necessary sets of memory regions that the functions write to in order to complete the proofs of memory preservation. Defining the necessary invariants for functions with complex control flow is generally a hard task, but targeting a property such as memory preservation does reduce the amount of work required as seen in Sections 5.4 and 5.5. The work is still not trivial, however.

5.2 Formal Definitions

A formal definition of memory preservation requires a formal basis to work with, and that basis is the machine model from Section 4.1.

5.2.1 Symbolic Execution for CFG-Driven Verification

While Chapter 4 provided a general overview of symbolic execution, this chapter requires a more specific look. The step function for this methodology takes the form $\textsc{step} : S \to (S \mid \bot_{\mathrm{E}})$. It takes the current state σ to execute from and returns the state σ' after execution of the current instruction, which is extracted from the current state based on the value of

the instruction pointer `rip`. If some sort of exception, such as a divide by zero, occurs, the function returns $\perp_{\text{E}}$ instead.

From the machine model, we manually derived a run function $\textsc{runUntil} : (S \to \mathbb{B}) \times S \to (S \mid \perp_{\text{E}} \mid \perp_{\text{NT}})$. This partial function takes as input a state predicate H and a state σ, producing a state σ' on successful completion. Predicate H denotes a *halting condition*, which typically instructs the run function to stop at a certain instruction address, such as that following a `ret`. The run function executes $\textsc{step}$ until H, applied to the current state, is true. Whenever an exception occurs, it stops and returns $\perp_{\text{E}}$. If the execution were to continue forever without an exception or reaching the halting condition (as would happen with an infinite loop), the function returns $\perp_{\text{NT}}$. Formally, this is achieved by a standard *least fixed point* (LFP) construction.

5.2.2 Hoare Triples for Memory Preservation

Unlike the usual formulation of Hoare logic [65, 88], Hoare triples for this work take one of the aforementioned halting conditions as their middle input rather than a program statement. The result is that the program statement, the block of instructions to execute, is characterized by the addresses of its initial and ending instructions, defined in P and H, rather than via specific syntax. Thus, we have the following definition:

Definition 5.1 (Hoare triple for memory preservation)**.** $\{P\}H\{Q\}$ denotes that, for any initial state that satisfies the precondition P and results in symbolic execution to the halting condition H terminating, the resultant state will be non-exceptional and satisfy postcondition Q.

This is formally expressed as:

$$\{P\}H\{Q\} \equiv \forall\sigma \cdot P(\sigma) \wedge \sigma' \neq \perp_{\text{NT}} \longrightarrow \sigma' \neq \perp_{\text{E}} \wedge Q(\sigma'), \tag{5.1}$$

where $\sigma' = \textsc{runUntil}(H, \sigma)$.

5.2.3 Floyd Invariant Foundation

Loops pose a significant problem when using symbolic execution to analyze code. One of the major issues is that they result in significant path explosion. While there are methodologies to reduce the number of paths to execute when using loops [96, 109], those methods are not currently formally verified and therefore not usable within Isabelle/HOL. Additionally, deciding the loop condition on a symbolic state may involve non-determinism (such as an event loop dependent on user input to exit), which can cause infinite execution.

Breaking up symbolic execution of loops is one method of resolving those issues. With the right annotations, it is possible to only need to symbolically execute one iteration per loop. This eliminates the above-mentioned loop issues. That breaking up of loops can be accomplished using a control-flow-based approach akin to Floyd verification [51]. A state predicate that

can be shown to hold for every iteration of a loop at some instruction within that loop will function as a *loop invariant*, symbolically characterizing the loop's behavior. This can be combined with a general methodology of structured preconditions and postconditions over annotated locations. If that methodology can show that the state at one such location satisfying the location's annotation will lead to any succeeding annotated locations also having states that satisfy their annotations, a Hoare triple as defined in Definition 5.1 can be inferred for the program as a whole (Theorem 5.3).

More formally, the *Floyd invariant* for a function is a partial function that takes the form $I : L \rightharpoonup (S \to \mathbb{B})$. This function maps from instruction addresses with invariants to the corresponding state predicate that is the invariant. As a technical detail, some function proofs require additional arguments to I that represent the arguments passed to the function.

Definition 5.2 (Floyd invariant)**.** A Floyd invariant I *holds* if and only if, for any state σ,

$$I(\operatorname{loc}\sigma)(\sigma) \longrightarrow \sigma' \neq \bot_{\mathrm{E}} \wedge (\sigma' = \bot_{\mathrm{NT}} \vee I(\operatorname{loc}\sigma')(\sigma')), \tag{5.2}$$

where $\sigma' = \textsc{RunUntil}((\lambda\sigma_r \cdot I(\operatorname{loc}\sigma_r) \neq \bot), \sigma)$ and $\operatorname{loc}\sigma_r$ is the current program location, stored in `rip` on x86-64 systems.

In words, if the Floyd invariant holds on the current state, then running to the next annotated location will not produce an exception. If that run terminates, then the state it produces will also satisfy the Floyd invariant.

The following theorem states that a Floyd invariant can be used to prove properties over its corresponding program or function as a whole:

Theorem 5.3 (Floyd and Hoare)**.** *Assume that Floyd invariant I holds and provides annotations for locations l_0 and l_f (the initial and final location). Let halting condition H stop at location l_f; that is, $H(\sigma) \longrightarrow \operatorname{loc}\sigma = l_f$. Then $\{I(l_0)\}H\{I(l_f)\}$.*

Proof. Remember from Definition 5.1 that

$$\{P\}H\{Q\} \equiv \forall\sigma \cdot P(\sigma) \wedge \sigma' \neq \bot_{\mathrm{NT}} \longrightarrow \sigma' \neq \bot_{\mathrm{E}} \wedge Q(\sigma').$$

Though there could be any number of additional annotations between l_0 and l_f, Floyd [51] showed by induction that a Floyd invariant that holds starting from some initial condition to an intermediate annotation at l_1 will also hold starting from that annotation. Thus, as I holds, we can substitute in $I(l_0)(\sigma)$ for $P(\sigma)$ and $I(l_f)(\sigma')$ for $Q(\sigma')$ without issue, resulting in the following statement:

$$I(l_0)(\sigma) \wedge \sigma' \neq \bot_{\mathrm{NT}} \longrightarrow \sigma' \neq \bot_{\mathrm{E}} \wedge I(l_f)(\sigma').$$

As we have already assumed that I holds, we can substitute in the right side of the implication from Definition 5.2 to obtain

$$\sigma' \neq \bot_{\mathrm{E}} \wedge (\sigma' = \bot_{\mathrm{NT}} \vee I(l_f)(\sigma')) \wedge \sigma' \neq \bot_{\mathrm{NT}} \longrightarrow \sigma' \neq \bot_{\mathrm{E}} \wedge I(l_f)(\sigma').$$

This then simplifies to

$$\sigma' \neq \bot_{\mathrm{E}} \wedge I(l_f)(\sigma') \longrightarrow \sigma' \neq \bot_{\mathrm{E}} \wedge I(l_f)(\sigma'),$$

which is trivial. □

In essence, Floyd-style verification models a program as a CFG where each edge is an implication.

5.2.4 Definition of Memory Preservation

The formal definition of memory preservation takes the form of a Hoare triple from Definition 5.1. Initially, there must be some predicate P that characterizes the initial state, at a minimum by setting the instruction pointer to the first instruction of the relevant function body. In addition, there is some set of memory regions M that the function is allowed to write to. M includes the stack frame and any utilized data sections from the source binary, as well as whatever heap memory was supplied or allocated, if any. Memory preservation formulates that any byte not within any of the regions in M has to remain unchanged throughout the execution of that function. The notation for this formulation is shown below.

Definition 5.4 (Memory preservation)**.** Let M be a set of memory regions, let P be a precondition, and let H denote a halting condition. A piece of assembly demonstrates *memory preservation* if and only if, for any address a and byte value v_0, the following implication holds:

$$(\forall r \in M \cdot r \bowtie [a, 1]) \longrightarrow \{P \wedge *[a, 1] = v_0\} H \{Q \wedge *[a, 1] = v_0\} \tag{5.3}$$

This definition states that, for every byte in memory outside of the memory region set, the following property holds:

1. if you start the current program fragment from a state that both satisfies the specified precondition and assumes that each byte has some value, then;
2. if you execute that program fragment to the specified halting condition, then;
3. you will end up with a state that satisfies the specified postcondition and retains the same value for all of those bytes outside of the specified memory regions.

5.3 Composition

As stated above, composition is used here for scalability. On the function call level, compositionality ensures that, when a function is called, a successful verification effort over that function can be reused if preexisting or developed later if need be. Taking this approach also allows minimizing symbolic execution even in non-loop situations, a form of internal compositionality.

5.3.1 Intra-Function

Consider the following pseudocode, which sequentially executes an if-statement and some program P:

Listing 5.1: Simple pseudocode

```
1 if b then x else y; P
```

The assembly corresponding to this code can be verified using symbolic execution. If executed in full, the symbolic execution engine would require first considering the case where b is true, executing x and subsequently symbolically executing program P. It would then consider the case where b is false, executing y followed by P. Program P would thus be symbolically executed twice. This repetition can be avoided by placing a cutpoint at the start of each block where control flow converges, resulting in all instructions being symbolically executed only once each. Each cutpoint, however, requires a state predicate contained in a Floyd invariant.

Reasoning about composition with the Hoare triples specific to this chapter requires a bit of work, as standard composition does not apply to Hoare triples that use halting conditions. Doing so is still possible, however.

Theorem 5.5 (Composition rule)**.** *Halting Hoare triples are compositional with respect to stronger halting conditions:*

$$\frac{\{P\}H\{Q\} \qquad \{Q\}H'\{R\} \qquad \forall\sigma \cdot H'(\sigma) \longrightarrow H(\sigma)}{\{P\}H'\{R\}}$$

Proof. Consider a symbolic run that executes until halting condition H'. It is possible to break this run into two parts by first running until a halting condition H and then until H'. This requires that H' is *stronger* than H; that is, H' implies H. Doing so ensures that the run first stops at H before it stops at H' (as it is possible for H to hold when H' does not, but not the other way around). □

Example 5.6. Now consider the block of assembly that could be generated for Listing 5.1. Let l_f denote the final location of the block while l_P denotes the initial location of program P. Theorem 5.5 can be used by instantiating H to halt at either location l_f or l_P and instantiating H' to halt at l_f. As long as programs x and y do not contain `goto`s or some other instruction that violates expected control flow, condition H will ultimately be equivalent to just halting at l_P. As H' is stronger than H, compositionality is possible.

5.3.2 Function Calls

Generally, compositionality over function calls requires proving that the stack pointer, after execution of a return, has the same value it did before the corresponding function call. Practically, this means proving that the body of the function results in `rsp` $= rsp_0 + 8$. This can be proven even for functions with optimized tail calls that just swap the final `call`+`ret` combo for a jump as long as the body of the called function is treated as part of the callee.

Example 5.7. Consider a function f starting in a text section at location l_0. The function is called from a different text section by the instruction `call f` at location l_{call}. This means

the return address for the call is $l_{call} + 5$.[1] After execution of `call f`, the program will be at location l_0 and the stack pointer, `rsp`, will have some value rsp_0. In order to apply compositionality to function calls, the pre- and postcondition have to meet the following requirements. First, the precondition must imply that the return address is pushed on the stack (a task performed by `call`): $*[rsp_0, 8] = l_{call}+5 \wedge \texttt{rsp} = rsp_0$. Second, the postcondition must imply that after `ret`, the net effect of the function body is that the stack pointer has been incremented by 8: $\texttt{rsp} = rsp_0 + 8 \wedge \text{loc} = l_{call} + 5$. Note that `call` itself decrements the stack pointer by 8, so this implies the net effect, from the point of view of the caller, is that the stack pointer is unchanged. The postcondition must also show that the location has been set back to the return address, $l_{call} + 5$.

Besides the stack pointer, modern calling conventions have other *callee-saved* registers, such as `rbp` and `r12-r15`. It is generally assumed that the net effect of a function call does not touch these registers. Consider a situation in which `rbp` contains an address, which will be used as the target of a write after a function call. In order to prove memory preservation, `rbp` must be shown to be preserved. Generally, this is easy to prove by strengthening the pre- and postcondition with a conjunct $\texttt{rbp} = rbp_0$. The proof is generally not complicated, as these callee-saved registers are pushed onto the stack at the beginning of functions that use them and popped off at the end.

In many cases, users of a verification methodology over functions will encounter calls to functions that are not included in the verification effort. These may be system calls or simply functions not currently under consideration due to unsupported features or lack of time. External functions are simply assumed to have correct behavior and are thus left out of the existing analysis, leaving those functions in the TCB. One issue that may occur is a desired function making an optimized tail call to an external function, which cannot be treated as part of the callee and prevents verification. One possible solution, used in the next chapter, is to revert the jump to `call`+`ret`. As the underlying function is assumed to be correct, this will result in the proper `rsp` restoration.

5.4 Examples

The following sections present some basic explanation of the procedure used in this chapter via two simple functions. The first, in Section 5.4.1, is a non-recursive function that features a loop. The second, in Section 5.4.2, gives an example of a recursive function and shows why those are difficult to reason about.

5.4.1 Non-recursive Loop Example: `pow2`

This simple loop-based function, shown in Listing 5.2, raises two to the power of its argument. The assembly code was obtained by compiling a C program containing the function with GCC 7.2.0 and disassembling it using the modified `reassembly` analysis mentioned in Section 5.1.

[1] the `call` instruction is five bytes long

Listing 5.2: `pow2` in C

```
unsigned long pow2(unsigned exponent) {
    unsigned long a = 1;

    for (unsigned i = 0; i < exponent; ++i) {
        a += a;
    }

    return a;
}
```

The input is stored in `edi`. The function uses memory in five places, all on the stack. These are expressed relative to the original value of the stack pointer rsp_0: 1. the caller's `rbp` at `rbp` $= rsp_0 - 8$ (eight bytes); 2. the argument to the function at `rbp` $-$ `0x14` (four bytes); 3. the accumulation variable and return value at `rbp` $- 8$ (eight bytes); 4. the counter variable at `rbp` $-$ `0xc` (four bytes); 5. and the address of the location to return to at rsp_0. The memory region for this function is thus $r_s = [rsp_0 - 28, 36]$. Assigning region r_s with ID i_s and the untouched region $[a, 1]$ with ID i_a, parent relationships can then be established as shown below:

$$\text{parent}([rsp_0, 8], i_s, r_s) \qquad \text{parent}([rsp_0 - 20, 4], i_s, r_s)$$
$$\text{parent}([rsp_0 - 8, 8], i_s, r_s) \qquad \text{parent}([rsp_0 - 28, 4], i_s, r_s)$$
$$\text{parent}([rsp_0 - 16, 8], i_s, r_s) \qquad \text{parent}([a, 1], i_a, [a, 1])$$

For the memory preservation proof of this function, we chose to associate annotations at the start of the function, an instruction that broke the loop, and the return address of the function (a logical variable, as the caller of the function is unspecified for this proof). Figure 5.2 shows the Floyd invariant in CFG form for this function. The invariant carries through the preservation of memory, showing that region $[a, 1]$ maintains its value throughout. The equalities over `rbp` and `rsp` are used by the memory region reasoner. For compositional purposes, as described in Section 5.3, the function is also shown to preserve the value of the stack pointer. Given the parent regions presented above, the proof that the Floyd invariant holds is executed automatically using the symbolic execution engine described in Chapter 4.

5.4.2 Recursion: Factorial

The factorial operation provides a simple example of recursion. The basic definition of factorial is $n! = \prod_{i=1}^{n} i$. This results in a number that is the product of the numbers from 1 to n. Expressed in recursive form, that definition is:

$$n! = \begin{cases} n * (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases} \tag{5.4}$$

Listing 5.3: `pow2` in x86-64 assembly

```
0  pow2:
1      push rbp ; Size:1
2      mov  rbp, rsp ; Size:3
3      mov  dword ptr [rbp - 0x14], edi ; Size:3
4      mov  qword ptr [rbp - 8], 1 ; Size:8
5      mov  dword ptr [rbp - 0xc], 0 ; Size:7
6      jmp  .label_10 ; Size:2
7  .label_11:
8      shl  qword ptr [rbp - 8], 1 ; Size:4
9      add  dword ptr [rbp - 0xc], 1 ; Size:4
10 .label_10:
11     mov  eax, dword ptr [rbp - 0xc] ; Size:3
12     cmp  eax, dword ptr [rbp - 0x14] ; Size:3
13     jb   .label_11 ; Size:2
14     mov  rax, qword ptr [rbp - 8] ; Size:4
15     pop  rbp ; Size:1
16     ret  ; Size:1
```

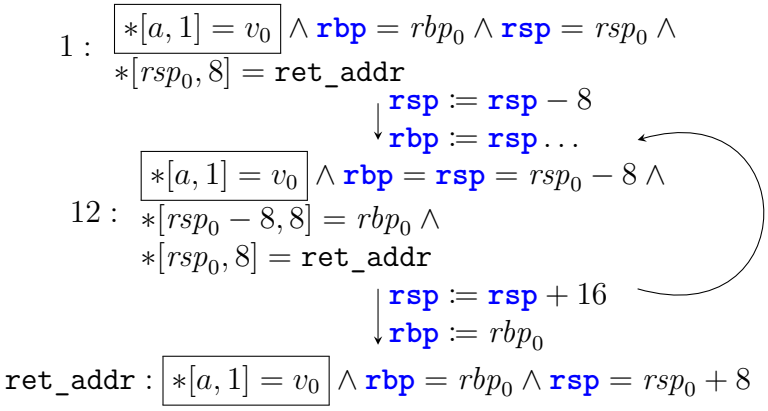


Figure 5.2: Floyd invariant for `pow2` in CFG form

Listing 5.4: Factorial in C

```
uint64_t factorial(uint8_t n) {
    if (n) {
        return n * factorial(n - 1);
    }
    return 1;
}
```

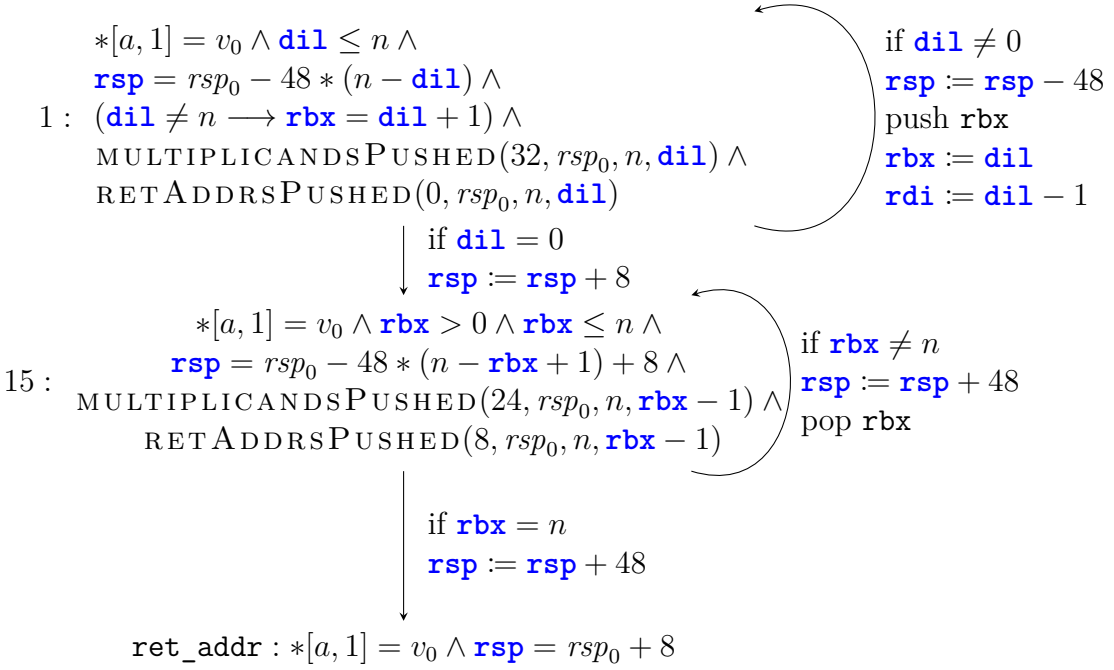


Figure 5.3: Floyd invariant for factorial in CFG form

The C equivalent of that function is shown in Listing 5.4. The assembly snippet shown in Listing 5.5 is again the result of a function compiled with GCC 7.2.0 and disassembled by the tweaked `reassembly` analysis [129]. In this case, the function performs a recursive factorial calculation on the value n stored in `dil` (the lowest eight bits of `edi`/`rdi`). It essentially consists of two loops, one loop to perform storing the integers from n to 2 on the stack as the function is called recursively and the second to multiply all those values together as each call returns.

As with `pow2`, the proof for this function relies on an rsp_0, though in this case that value specifically refers to the value of `rsp` for the first/topmost call to the recursive function. The memory locations operated on by this function are similar to those of `pow2`, but the memory locations themselves cannot be directly offset from rsp_0 due to the function's recursive nature. The set M of separated parent regions is characterized by the following assumptions:

Listing 5.5: x86-64 assembly of factorial example

```
0  factorial:
1      push   rbp
2      mov    rbp, rsp
3      push   rbx
4      sub    rsp, 0x18
5      mov    eax, edi
6      mov    byte ptr [rbp - 0x14], al
7      cmp    byte ptr [rbp - 0x14], 0
8      je     .label_12
9      movzx  ebx, byte ptr [rbp - 0x14]
10     movzx  eax, byte ptr [rbp - 0x14]
11     sub    eax, 1
12     movzx  eax, al
13     mov    edi, eax
14     call   factorial
15     imul   rax, rbx
16     jmp    .label_13
17 .label_12:
18     mov    eax, 1
19 .label_13:
20     add    rsp, 0x18
21     pop    rbx
22     pop    rbp
23     ret
```

$$\forall m \leq n \cdot (i_m, [rsp_0 + 48 * m - (n * 48) - 40, 40]) \in M \tag{5.5a}$$
$$\forall m \leq n \cdot (i'_m, [rsp_0 + 48 * m - (n * 48), 8]) \in M \tag{5.5b}$$
$$(i_a, [a, 1]) \in M \tag{5.5c}$$

The first assumption models all stack frames of size 40. The second models the parts of the stack where return addresses are pushed. The parent relations are defined in a similar way.

The following functions assist in characterizing the stack frame for any particular call in the recursive chain. To start with, MULTIPLICANDSPUSHED characterizes the multiplicands currently stored on the stack for any particular recursive call, both in the initial recursing loop as well as in the second loop as the recursive calls return. The following functions, RETADDRESS and RETADDRSPUSHED, establish that return address 15 is pushed to the correct memory location for every call except the first. For the first call, the topmost stack frame, the initial return address must also be properly stored.

$$\text{MULTIPLICANDSPUSHED}(\mathit{offset}, rsp_0, n, x) \equiv$$
$$\forall i \cdot n > x \wedge i < n - x - 1 \longrightarrow *[\texttt{rsp} + i * 48 + \mathit{offset}, 8] = i + 1 + \texttt{rbx} \tag{5.6}$$

$$\text{RETADDRESS}(rsp, rsp_0, \mathit{offset}, n, x, i) \equiv$$
$$\text{if } rsp + (n - x - i) * 48 - \mathit{offset} = rsp_0 \text{ then } \texttt{ret\_addr} \text{ else } 15 \tag{5.7}$$

$$\text{RETADDRSPUSHED}(\mathit{offset}, rsp_0, n, x) \equiv$$
$$\forall i \leq n - x \cdot *[\texttt{rsp} + (n - x - i) * 48 - \mathit{offset}, 8] = \text{RETADDRESS}(\texttt{rsp}, rsp_0, \mathit{offset}, n, x, i) \tag{5.8}$$

The Floyd invariant for the factorial function is shown in Fig. 5.3 and Eqs. (5.6) to (5.8). The first loop, from location 1 back to 1, goes deeper into recursion, pushing values onto the stack until `dil` becomes 0. As stated, MULTIPLICANDSPUSHED and RETADDRSPUSHED characterize the stack frame for every call, ensuring that all necessary information is properly stored. Once $\texttt{dil} = 0$, the function has executed its final recursion and control reaches location 15. The second loop then pops values of the stack until $n = \texttt{rbx}$, at which point there are no more recursive stack frames to pop and the final result of the factorial operation can be returned.

A Hoare triple can now be derived from the the Floyd invariant. This is done by instantiating variable n with `dil`. Doing so simplifies the precondition, as initially, no values or return addresses are pushed other than the upmost return address. The resulting Hoare triple becomes:

$$\begin{array}{l} \{*[a, 1] = v_0 \wedge \texttt{rsp} = rsp_0 \wedge *[rsp_0, 8] = \texttt{ret\_addr}\} \\ H \\ \{*[a, 1] = v_0 \wedge \texttt{rsp} = rsp_0 + 8 \wedge \text{loc} = \texttt{ret\_addr}\}. \end{array} \tag{5.9}$$

This, combined with the regions presented above as assumptions, provide us with our theorem of memory preservation.

5.5 Application: HermitCore

The concept of *unikernels* has existed in the world of virtualization for over five years now. The term "unikernel" can refer to any single-address-space program. All that is required is that it be compiled with a library that provides all kernel code necessary to run the program. This bypasses the need for a separate OS [82], allowing the program to be used directly with a hypervisor or even run on a bare metal system with no additional support. This allows for reduced overall size and a reduction in attack surface by leaving out those kernel components that are not necessary.

Slightly implied by the mention of hypervisors, unikernels are intended for use in the same situations as traditional *virtual machine*s (VMs) or Docker containers. They are meant for simultaneous juxtaposed execution in a virtualized setting, with many single-purpose unikernels all performing their own tasks in isolation. This makes unikernels an interesting target for verification, as they aim to provide a high speed and real-time environment for cloud software.

The unikernel library HermitCore [78] was chosen to demonstrate the applicability of this methodology due to its established functionality and decent size. Designed for the x86-64 ISA, HermitCore is mostly written in C. While it does use some inline assembly, not uncommon in kernel code, that is no issue for the assembly-level methodology presented here. The subset of HermitCore functions that were verified feature features such as loops, pointers, complex data structures, function calls, and recursion. The 63 functions analyzed were generally compiled unoptimized, but twelve of those functions were also analyzed in their optimized forms. This was done to show that the more complex code produced by optimizing compilers can also potentially be handled. The proofs and all associated code are available on Figshare [16].

5.5.1 Functions Analyzed

The functions from Hermitcore that were selected for analysis are summarized in Table 5.1. The `dequeue_*` functions involve operations on a generic circular queue or ring buffer. The `buddy_*` functions, meanwhile, are internal to HermitCore's implementation of `kmalloc`. HermitCore's task scheduler is assisted by the linked list manipulation `task_list_*` functions as well as various functions from `tasks.c`. Next, the `vring_*` functions are involved with virtual I/O operations. Various system call wrappers from `syscall.c` were also handled, as well as eight functions from `spinlock.h`. In addition to those sets of functions, the following `string.h` functions were verified: `memcpy`, `memcmp`, `memset`, `strlen`, `strcpy`, `strncpy`, `strcmp`, and `strncmp`.

The string functions were of particular interest due to the implicit assumption of null termination for those functions that do not have an explicit ending count. Those functions, the ones whose names do not contain `n`, require an explicit assumption of null termination in their verification process. Otherwise they would continue to execute past the desired end of the supplied arrays, reading/writing memory until a memory error occurs. As the

Table 5.1: Summary of functions analyzed

Functions	Count	SLOC	Insts†	Loops	Rec.	Pointer args	Globals	Subcalls	-O3
`dequeue_*`	3	46	159			3		3	3
`buddy_*`	5	67	225	1	1	1	3	3	3
`task_list_*`	3	43	128			3			3
`vring_*`	3	19	80			1			3
`string.h`	8	81	280	8		8			
`syscall.c`	23	293	857	5		19	7	17	
`tasks.c`	10	122	396	2		3	9	4	
`spinlock.h`	8	89	254	2		8	2	6	
Total	63	760	2379	18	1	46	21	33	12

† Non-optimized count

memory model used in this dissertation does not support detection of access violations for unallocated areas of memory, that would effectively mean an infinite loop. Those functions with an explicit iteration limit do not need to assume null termination, as they will eventually terminate even if a null character is not encountered. Due to the lack of access violation support, we assume the arrays are of sufficient length even if they do not possess a null terminator within the specified range.

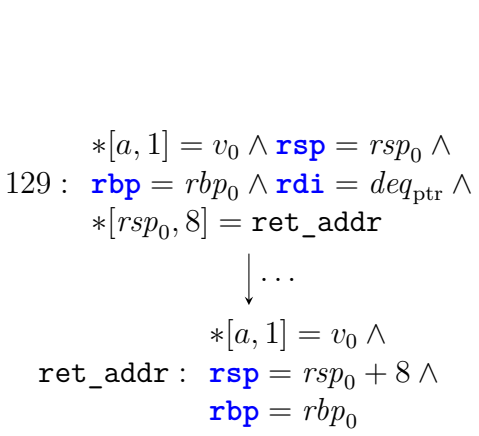
All of these functions were isolated and then compiled into binaries. Because of this, functions marked as `static inline` had those qualifiers removed. This prevented them from being eliminated when compiled with optimizations, as most of the functions would otherwise have their bodies inlined.

Figures 5.4a and 5.4b show the CFGs for two of the HermitCore functions verified here, `dequeue_push` and `buddy_large_avail`. The former pushes a value onto a generic array-based queue while the latter checks for the smallest available reused memory block for a given allocation size. The former, lacking any loops, requires only pre- and postconditions (though additional invariants may be added). In contrast, the latter function requires a loop invariant in addition to the pre- and postconditions.

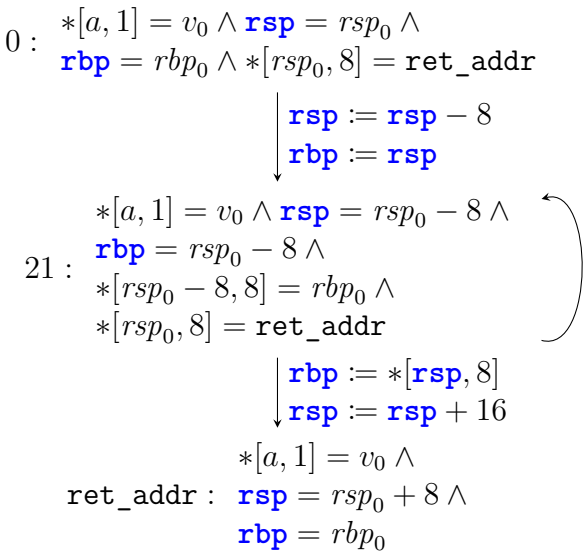
5.6 On Usability

The three main aspects of per-function user interaction for this methodology are 1. defining a Floyd invariant, 2. strengthening the precondition,[2] and 3. finishing the proof of memory preservation. The functions analyzed in the above case study provided some significant insight into the usability of those aspects.

[2]Includes adding additional memory regions and region relationships



(a) `dequeue_push`



(b) `buddy_large_avail`

Figure 5.4: Floyd invariants for the described case study functions in CFG form

5.6.1 Defining the Invariant

While restricting the verification effort to memory preservation does reduce the effort required to provide Floyd invariants, it does not eliminate it. This is more of a problem for loops with complex behavior and is a significant problem for recursive functions. With non-looping control flow, the primary effort required for invariant predicates is showing how input arguments are carried through the program (stored on the stack, in registers, etc.). With loops, the exact formulation relies on development of a symbolic representation of the behavior of the loop as it relates to memory accesses.

Meanwhile, recursive functions that cannot be flattened into tail recursion [100], such as those described in this chapter, are equivalent to two loops operating on the stack. Of course, every loop needs an invariant. The first loop invariant must characterize every call of the recursive function, which pushes data onto the stack, and the second every return, which pops data off.

At a minimum, the individual stack and frame pointers, as well as all the return addresses, must be shown to be preserved for their extant, being pushed on and popped off the stack. Any additional stored conditions that may affect memory usage must be kept track of as well.

On a nicer note, an advantage of the requirements for proofs over recursion is that they essentially require showing termination of the recursion and thus the (conceptual) avoidance of stack overflows. Proving lack of overflow for a specific stack size would require some additional clauses in the analysis.

5.6.2 Strengthening the Precondition

Another aspect of Floyd invariant development that is not easily determined ahead of time is how the function precondition must be strengthened. Making reasoned guesses about the necessary precondition clauses is one way to proceed, and source code annotations as well as reference documentation may provide additional help, but sometimes it is necessary to just symbolically execute until non-determinism is encountered. At that point, the cause of the non-determinism can be identified and the precondition can be strengthened in such a way so as to eliminate that non-determinism. Because this proof methodology works on the assembly level, it may well expose implicit or undocumented preconditions.

Formulation of the memory region set M as well as parent relationships, if necessary, are also generally manual. If a necessary region is not present, symbolic execution will result in non-determinism, requiring another round of user input.

5.6.3 Finishing the Proof

After symbolic execution for a basic block has completed, a proof that the resulting symbolic state satisfies the Floyd invariant is generally required. In most cases, that proof can be handled by Isabelle/HOL using standard off-the-shelf libraries, either ones included with Isabelle or ones from the Archive of Formal Proofs [44] (though not necessarily efficiently). Recursion is the primary exception, with the proofs of stack and frame pointer preservation requiring significant ITP over word arithmetic.

5.7 Summary

This chapter covered one method for formal verification of memory preservation in x86-64 binaries, showing that functions in a binary restrict themselves to certain regions of memory. The approach here aimed to automate verification while still allowing user interaction wherever necessary. As a semi-automated approach, it requires setting up an invariant, which traditionally is a hard problem in itself. Requirements for memory preservation invariants were provided for several examples. For recursive functions, more involved invariants are required, along with ITP to show preservation of the stack and frame pointers. Invariants may include preconditions necessary for excluding exceptional behavior, which can include stack or buffer overflows. Such preconditions can be exposed directy by applying the methodology to a disassembled binary instead of deriving them from documents or source code annotations.

The approach was applied to functions of HermitCore, a unikernel OS. Memory preservation was formally proven for functions with loops, recursion, C structs and unions, and dynamic memory operations. All verified functions were verified with non-optimized compilation, and some had their optimized versions verified as well.

Chapter 6

Syntax-Driven Verification

While the methodology presented in the previous chapter for verifying memory preservation works well, it is not ideal. The need to manually formulate regions and the amount of work required for developing invariants reduces potential scalability.

In order to deal with those downsides, this chapter introduces the concept of *formal memory usage certificate*s (FMUCs) generated by untrusted, informal tools. FMUCs consist of two main components: theorems on memory preservation and *proof ingredients*. The proof ingredients are assumptions on memory layout, control flow information, and invariants generated to reduce the amount of work required from end users.

Certificate generation is presented in Section 6.1, while the process of verification in Isabelle is documented in Section 6.2. A full example of FMUC usage can then be found in Section 6.3. That example could theoretically overwrite its own return address due to its pointer arguments, causing CFI issues. The associated FMUC provides preconditions to prevent such cases along with a formal proof of return address preservation under those conditions. Following the example in Section 6.4 is an in-depth case study on the Xen Project hypervisor [29]. In total, FMUCs were generated and proofs discharged in Isabelle for 251 Xen functions.

My primary contributions to the certificate generation and verification approach presented in this chapter include the Hoare rules developed for memory preservation as well as the VCG used to apply those rules to *syntactic control flow* (SCF) in Isabelle (Section 6.2). I also greatly expanded the code for full translation of the FMUCs into the Isabelle/HOL language and helped adapt the SCF into a form suitable for verification in Isabelle (Section 6.2.1). Additionally, I contributed to invariant generation (Section 6.1.3) and performed much of the verification work for our large case study, presented in Section 6.4.

6.1 FMUC Generation

FMUCs require the assembly code of a program as input. That source assembly could be

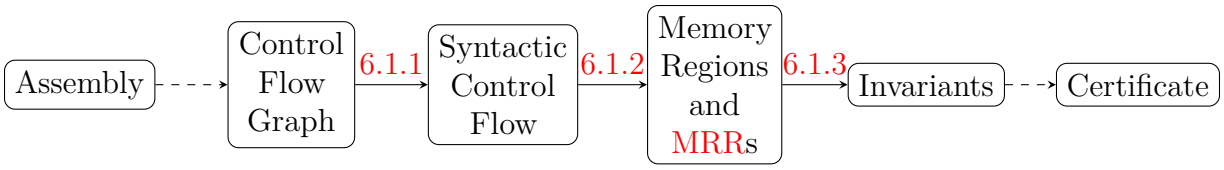


Figure 6.1: Overview of FMUC generation

obtained from a binary using a disassembler, such as `objdump`, IDA[1], Ghidra's decompiler[2], or Capstone [92]. If source code is available, it could be generated directly by a compiler instead. Each function specified for verification receives an FMUC; those that are not included in the verification effort, including system calls and functions from dynamic libraries, can be treated as black boxes, the usage of which is described in Section 6.2.5.

The general procedure for generating FMUCs, laid out in Fig. 6.1, can be broken up into three main parts. The first part involves control flow extraction from the supplied assembly using a CFG analysis similar to angr's CFGFast [117], ultimately producing an SCF (the details of which are presented in Section 6.1.1). Afterwards, per-basic block symbolic execution is utilized to generate the set of memory regions read and written by the function in question. This was detailed in Chapter 4. To eliminate duplicates and produce MRRs showing which regions overlap or are enclosed or separate, the region sets are then fed to the SMT solver Z3 [41]. Symbolic execution is also used in the process of generating the pre- and postconditions for each basic block, elaborated on in Section 6.1.3.

With the exception of MRR generation, none of the steps in this procedure are included in the TCB. The process of verifying the generated FMUC (see Section 6.2) will fail if there are issues in control flow extraction, SCF generation, informal symbolic execution, or invariant generation. MRR generation is an exception because the MRRs are formulated as assumptions, and thus inconsistent MRRs will result in vacuous proofs. This is why the methodology relies on Z3 for MRR generation; using a known-reliable tool greatly reduces the possibility of issues.

6.1.1 Control Flow Extraction

As described in Section 3.1.2, in order to apply a VCG that utilizes Hoare rules to verify a Hoare triple, there must be some syntactic structure to apply those rules to. This chapter uses a syntactic representation of control flow called SCF in part for that purpose. SCF expresses assembly programs as a combination of basic blocks, branches, loops, and function calls. The following grammar provides a description of SCF produced by the extraction code. Each basic block is represented by the polymorphic type β, while branching conditions are

[1] https://www.hex-rays.com/products/ida/index.shtml
[2] https://ghidra-sre.org/

represented using the polymorphic type Φ.

$$\begin{aligned}
\langle \text{scf} \rangle &\models \langle \text{scf} \rangle \texttt{ ; } \langle \text{scf} \rangle \mid \texttt{Block } \beta \mid \texttt{Skip} \mid \texttt{Continue} \mid \texttt{Break } \langle \text{br} \rangle \\
&\quad \mid \texttt{If } \Phi \texttt{ Then } \langle \text{scf} \rangle \texttt{ Else } \langle \text{scf} \rangle \texttt{ Fi} \mid \texttt{Loop } \langle \text{scf} \rangle \texttt{ Pool } \langle \text{res} \rangle && (6.1) \\
\langle \text{br} \rangle &\models \mathit{ID} \mid \epsilon && (6.2) \\
\langle \text{res} \rangle &\models \texttt{Resume } \{(\mathit{ID}, \langle \text{scf} \rangle), \ldots\} \mid \epsilon && (6.3)
\end{aligned}$$

Loops in this formulation have no exit condition; instead, they rely on having one or more internal `Break` statements, which may have an identifier to indicate how the loop was exited, for termination. `Continue`s function the same as in C, causing loop execution to skip to the next iteration. For loops that have multiple exit points, `Resume` statements provide different code to execute based on which exit was taken as indicated by the `Break` identifier.

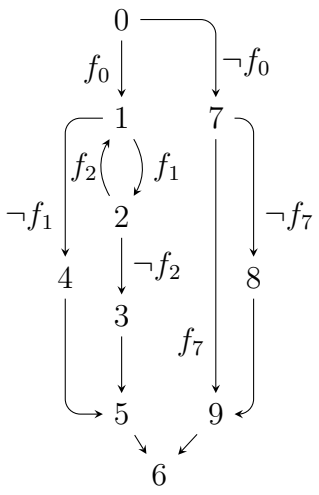
Notably, the above data structure does not explicitly contain control flow statements such as `goto` or `throw/catch`. Unconditional jumps like `goto`s make code harder to reason about in a structured way and can be modeled by the existing syntactic constructs, while structured exception handling as used in C++ is generally provided by external function calls.

Example 6.1. Figure 6.2 provides an example of SCF extracted from a CFG. The CFG in Fig. 6.2a can be seen to have two branching conditions that do not involve loops, one from `Block` 0 with condition f_0 and one from `Block` 7 with condition f_7. This leads to if statements with those conditions being added to the SCF in Fig. 6.2b after their respective blocks. The one loop present has two exit points. If condition f_1 is false after execution of `Block` 1, the loop will exit to `Block` 4, while `Block` 2 will exit to `Block` 3 if f_2 is false. This leads to the `Break` statements present in the extracted SCF in their respective conditional statements annotated with the IDs for their associated exit blocks. Those two exit points also result in the generation of a `Resume` clause indicating where those `Break`s exit to.

Restrictions

There two important restrictions on the current control flow extraction approach, the more severe of which is the lack of support for indirect branching. The CFG analysis done by the current extraction algorithm is not strong enough to handle indirect branching at the moment. In some cases, the set of possible branches can be determined based on the local function context, but the result of an indirect branch is often based off of input arguments, as well. Even if the result set might be determinable with static analysis, it would have to be interprocedural, and branch destinations based on external input cannot be determined.

Finally, as the if-then-else statement provides the sole form of branching control flow, the algorithm is not optimal due to the duplication of blocks to fit less structured control flow into a more structured model. In the worst case, it can result in SCF explosion, described below.



(a) Example CFG

```
Block 0;
If f_0 Then
    Loop
        Block 1;
        If f_1 Then
            Block 2;
            If f_2 Then Continue Else Break 2 Fi
        Else
            Break 1
        Fi
    Pool Resume {(2, Block 3), (1, Block 4)};
    Block 5
Else
    Block 7;
    If f_7 Then Skip Else Block 8 Fi;
    Block 9
Fi;
Block 6
```

(b) Syntactic Control Flow

Figure 6.2: Example of control flow extraction

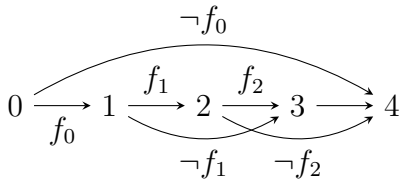
SCF Explosion

The algorithm is not optimal in terms of generated SCF size as certain basic blocks may be duplicated. There are two situations where basic block duplication occurs, one less common than the other. The less common situation is when loop having multiple entry points, which can occur in situations that involve less-structured control flow, such as a C program that jumps into a loop using `goto`. Such situations are relatively uncommon, even in optimized code. If it does happen, the entire loop must be duplicated. The more common situation, by contrast, involves complex conditional branching that can occur even without loops.

Example 6.2. Figure 6.3 shows a small example of branching control flow that results in `Block` 3 being duplicated. That block could itself be an even more complicated subgraph, possibly leading to exponential code duplication.

6.1.2 Symbolic Execution for Generation

In Section 6.1.1, the semantics of assembly were expressed in terms of control flow between basic blocks. This section now covers the symbolic execution of those individual blocks. The Haskell symbolic execution engine takes as input a data structure of type $\mathrm{scf}(B, E_F)$, which is formulated over basic blocks, and produces $\mathrm{scf}(\mathcal{P}(A_{\mathrm{SP}}), E_{\mathrm{SP}})$, $\mathscr{P}$ which is formulated over sets of assignments. It keeps track of all used memory regions, both the actual regions used



(a) CFG

```
Block 0;
If f0 Then Block 1;
    If f1 Then Block 2;
        If f2 Then Block 3
        Else Skip Fi
    Else Block 3 Fi
Else Skip Fi;
Block 4
```

(b) SCF

Figure 6.3: Example of code duplication

by instructions as well as merged regions, in order to supply those regions as part of an FMUC.

Generating Memory Region Relations

Because symbolic execution uses symbolic state, the relations of enclosure, separation, and overlap, defined in Section 4.2.1, must be determined for symbolic expressions. Unfortunately, there is no single solution, no one decision procedure, that can determine these properties for all symbolic expressions automatically.

As an example of the potential issues that can occur, take the completely symbolic regions $r_0 = [a_0, s_0]$ and $r_1 = [a_1, s_1]$. Without additional information, we cannot determine any relations for these regions. If they are *possibly* different then they must be treated as different regions, while if they *necessarily* overlap then they must be treated as a single merged region.

To deal with such symbolic issues, the three aforementioned relations of enclosure, separation, and overlap are formulated as SMT problems. The SMT formulations are negations of the equations presented in Definitions 4.3 and 4.4; the result states the property holds if the resultant problem is *unsatisfiable*. These SMT problems can be solved by Z3 [41] for a wide range of expressions over bitvectors using the `QF_UFBV` logic [102, 122]. Z3 is also used in this work for determining the sign of two values in the region merge algorithm, originally presented in Definition 4.5. Additionally, reads of overlapping regions may require merging and separation analysis as described in Section 4.2.1, so they also rely on Z3.

The result of evaluating the above SMT problems over all pairs of memory regions for a basic block, each region being given a unique ID, is two sets with element type $\mathbb{N} \times \mathbb{N}$, *enc* and *ovl*. Every element (i_0, i_1) in *enc* indicates that the region with ID i_0 is enclosed by the region with ID i_1. Every element (i_0, i_1) in *ovl* indicates that the two regions with those IDs overlap. Those two sets are the MRRs for the block, and using them as assumptions allows for efficient execution of the rewrite rules in Section 4.2.2.

6.1.3 Invariant Generation

Invariants, formalized as sets of assignments of the aforementioned type A_{SP}, are generated by starting from a precondition for the entry point of the function and *propagating* it throughout.

The initial precondition of the function as a whole is generated by including initial symbolic values for all registers that are read before they are written as well as all used memory regions that are not enclosed in another. The concrete initial value of the instruction pointer, `rip`, must also be included, and the (symbolic) address to return to after function completion must be indicated as stored on the stack. In Haskell, the conditions in question are represented as sets of assignments.

Example 6.3 (Initial invariant)**.** To reuse Example 4.6, its initial precondition would be:

$$\phi = \{\texttt{rip} \coloneqq \texttt{a0}, \texttt{rsp} \coloneqq rsp_0, [\texttt{rsp} - 8, 8] \coloneqq v_0, [\texttt{rsp}, 8] \coloneqq \texttt{ret\_addr}\}. \tag{6.4}$$

Propagation requires performing *substitution*, which is defined over assignments, state parts, and expressions, all with respect to invariant ϕ.

$$\begin{aligned}
\text{subst}(\phi, sp \coloneqq v) &= \text{subst}(\phi, sp) \coloneqq \text{subst}(\phi, v) && \text{(6.5a)}\\
\text{subst}(\phi, sp) &= \text{if } \exists v \cdot (sp, v) \in \phi \text{ then } v \text{ else } sp && \text{(6.5b)}\\
\text{subst}(\phi, e_0 \bigcirc e_1) &= \text{if } \exists v \cdot (e_0 \bigcirc e_1, v) \in \phi \text{ then } v \text{ else } \text{subst}(\phi, e_0) \bigcirc \text{subst}(\phi, e_1) && \text{(6.5c)}\\
\text{unary ops} &= \ldots\\
\text{ternary ops} &= \ldots
\end{aligned}$$

Algorithm 6.2 performs invariant propagation. Each block is modified by applying all applicable substitutions with respect to ϕ. Invariant ϕ is then modified based off of the semantics of the block. Treating α as the set of assignments in the block, ϕ is modified by taking the subset of substitutions where the substitutees are overwritten by α and combining them with the subset of substitutions that were completely unmodified by any assignment in α:

$$\text{post}(\phi, \alpha)^3 \equiv \{(v, e) \mid (v \coloneqq e \in \alpha \wedge (v, \_) \in \phi) \vee ((v, e) \in \phi \wedge (v, e) \text{ is unmodified by } \alpha)\}. \tag{6.6}$$

Example 6.4 (Invariant propagation)**.** Once again consider Example 4.6. Propagation of the initial precondition through the single basic block produces the following postcondition:

$$\begin{aligned}
\phi = \{&\texttt{rip} \coloneqq \texttt{ret_addr},\\
&\texttt{rsp} \coloneqq rsp_0 + 8,\\
&[rsp_0 - 8, 8] \coloneqq \texttt{0xAABBCCDD} \bullet \langle 31, 16\rangle v_0 \bullet \texttt{0xEEFF},\\
&[rsp_0, 8] \coloneqq \texttt{ret_addr}\}.
\end{aligned} \tag{6.7}$$

Algorithm 6.2 Invariant propagation

Require: Input is of type $\mathrm{scf}(\mathcal{P}(A_{\mathrm{SP}}), E_{\mathrm{SP}})$
Ensure: Output is a tuple of possibly-updated ϕ and SCF updated with current ϕ: $A_{\mathrm{SP}} \times \mathrm{scf}(\mathcal{P}(A_{\mathrm{SP}}), E_{\mathrm{SP}})$

function PROP(ϕ, `Block` α)
  $\phi' \leftarrow \mathrm{post}(\phi, \mathrm{subst}(\phi, \alpha))$
  return (ϕ', `Block` (α annotated with ϕ))
end function
function PROP(ϕ, α_0 `;` α_1)
  $(\phi', \alpha_0') \leftarrow$ PROP(ϕ, α_0)
  $(\phi'', \alpha_1') \leftarrow$ PROP(ϕ', α_1)
  return (ϕ'', α_0' `;` α_1')
end function
function PROP(ϕ, `If` f `Then` α_0 `Else` α_1 `Fi`)
  $(\phi_0, \alpha_0') \leftarrow$ PROP(ϕ, α_0)
  $(\phi_1, \alpha_1') \leftarrow$ PROP(ϕ, α_1)
  $\phi' \leftarrow \phi_0 \cap \phi_1$
  return (ϕ', `If` $\mathrm{subst}(\phi, f)$ `Then` α_0' `Else` α_1' `Fi`)
end function
function PROP(ϕ, `Loop` α `Pool`)
  $(\phi', \alpha') \leftarrow$ PROP(ϕ, α)
  if $\phi \subseteq \phi'$ **then**
    return (ϕ, `Loop` α' `Pool`)
  else
    return PROP($\phi \cap \phi'$, `Loop` α `Pool`)
  end if
end function
function PROP(ϕ, `Resume` *resumes*)
  for all $\alpha_i \in$ *resumes* **do**
    $(\phi_i', \alpha_i') \leftarrow$ PROP(ϕ, α_i)
  end for
  $\phi'' \leftarrow \bigcap \phi'$
  return (ϕ'', `Resume` ZIP(i, α'))
end function
function PROP(ϕ, α) ▷ Default case
  return (ϕ, α)
end function

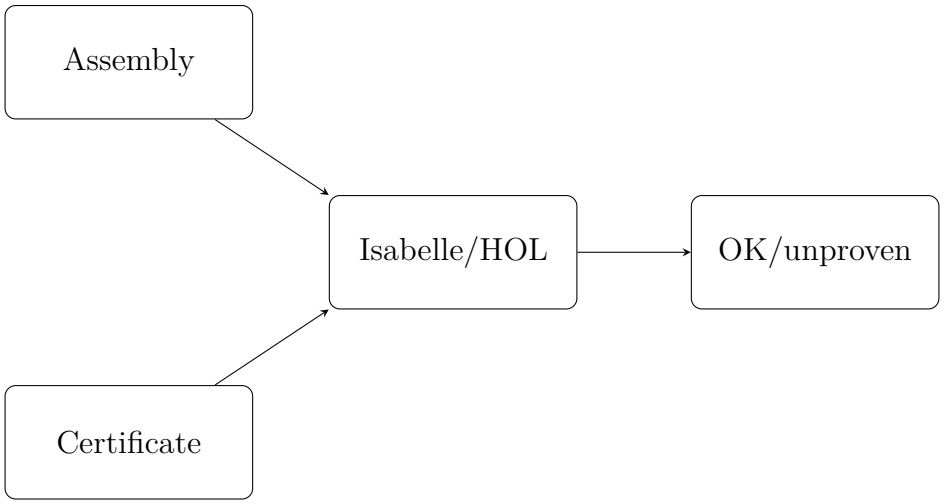


Figure 6.4: Overview of FMUC verification

Invariant propagation is straightforward for sequencing and if statements, with sequencing simply chaining invariant propagation and if statements producing an invariant that is the common result of propagating the initial invariant down both branches.

In contrast, a loop with body α requires continual propagation until the invariant ϕ stabilizes, possibly by becoming $\varnothing$. This stabilization is identified by checking if ϕ is a subset of its propagated self. If it is, then PROP returns the propagated ϕ and a new loop with the propagated body. Otherwise, the original loop is propagated again with the intersection of ϕ and its propagated self. This process effectively computes the loop invariant as the greatest subset of the initial invariant that is preserved by execution of the loop body. For loops that have multiple exits, each exit's resume is propagated with the invariant at the point of exit evaluation. In a similar fashion to the process for if statements, the invariants that result from individual resume propagation are intersected to produce a singular invariant for all resumes, which is then returned along with all of the propagated resumes.

6.2 FMUC Verification

This section presents verification of an FMUC as shown in Fig. 6.4, one of the primary contributions for this chapter as mentioned in its preamble. Both the FMUC and the original assembly are loaded into Isabelle/HOL, where the memory preservation theorem is then proven using the proof ingredients provided by the FMUC. By this method, which requires a step function that models the semantics of the assembly instructions and a process to apply it repeatedly, the FMUC's memory preservation Hoare triple can be verified.

[3]This is a different post from that used to identify CFG block children.

6.2.1 Syntactic Control Flow in Isabelle/HOL

As described previously, syntactic control flow is a representation of the control flow of a function in terms of syntactic structures such as basic blocks, loops, and if-then-else statements. While very similar to the SCF used when generating FMUCs, there are some modifications that must be made when the generated SCFs are to be loaded into Isabelle/HOL. These modifications are required due to subtle differences in the semantics of the generating tool versus the verifying tool, and are required to properly support the Hoare rules described in Section 6.2.4 below.

In the Isabelle/HOL representation, there are no `Breaks` or `Continues`; any occurrences of such statements are translated to `Skip`. This does mean that programs that cannot be easily transformed such that that translation does not modify the overall semantics are not easily handled in this framework. However, none of functions encountered in the case study presented in Section 6.4 had that issue, so it does not appear to be a significant drawback.

Additionally, loops in the Isabelle/HOL SCF syntax do rely on a explicit exit condition. This condition is simply the precondition of the entry block of the loop as generated using the methodology in Section 6.1.3.

Another important difference is that basic blocks in Isabelle take the form `Block` n a i, where n indicates the number of instructions in the block, a is the address of the last instruction in the block, and i is an ID that uniquely identifies the block in the current SCF. This style is used to assist with the symbolic execution methodology described in Section 6.2.2.

Finally, to properly handle function calls in the Isabelle/HOL syntax, the analyzed CFGs are preprocessed prior to performing extraction in order to isolate `call` instructions into their own basic blocks. These single-instruction blocks are then translated into `Call` f entries in the Isabelle/HOL SCF, where f is the textual label of the function called. This allows for proper matching with the Hoare rules presented below.

6.2.2 Symbolic Execution for Verification

Section 5.2.1 previously presented a formal symbolic execution engine based on the machine model described in Section 4.1. It provides a function runUntil that describes the symbolic execution of blocks in a control flow graph.

The formal function for block-level symbolic execution presented in this chapter, by contrast, is a *transition relation* formulated as

$$\textsc{symbExec} : \mathbb{N} \times W \times \mathbb{N} \times S \times S \to \mathbb{B}.$$

Its inputs are the number of instructions left to execute in the block, the address of the last instruction in the block, the block's ID, the current state σ, and an ending state σ'. Its result is true if and only if execution starting from the current instruction in state σ and running to the ending address can produce state σ'. The other arguments are used to ensure termination

and block matching. Undefined behavior, such as null-pointer dereferencing, is modeled by relating the state in which it occurs to any successor state supplied with it.

The internal step function has type $\textsc{step} : A \times \mathbb{N} \times S \to S$, with its first argument being an instruction, its second being the size of the instruction, and its third being the current state. The function returns the state after instruction execution, incrementing `rip` by the supplied size if it was not changed by a control-flow instruction instead.

The SYMBEXEC function is used internally by another transition relation, this one for the symbolic execution of entire SCFs: $\textsc{execScf} : \mathit{SCF} \times S \times S \to \mathbb{B}$. That function recurses through an SCF and checks SYMBEXEC on every block it finds, performing the necessary state transformations to deal with the semantics of the individual SCF components encountered. Any loops encountered are dealt with using an LFP construction. This means that, if there are any infinite loops present, the function will have no related successor states. The only matching state would be $\bot_{\text{NT}}$. Strictly speaking, EXECSCF is not actually executed when used in FMUC proofs; it exists to allow proving the correctness of the Hoare rules shown below in Section 6.2.4.

Unlike the symbolic execution for generation, this symbolic execution methodology is implemented fully in Isabelle/HOL, meaning that every rewrite rule has been formally proven correct.

6.2.3 Per-Block Verification

The verification methodology presented here occurs by first verifying the functionality of each basic block in the corresponding function. This is done for each block by proving the lemma shown below, using the EXECSCF function from the previous section. To do this, however, a formal notion of *memory preservation* with respect to state changes is required.

Definition 6.5 (Memory preservation with respect to state changes)**.** The set of memory regions M' characterizes memory preservation with respect to the change from some state σ to some other state σ' if and only if every byte outside of the regions in M' is the same in both states. The addresses of those bytes are represented by the variable a.

This is formally expressed as:

$$\text{preserve}(M', \sigma, \sigma') \equiv \forall a \cdot (\forall r \in M' \cdot [a, 1] \bowtie r) \longrightarrow \sigma : *[a, 1] = \sigma' : *[a, 1]. \tag{6.8}$$

Using Definition 6.5, each block gets a lemma of the form

$$P(\sigma) \longrightarrow \textsc{symbExec}(n, a, i, \sigma, \sigma') \wedge Q(\sigma') \wedge \text{preserve}(M(\sigma), \sigma, \sigma'). \tag{6.9}$$

Note that M is a state-dependent function. Every generated version of Eq. (6.9) is discharged with an Isabelle/HOL proof method written in Eisbach [83], Isabelle's proof automation language. For each block, the method takes the block-related proof ingredients from the FMUC and runs symbolic execution to prove the postcondition and thus establish memory preservation for the block. The open variables P, Q, n, a, i, and M are all provided by

the FMUC. No user interaction is required outside of cases where semantics for specific instructions are unavailable or the Isabelle libraries in use do not have the right simplification lemmas for automatic reasoning. Those cases are rare and become rarer as more relevant lemmas are developed, so for basic blocks, the proof is essentially automated.

6.2.4 Function Body Verification

While symbolic execution works well to establish memory preservation on the level of basic blocks, the goal of this verification effort is to formally establish memory preservation on the function level. This section describes that process, which occurs after the individual blocks have had their semantics and memory preservation derived and relies on Hoare logic (described in Section 3.1.2).

Hoare Rules

The Hoare triple formulation used for this work, $\{P\}f\{Q;M\}$, resembles traditional Hoare triples a bit more than the version from Chapter 5, as rather than a halting condition it takes a syntactic representation of the program, an SCF. Unlike traditional Hoare triples, however, it also explicitly contains the set of memory regions, M, that contain the areas of memory read and written by the program the SCF encodes. Syntactic structure is required because Hoare logic is a syntax-guided approach.

Definition 6.6 (Hoare triple for SCF)**.**

$$\{P\}f\{Q;M\} \equiv \forall\sigma\ \sigma' \cdot P(\sigma) \wedge \textsc{execScf}(f,\sigma,\sigma') \longrightarrow Q(\sigma') \wedge \text{preserve}(M,\sigma,\sigma') \quad (6.10)$$

The above definition states the following: if precondition P holds on the initial state σ and σ' would be the result of symbolically executing the SCF f, postcondition Q will hold on the produced state and any values stored in the regions of memory outside set M remain unchanged.

While Definition 6.6 focuses on the regions written to, the regions read must also be included as symbolic execution relies on those regions being included. Without them, proofs that require symbolically executing the related instructions will not complete.

The Introduction Rule

This rule, depicted in Fig. 6.5a, is the rule that ties the per-block verification to the function-body verification. The first assumption requires the symbolic execution method be run from a universally quantified initial symbolic state σ that satisfies the precondition. As long as any resulting state σ' satisfies the postcondition Q, the set of memory regions M generated for the block should be correct.

$$\frac{\begin{array}{l}\forall \sigma\ \sigma' \cdot P(\sigma) \longrightarrow \\ \quad \textsc{symbExec}(n, a, i, \sigma, \sigma') \wedge \qquad M' = \{r \mid \exists \sigma \cdot P(\sigma) \wedge r \in M(\sigma)\} \\ \quad Q(\sigma') \wedge \text{preserve}(M(\sigma), \sigma, \sigma')\end{array}}{\{P\}\texttt{Block}\ n\ a\ i\{Q;M'\}}$$

(a) Introduction rule

$$\frac{\{P\}f\{Q;M_1\} \qquad \{Q\}g\{R;M_2\} \qquad M = M_1 \cup M_2}{\{P\}f\ ;\ g\{R;M\}}$$

(b) Sequence rule

$$\frac{\{P \wedge B\}f\{Q_1;M_1\} \qquad \{P \wedge \neg B\}g\{Q_2;M_2\} \qquad Q_1 \vee Q_2 \longrightarrow Q \qquad M = M_1 \cup M_2}{\{P\}\texttt{If}\ B\ \texttt{Then}\ f\ \texttt{Else}\ g\ \texttt{Fi}\{Q;M\}}$$

(c) Conditional rule

$$\frac{\{I \wedge B\}f\{I';M\} \qquad I' \longrightarrow I \qquad I \wedge \neg b \longrightarrow Q}{\{I\}\texttt{While}\ B\ \texttt{DO}\ f\ \texttt{OD}\{Q;M\}} \qquad \frac{M = \varnothing \qquad P \longrightarrow Q}{\{P\}\texttt{Skip}\{Q;M\}}$$

(d) While rule (e) Skip rule

$$\frac{\forall 0 \le j \le n \cdot \{P\}a_j\{Q_j;M_j\} \qquad (\bigvee_{0 \le j \le n} Q_j) \longrightarrow Q \qquad M = \bigcup_{0 \le j \le n} M_j}{\{P\}\texttt{Resume}\{(i_0, a_0), \ldots, (i_n, a_n)\}\{Q;M\}}$$

(f) Resume rule

$$\frac{P \longrightarrow P' \qquad \{P'\}b\{Q\}M}{\{P\}b\{Q\}M} \qquad \frac{Q' \longrightarrow Q \qquad \{P\}b\{Q';M\}}{\{P\}b\{Q;M\}}$$

(g) Precondition weakening (h) Postcondition strengthening

Figure 6.5: Hoare rules for memory preservation

The second assumption is required because of an important subtlety: the regions generated in the FMUC are state dependent. As previously stated, the M for a block is actually a function based on the block's initial state: its regions depend on the values stored in memory. However, it makes no sense to express the regions used by individual blocks within a larger function in terms of their own individual initial state alone. It would be unsound for regions that depend on values calculated in the middle of the function to be expressed solely in terms of the initial state. As such, the Hoare triples are defined over a state-independent set of memory regions, M'. That set is obtained for each block by taking the generated state-dependent set of memory regions and applying that set to any state that satisfies the current invariant.

The Other Rules

While the introduction rule for basic blocks is the ultimate target of our Hoare rule application process, the rest of the rules are required to decompose the syntax above the level of blocks. The remainder of Fig. 6.5 describes those additional rules. The Sequence, Conditional, and Resume rules are straightforward: their ultimate memory region sets are the unions of the region sets of their constituents. Note that the sequence rule is sound only because the memory predicates are independent of state as discussed in Section 6.2.3.

The while rule is based on a loop invariant, I. If the memory preservation of one iteration of loop body f is constrained to set of memory regions M, then so is the memory preservation of every other iteration. This may sound counterintuitive, so consider a simple C-like loop that starts from $i = 0$ and iterates while $i < 10$. The body of this example loop contains single-byte array assignment operations along the lines of $a[i] = v$. Verification of the loop requires the loop invariant $I(\sigma) = i(\sigma) < 10$. The FMUC of the loop body will have, as a state-dependent set of memory regions, $M(\sigma) = \{[a + i(\sigma), 1]\}$, which is a single region of one byte. If the Hoare logic introduction rule were to be applied to the block that is the body of the loop, the result would be as follows:

$$M' = \{r \mid \exists\sigma \cdot I(\sigma) \wedge r \in M(\sigma)\} \tag{6.11a}$$
$$= \{r \mid \exists\sigma \cdot i(\sigma) < 10 \wedge r = [a + i(\sigma), 1]\} \tag{6.11b}$$
$$= \{[a', 1] \mid a \leq a' \leq a + 10\} \tag{6.11c}$$

The set M' contains the regions of memory used by the entire loop, not just one iteration. This is because the introduction rule applies the state-dependent set of memory regions to any state that satisfies the invariant. Thus, the strength of the generated invariants directly influences the tightness of the overapproximation of memory preservation and of memory usage as a whole. A weaker invariant, such as $i < 20$, would result in a larger set of memory regions by relaxing the constraints on symbolic addresses and, for other situations, symbolic region sizes.

Listing 6.1: VCG step method

```
1 method vcg_step =
2   ((rule htriples)+, rule blocks)+,
3   (simp add: pred_logic Ps Qs)?,
4   (((auto simp: eq_def)[])+)?
```

Listing 6.2: Main VCG method

```
1 method vcg uses scf =
2   subst scf,
3   vcg_step+
```

Verification Condition Generation

The VCG presented here is a set of Eisbach proof methods, the entry point of which is shown in Listing 6.2. It is designed to automatically apply the proper Hoare logic rules as much as possible via the `vcg_step` method in Listing 6.1, driven by the formal SCF provided by the FMUC.

Internally, `vcg_step` repeatedly applies one of the Hoare rules from Fig. 6.5 (excluding the While, strengthening, and weakening rules) to the current state of the SCF until no more rules can be applied. At that point, it assumes that the introduction rule has been applied, resulting in a block goal being generated, and attempts to discharge that goal using one of the lemmas generated for Section 6.2.3. This process is repeated until no more of the restricted set of rules can be applied or the last rule application resulted in a non-block goal. At that point, Line 3 cleans up any preconditions and postconditions in the current goal. The last step, Line 4, then tries to eliminate as many goals as it can, one at a time, with Isabelle's basic `auto` method. If there are no loops present in the SCF under consideration, this method will complete the proof without any need for user interaction.

In the case where loops are present, the VCG provides an alternate `vcg_while` method, shown in Listing 6.4 that relies on the loop rule presented in Fig. 6.5d. That loop rule is structured such that the majority of work required to support the loops is identifying the

Listing 6.3: Alternate step method for `Resume` clauses

```
1 method vcg_step' =
2   (rule htriples)+,
3   simp,
4   ((rule htriples)+, rule blocks)+,
5   (simp add: pred_logic Ps Qs)?,
6   (((auto simp: eq_def)[])+)?
```

Listing 6.4: VCG method for loops

```
1  method vcg_while for P :: state_pred =
2    ((rule htriples)+)?,
3    rule HTriple_weaken[where P=P],
4    simp add: pred_logic Ps Qs,
5    rule HTriple_while,
6    vcg_step+,
7    (simp add: pred_logic Ps Qs)+,
8    (
9      (vcg_step' | vcg_step)+,
10     (simp+)?
11   )?
```

preconditions of their exit blocks and then supplying their disjunction to `vcg_while`. This method relies on application of the weakening rule presented in Fig. 6.5g on Line 3 to show that the postcondition of the block before entry implies the loop invariant.

The method `vcg_step'`, used within `vcg_while`, is provided for those cases where a loop has multiple exit points. A `Resume` statement will be present in such cases, and the process of rule application and simplification must occur in a slightly different order. On occasion, there will also be a loop that has a single exit point but gets a `Resume` statement anyway due to how the control flow extraction algorithm is set up. The process of dealing with such statements is roughly the same, however.

After application of `vcg_while`, nested loops and those with multiple exit points may require additional applications of condition simplifying or plain `simp` usage around further applications of `vcg_step`. Nothing beyond that should be necessary.

Without exception, each of the proofs we produced could be finished using standard, off-the-shelf Isabelle/HOL methods, though finishing them was not always an automatic process. The part that is usually the most involved, defining the invariants (as seen in the previous chapter) is taken care of by the FMUC generation. This leaves dealing with loops, particularly ones with multiple exit points, as the biggest challenge for most situations.

6.2.5 Composition

In order to achieve a scalable verification methodology, it must support some form of compositionality.

Consider the body of an already-verified function f with the following Hoare triple:

$$\{P_f\}f\{Q_f;M_f\}.$$

In order to reuse that function's proof in a compositional fashion, it is treated as a black box. Now consider the assembly of a function g that calls f:

$$\frac{\{P_f\}f\{Q_f;M_f\} \qquad P \longrightarrow P_f \wedge P_{\text{sep}} \qquad \forall s\ s' \cdot \begin{array}{l}(\text{preserve}(M_f, s, s') \wedge \\ P_{\text{sep}}(s)) \longrightarrow P_{\text{sep}}(s')\end{array} \qquad Q_f \wedge P_{\text{sep}} \longrightarrow Q}{\{P\}\texttt{Call}f\{Q;M_f\}}$$

Figure 6.6: Frame rule for composition of memory usage

```
a0: push rbp
a1: call f
a2: pop  rbp
a3: ret
```

P and Q are the pre- and postconditions just before executing `call` and just after it returns. P contains the equality $*[rsp_0^g - 8, 8] = rbp_0^g$, expressing that g has pushed the frame pointer `rbp` into its own local stack frame. The ultimate postcondition of g expresses that the callee-saved register `rbp` is properly restored: `rbp` $= rbp_0^g$. That operation is indeed performed by `pop rbp`. In order to prove proper restoration of `rbp`, a proof that function f did not overwrite any byte in the region $[rsp_0^g - 8, 8]$ is required. The proof must also show that f does not overwrite region $[rsp_0^g, 8]$, which stores the address g returns to. That proof would be specific to this particular instance of calling f.

Of course, g may not be the only function that calls f. It may even be called multiple times by the same function. Every call has specific requirements on which memory regions must be preserved, based on the calling context. Thus, to be able to verify function f once but reuse its proof for each call, the proof must at least contain an overapproximation of the memory written to by function f. This is exactly what *separation logic* [75, 94, 104] requires. As described in Section 1.1.2, separation logic provides a *frame rule* for compositional reasoning. Informally, this rule states that, if a program can be confined to a certain part of state, properties of that program will carry over when the program is used as part of a bigger system.

In order to achieve that same behavior specifically for memory preservation verification, we developed the frame rule presented in Fig. 6.6. This rule is used to prove that the memory usage of a caller function g is equal to the memory it itself uses, *plus* the memory used by function f. It must have the following four assumptions. First, that f has been verified for memory preservation, with M_f denoting the memory regions f uses. Second, that precondition P can be split up into two parts: precondition P_f, required to verify f, and a separate part P_{sep}. The separate part is specific to the specific call of the function where the frame rule is applied. In the example above, P_{sep} must contain the equality $[rsp_0^g - 8, 8] = rbp_0^g$. Third, the correctness of M_f, the set of memory regions, should suffice to prove that P_{sep} is preserved. This effectively means that, for the above example, M_f should not overlap with the two regions of g. Fourth and finally, P_{sep} and Q_f should imply postcondition Q.

In practice, many functions will not be part of the assembly code under verification, such as dynamic library or system calls. Those cases necessitate generating the assumptions required

to proceed with verification. The following box notation supports those cases:

$$\{P\}\boxed{\texttt{f}}\{Q;M_f\} \equiv \exists P_f\ Q_f\ P_{\text{sep}} \cdot \text{all four assumptions of the frame rule are satisfied.} \quad (6.12)$$

This assumption informally expresses that function f has been verified. Its memory usage M_f is assumed to suffice to prove that the states that satisfy precondition P lead to the states that satisfy postcondition Q.

6.3 Full Example

This section presents an execution of the entire toolchain on the example given in Fig. 6.7a as a summary of Sections 6.1 and 6.2. The C code is provided solely for presentation, as the only inputs to the FMUC generation are the assembly created by disassembling the corresponding binary and a basic configuration file indicating which functions to analyze. Figure 6.7b presents the generated SCF. The example has one loop, which starts at instruction address `0x120`. Zooming in on `Block 123e->1244`, we see from Fig. 6.7d that the FMUC provides 13 regions, of which four are shown. Region r_0 stores the return address while region r_1 depends on the segment register **`fs`** and stores the canary value used to detect stack buffer overflows [35]. Region r_2 is based on the pointer passed as the second argument to the function, and region r_3 is part of the stack frame. The generated MRRs assume that all these regions are separate.

The precondition assigned to `Block 123e->1244` is effectively a loop invariant (see Fig. 6.7d). The frame pointer **`rbp`** is equal to the original stack pointer minus eight. Register **`rdi`** has not been touched. Some of the more complex assignments are also shown, such as the current value of the stack pointer. In total, the loop invariant provides information on 11 registers and 12 memory locations for this basic block. The process of verification shows that, for any state satisfying this invariant, executing one iteration of the loop body will result in a state that again satisfies the loop invariant. The only interactions required in verifying the FMUC of the entire function are: 1. showing that the postcondition after `Block 1149->120b` implies the loop invariant, and 2. showing that, in the case of a break, the postcondition of the loop body implies the precondition of `Block 1246->1249`. This amounts to two manually written lines of Isabelle proof code.

To demonstrate the black-box functionality from Section 6.2.5, `is_even` was treated as external to the example's analysis. This resulted in the generation of an assumption stating that the memory usage of `is_even` suffices to show that the invariant for the call site (instruction address `124b`) implies the invariant for the instruction address immediately following, `1250`. This means that $M_{\texttt{is\_even}}$ is assumed to not overlap with regions a through d, among other things.

Figure 6.7f shows the sole manual effort required to prove the FMUC for this function. All it involves is calling the proper predefined Eisbach proof methods, previously described in Section 6.2.4. The first proof method applied is `vcg`, which initializes the proof with the function's SCF, applies Hoare rules, and proves correctness of all memory preservation up

```
1  int main(int argc, char* argv[])
   {
2    int* a = (int*)argv;
3    int* b = (int*)(argv + 4);
4    *(int*)(argv + 2) = *a + *b;
5    *(char*)argv = 'a';
6
7    int array[argc];
8    for (int i = 0; i < argc; i
       ++)
9      array[i] = argv[i][0] *
         2;
10
11   if (is_even(argc))
12     return array[argc];
13   return array[0];
14 }
```

(a) C code

```
Block 1149->120b;
Loop
  Block 123e->1244;
  If SF ≠ OF Then Block 120d->123a
    Else Break Fi
Pool;
Block 1246->1249;
Block 124b->124b;
Block 1250->1252;
If ZF Then Block 1263->1267
  Else Block 1254->1261 Fi;
Block 1269->1279;
If ZF Then Block 1280->1285
  Else Block 127b->127b Fi
```

(b) Syntactic control flow for the assembly

$$M = \{r_0 = [rsp_0, 8], r_1 = [fs_0 + 40, 8], r_2 = [rsi_0 + 36, 4], r_3 = [rsp_0 - 8, 8], \dots\}$$
$$MRR = \{r_0, r_1, r_2, r_3, \dots, r_{12}\} \text{ are separate}$$

(c) Some memory regions and their relations for block `123e->1244`

$$\begin{aligned} P_{\texttt{123e}}(\sigma) = \quad & \texttt{rip} = \texttt{0x123e} \\ & \texttt{rbp} = rsp_0 - 8 \\ & \texttt{rdi} = rdi_0 \\ & \texttt{rsp} = rsp_0 - (88 + 16 * ((15 + 4 * \text{sextend}(\langle 31, 0\rangle rdi_0)/16)) \\ & *[rsp_0 - 40, 8] = rsp_0 - (85 + 16 * ((15 + 4 * \text{sextend}(\langle 31, 0\rangle rdi_0))/16)) \gg 2 \ll 2 \\ & *[rsp_0 - 48, 8] = \text{sextend}(\langle 31, 0\rangle rdi_0) - 1 \\ & *[rsp_0 - 56, 8] = rsi_0 + 32 \end{aligned}$$

(d) Invariant for address `0x123e` (only 7 out of 23 equalities shown)

$$\{P_{\texttt{124b}}\}\boxed{\texttt{is\_even}}\{P_{\texttt{1250}}; M_{\texttt{is\_even}}\}$$

(e) Assumption due to call of `is_even`

```
apply (vcg scf: main_scf)
apply (vcg_while <P_123e || P_1246>)
apply vcg_step+
```

(f) Isabelle proof code (manual effort)

Figure 6.7: Application of entire methodology on example

until the loop. Following that, the proof method for dealing with loops, `vcg_while`, is applied with the invariant formed from the disjunction of the precondition of the loop's entry block and the precondition of the loop's exit block, both manually identified from the generated SCF. As the last manual step, `vcg_step` is called repeatedly to verify the remainder of the function.

Finally, note that, without any assumptions, the function could overwrite its own return address at various places. The MRRs are strong enough to exclude this scenario. Those relations thus form the preconditions under which a return-address exploit is impossible. For example, they assume that regions a and c are separate. This means that the address stored in argument `argv` (mapped to rsi_0 on the assembly level) is not allowed to point to a region within the stack frame of the `main` function.

6.4 Application: Xen Project

The Xen Project [29] is a mature, widely-used virtual machine monitor, also known as a *hypervisor*. Hypervisors provide a method of managing multiple VMs (called domains in the Xen documentation) on a physical host.

The Xen hypervisor is a suitable case study because of its security relevance and its complex build process involving real production code. Security is a significant issue in environments where hypervisors are used, such as the Amazon Elastic Compute Cloud, Rackspace Cloud, and many other cloud service providers. For example, when one or more hosts support guest domains for any number of distinct users, ensuring isolation of the domains is important.

The Xen build process produces multiple binaries that contain functions not present in the Xen source itself. This is due to the inclusion of external static libraries and programs. Xen version 4.12 was compiled with GCC 8.2 via the standard Xen build process. This build process uses various optimization levels, ranging from `O1` to `O3`. The version of `objdump` used to disassemble the compiled binaries was 2.31.1.

The verification effort presented here covered three of the binaries produced by the Xen build process: `xenstore`, `xen-cpuid`, and `qemu-img-xen`. The `xenstore` binary is involved in the functionality of XenStore[4], a hierarchical data structure shared amongst all Xen domains. This sharing allows for the possibility of inter-domain communication, though in general XenStore is intended for simple configuration information. A smaller program than `xenstore`, `xen-cpuid` provides functionality similar to that of the `cpuid` utility[5]. This utility queries the underlying processors and displays information about the features they support. Such functionality is important for Xen as it supports migrating domains between processors with different variants of the same ISA [125]. The third binary used, `qemu-img-xen`, consists of over three hundred functions that are not present in the Xen source code. It provides some of the functionality of *Quick Emulator* (QEMU). QEMU is a free, open-source emulator[6].

[4] https://wiki.xen.org/wiki/XenStore
[5] https://linux.die.net/man/1/cpuid
[6] https://www.qemu.org/

Binaries	Function Count	Instruction Count	Loops	Manual Lines of Proof
`xenstore`	2/6	100	0	6
`xen-cpuid`	2/3	210	2	39
`qemu-img-xen`	247/343	11 942	64	1002
Total	251/352	12 252	65	1047

Table 6.1: Verified Xen Functions

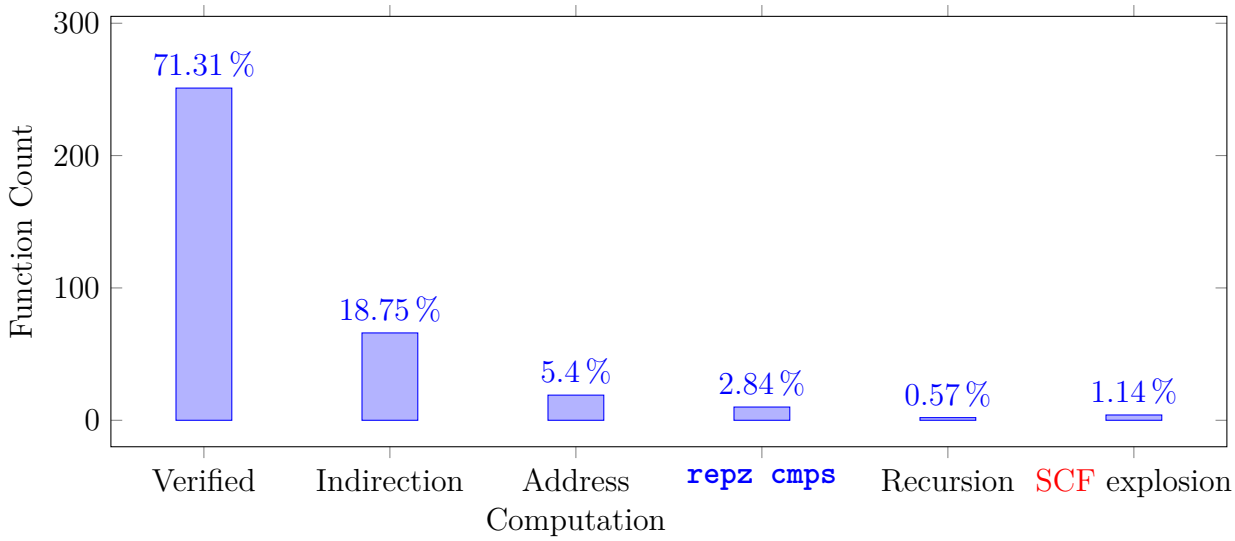


Figure 6.8: Analyzed Xen functions compared to unverified features

Xen uses it to emulate device models, which provide interfaces for hardware storage.

This methodology is currently capable of dealing with 71 % of the functions present in the aforementioned binaries (see Fig. 6.8). The supported features include (nested) loops, subcalls, variable argument lists, jumps into other function bodies, string instructions with the `rep` prefix, and SIMD instructions. There is no particular limit on function size. The average number of instructions per function analyzed is 49. Some of the functions analyzed have over 300 instructions and over 100 basic blocks.

There are five categories of features not currently supported. The first and most common, previously mentioned in Section 6.1.1, is *indirection*, accounting for 19 %. Indirection involves a call or jump instruction that loads the target address from a register or memory location rather than using a static value. Switch statements and certain uses of `goto` are the most common causes of indirect jumps. Indirect calls generally result from usage of function pointers. For example, the `main` functions of all three verified binaries used switch statements in loops in the process of parsing command line options. These statements introduced indirect branches.

The second category involves issues related to generating the MRRs. This step requires solving linear arithmetic over symbolically computed addresses. Sometimes, addresses are

computed using a combination of arithmetic operators with bitwise logical operators. In some of these cases, our translation to Z3 does not produce an answer. As an example, function `qcow_open` uses the rotate-left function to compute an address. As another example, function `AES_set_encrypt_key` produces addresses that are obtained via combinations of bit-shifting, bit masking, and `xor`-ing. For these cases, separation and enclosure relations cannot be generated.

The instruction `repz cmps` is currently not supported for technical reasons. It is the assembly equivalent of the function `strncmp`, but instead writes its result to a flag. Various other string-related instructions with the `rep` prefix are supported, however.

Functions with *recursion*, a minority in systems code, are also not supported as they are not well-suited to automation in this framework. The two recursive functions encountered in the analyzed Xen binaries both perform file-system-like tasks. Functions `do_chmod` and `do_ls` are similar to the permission-setting `chmod` utility and the directory-displaying `ls`, respectively.

The final category is functions whose SCF explodes. The issue can occur when the pattern in Fig. 6.3 shows up extensively or when while loops have multiple entry points.

Table 6.1 provides an overview of the verification effort. The table shows the absolute counts of functions verified as well as the total number of instructions for those functions. Alongside that information is the number of functions with loops that were verified and how many manual lines of proof were required in total. The vast majority of those manual proof lines were related to the loop count. Meanwhile, a comparison with those functions not verified can be found in Fig. 6.8.

6.5 Summary

This chapter presented an approach to formal verification of memory preservation for functions in a disassembled program. As in the previous chapter, the memory usage reported for those functions is an overapproximation of the memory that would be used when actually executing the assembly code. The approach automatically generates an FMUC that includes 1. a set of memory regions read from and written to, 2. preconditions necessary for formal verification, 3. postconditions that express sanity constraints over the function (the return address has not been overwritten, callee-saved registers are restored, etc.), and 4. proof ingredients. The certificate is loaded into a theorem prover, where it can be verified. The proof ingredients, combined with custom proof methods, provide a large degree of automation. They deal with memory aliasing and provide both the control flow of the function as well as invariants.

The approach was applied to three binaries produced by the Xen hypervisor build process. They contain nested loops, complex data structures, variadic functions, and both internal and external function calls. A certificate could be generated and verified for 71 % of the functions from those binaries. The amount of user interaction was roughly 85 lines of proof code per 1000 lines of assembly code. The greatest issue was indirect branching, which could be found in 19 %of the functions examined.

Chapter 7

Conclusions

Certain properties, such as memory preservation, can only be proven on the assembly level. This is due to memory preservation requiring a concrete representation of memory. Unfortunately, assembly-level verification is a fundamentally harder problem than source-level verification due to the low level of abstraction. However, it can also produce highly reliable claims over software. By eliminating the need to trust the compiler and the semantics of whatever source language the program was written in, you can drastically decrease the TCB in use.

Additionally, the property of memory preservation cannot be determined automatically under all circumstances. It is an undecidable property. Because of this and the overheads that ITP can have, we designed *semi-automated* methodologies for that purpose. Those methodologies are briefly revisited below.

7.1 Contributions Revisited

This dissertation presented two methods for proving memory preservation, control-flow-driven verification and syntax-driven verification. Both approaches rely on the same symbolic execution model and memory-related rewrite rules documented in Chapter 4, but differ in several major aspects.

Technically they also both use Hoare triples, but only Chapter 6 uses proper Hoare rules. Chapter 5 uses a modified style that takes a halting condition H instead of a syntactic construct in the middle.

7.1.1 Control-Flow-Driven Verification

This methodology uses a Floyd-style approach [51] with automatically-selected cutpoints. It is very similar to the work of Matthews et al. [85], but with a focus on memory preservation specifically. The rewrite rules over memory accesses from Section 4.2.2 result in additional

VCs that would not be present in their framework. Those VCs require time-consuming word arithmetic when the appropriate preconditions/assumptions are not present. The preconditions/assumptions simply establish separation and enclosure relations for the necessary memory regions.

This methodology was applied to 63 functions extracted from the HermitCore unikernel library, plus 12 optimized versions, resulting in more than 2379 assembly instructions verified.

7.1.2 Syntax-Driven Verification

Rather than using a more general CFG to guide the verification, a more structured SCF is extracted from the assembly under test. This SCF is used as one of the generated FMUC's proof ingredients. The other proof ingredients are the generated memory regions, MRRs, and block conditions. With the invariant generation as it currently is, the only user interaction required under normal circumstances is weakening the condition for a loop entry block by merging it with the condition for all of the loop's exit blocks.

This methodology was applied to 251 functions from the Xen Project binaries examined, 71 % of the total functions from the examined binaries. Ultimately, 12 252 instructions were covered with only 1047 manual lines of proof required.

7.2 Proposed Post-Preliminary Exam Work

As a formal property, memory preservation has been proven to never miss any memory regions written to, assuming the correctness of the semantics and model it is applied to [15, 127]. Put another way, however, this means that the methodology *must* be conservative. If it cannot make a determination about the usage status of some region of memory, it must assume that that region is used. It must *overapproximate*. It does not matter if the cause was an underdeveloped state or too large of one to easily reason about.

There are two potential ways to reduce that overapproximation detailed below.

7.2.1 Strengthen Invariants

In order to enhance automation, we currently generate very weak invariants. While this works reasonably well, being able to generate stronger invariants would be advantageous. Stronger invariants mean stricter memory preservation proofs.

One possible way to generate stronger invariants would be to use *abstract interpretation* [33, 34]. The methodology used by Crab (`https://github.com/seahorn/crab`) for loop invariant restriction may prove useful for this purpose [56].

Abstract interpretation is a form of approximation in which the possible values for some variables in a program are constrained to, for example, a polyhedral range. It functions

somewhat like a partial execution of the program under test in order to determine semantic information. It is sound and complete, meaning there would be no drawbacks to using it aside from perhaps additional execution time.

7.2.2 Model a More Realistic Memory Model

Most applications do not run in isolation. Their behavior is limited by the kernel of whatever OS is in use, and that includes limits on the amount of memory they are allowed to use.

In particular, process and thread stacks are limited by how they are laid out in (virtual) memory, and on top of that most modern OS kernels put limits on stack size as sanity checks. The kernel limits are generally configurable, both at compile time as well as at runtime, but can require privileged access. Properly modeling those restrictions would potentially require formulating a more in-depth memory model as the stack limits that are changed at runtime come in two forms. There is a *soft* limit on stack size that unprivileged users can modify, but there is also a *hard* limit that requires root access to modify.

Additional features that would be desirable would be the ability to treat memory as *allocated* and *deallocated*. On modern systems, this is usually handled on the page level, meaning via virtual memory management. Modeling virtual memory would likely not be a good idea, as we are currently focusing on userspace analysis.

The main restriction that would be interesting to model, however, would be the restriction of addresses to their lower 48 bits for actual addressing, with the remaining 16 bits being equal in value to the 48th bit.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity: Principles, Implementations, and Applications". In: *ACM Transactions on Information and System Security* 13.1 (Oct. 2009), 4:1–4:40. ISSN: 1094-9224. DOI: 10.1145/1609956.1609960.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 71:1–71:31. ISSN: 2475-1421. DOI: 10.1145/3290384.

[3] Clark Barrett and Sergey Berezin. "CVC Lite: A New Implementation of the Cooperating Validity Checker". In: *Computer Aided Verification*. Proceedings of the 16th International Conference. (Boston, MA, USA, July 13–17, 2004). Ed. by Rajeev Alur and Doron A. Peled. Lecture Notes in Computer Science 3114. Berlin Heidelberg: Springer-Verlag, 2004, pp. 515–518. DOI: 10.1007/978-3-540-27813-9_49.

[4] Clark Barrett and Cesare Tinelli. "CVC3". In: *Computer Aided Verification*. Proceedings of the 19th International Conference. (Berlin, Germany, July 3–7, 2007). Ed. by Werner Damm and Holger Hermanns. Lecture Notes in Computer Science 4590. Berlin Heidelberg: Springer-Verlag, 2007, pp. 298–302.

[5] Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. "A High Assurance Virtualization Platform for ARMv8". In: *2016 European Conference on Networks and Communications*. (Athens, Greece, June 27–30, 2016). Piscataway, NJ, US: IEEE, Sept. 8, 2016, pp. 210–214. DOI: 10.1109/EuCNC.2016.7561034.

[6] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *Fast Software Encryption*. 12th International Workshop Revised Selected Papers. (Paris, France, Feb. 21–23, 2005). Ed. by Henri Gilbert and Helena Handschuh. Lecture Notes in Computer Science 3557. Berlin Heidelberg: Springer-Verlag, 2005, pp. 32–49. ISBN: 978-3-540-31669-5.

[7] Yves Bertot and Pierre Castéran. "* Proof by Reflection". In: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Ed. by Wilfried Brauer, Grzegorz Rozenberg, and Arto Salomaa. Texts in Theoretical

Computer Science. Berlin Heidelberg: Springer-Verlag, 2004, pp. 433–448. ISBN: 978-3-662-07964-5. DOI: 10.1007/978-3-662-07964-5_16.

[8] William R. Bevier. "A Verified Operating System Kernel". PhD thesis. Oct. 1987.

[9] William R. Bevier. "Kit and the Short Stack". In: *Journal of Automated Reasoning* 5.4 (Dec. 1989), pp. 519–530. ISSN: 1573-0670. DOI: doi.org/10.1007/BF00243135.

[10] William R. Bevier. "Kit: A Study in Operating System Verification". In: *IEEE Transactions on Software Engineering* 15.11 (Nov. 1989), pp. 1382–1396. ISSN: 0098-5589. DOI: 10.1109/32.41331.

[11] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. "An Approach to Systems Verification". In: *Journal of Automated Reasoning* 5.4 (Dec. 1, 1989), pp. 411–428. ISSN: 1573-0670. DOI: 10.1007/BF00243131.

[12] *BitBlaze: Binary Analysis for Computer Security*. July 16, 2013. URL: http://bitblaze.cs.berkeley.edu/ (visited on 08/22/2019).

[13] Sandrine Blazy and Xavier Leroy. "Mechanized Semantics for the Clight Subset of the C Language". In: *Journal of Automated Reasoning* 43.3 (Oct. 1, 2009), pp. 263–288. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9148-3.

[14] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures". In: *High Performance Computer Architecture*. IEEE 19th International Symposium. (Shenzhen, China, Feb. 23–27, 2013). Piscataway, NJ, US: IEEE, June 3, 2013, pp. 1–12. DOI: 10.1109/HPCA.2013.6522302.

[15] Joshua A. Bockenek, Freek Verbeek, Peter Lammich, and Binoy Ravindran. "Formal Verification of Memory Preservation of x86-64 Binaries". In: *Computer Safety, Reliability and Security*. Proceedings of the 38th International Conference. (Turku, Finland, Sept. 11–13, 2019). Ed. by Alexander Romanovsky, Elena Troubitsyna, and Friedemann Bitsch. Vol. 11698. Lecture Notes in Computer Science. Berlin Heidelberg: Springer-Verlag.

[16] Joshua A. Bockenek, Freek Verbeek, Peter Lammich, and Binoy Ravindran. *Formal Verification of Memory Preservation of x86-64 Binaries. SAFECOMP 2019 supplemental material*. Version 4. July 24, 2019. DOI: 10.6084/m9.figshare.7356110.v4.

[17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code". In: *26th USENIX Security Symposium*. (Vancouver, BC, Canada). USENIX Association, Aug. 2017, pp. 917–934. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[18] Jonathan Bowen. "Standard Microprocessor Programming Cards". In: *Microprocessors and Microsystems* 9.6 (July–Aug. 1985), pp. 274–289. DOI: 10.1016/0141-9331(85)90116-4.

[19] Robert S. Boyer and J Strother Moore. *A Computational Logic*. New York, NY, USA: Academic Press, Inc., 1979. ISBN: 978-0-12-122950-4. DOI: 10.1016/C2013-0-10411-4.

[20] Robert S. Boyer and Yuan Yu. "Automated Proofs of Object Code for a Widely Used Microprocessor". In: *Journal of the ACM* 43.1 (Jan. 1996). Ed. by Frank Thomson Leighton, pp. 166–192. DOI: 10.1145/227595.227603.

[21] Jörg Brauer, Bastian Schlich, Thomas Reinbacher, and Stefan Kowalewski. "Stack Bounds Analysis for Microcontroller Assembly Code". In: *Embedded Systems Security*. Proceedings of the 4th Workshop. (Grenoble, France, Oct. 15, 2009). New York, NY, USA: ACM, 2009, 5:1–5:9. ISBN: 978-1-60558-700-4. DOI: 10.1145/1631716.1631721.

[22] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification*. Proceedings of the 23rd International Conference. (Snowbird, UT, USA, July 14–20, 2011). Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science 6806. Berlin Heidelberg: Springer-Verlag, 2011, pp. 463–469. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_37.

[23] Ricky W. Butler. *What is Formal Methods?* Apr. 10, 2016. URL: https://shemesh.larc.nasa.gov/fm/fm-what.html (visited on 08/16/2019).

[24] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NASA Formal Methods*. Proceedings of the Third International Symposium. (Pasadena, CA, USA, Apr. 18–20, 2011). Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Lecture Notes in Computer Science 6617. Berlin Heidelberg: Springer-Verlag, 2011, pp. 459–465. ISBN: 978-3-642-20398-5. DOI: 10.1007/978-3-642-20398-5_33. URL: https://fbinfer.com/.

[25] Bernard A. Carré, I. M. O'Neill, D. L. Clutterbuck, and C. W. Debney. "SPADE–The Southampton Program Analysis and Development Environment". In: *Software Engineering Environments*. Peter Peregrinus, Ltd. Stevenage. 1986.

[26] Arthur Charguéraud. "Characteristic Formulae for the Verification of Imperative Programs". In: *Functional Programming*. Proceedings of the 16th ACM SIGPLAN International Conference. (Tokyo, Japan, Sept. 19–21, 2011). New York, NY, USA: ACM, 2011, pp. 418–430. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034828.

[27] Arthur Charguéraud and François Pottier. "Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation". In: *Interactive Theorem Proving*. Proceedings of the 6th International Conference. (Nanjing, China, Aug. 24–27, 2015). Ed. by Christian Urban and Xingyuan Zhang. Lecture Notes in Computer Science 9236. Cham: Springer International Publishing, Aug. 19, 2015, pp. 137–153. ISBN: 978-3-319-22102-1.

[28] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. "Toward Compositional Verification of Interruptible OS Kernels and Device Drivers". In: *Journal of Automated Reasoning* 61.1 (June 1, 2018). Ed. by Jeremy Avigad, Gerwin Klein, and Lawrence C. Paulson, pp. 141–189. ISSN: 1573-0670. DOI: 10.1007/s10817-017-9446-0.

[29] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. London, England, UK: Pearson Education, 2008.

[30] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. Cambridge, MA, USA: MIT Press, Dec. 6, 2013.

[31] D. L. Clutterbuck and Bernard A. Carré. "The Verification of Low-Level Code". In: *Software Engineering journaltitle* 3.3 (May 1988), pp. 97–111. ISSN: 0268-6961. DOI: 10.1049/sej.1988.0012.

[32] David Costanzo, Zhong Shao, and Ronghui Gu. "End-to-end Verification of Information-flow Security for C and Assembly Programs". In: *Programming Language Design and Implementation*. Proceedings of the 37th ACM SIGPLAN Conference. (Santa Barbara, CA, USA, June 13–17, 2016). New York, NY, USA: ACM, 2016, pp. 648–664. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908100.

[33] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages*. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium. (Los Angeles, CA, USA). New York, NY, USA: ACM, Jan. 19–19, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[34] Patrick Cousot and Radhia Cousot. "Static Determination of Dynamic Properties of Programs". In: *Programming*. Proceedings of the 2nd International Symposium. (Paris, France, Apr. 13–15, 1976). Ed. by B. Robinet. Dunod.

[35] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *USENIX Security Symposium*. Proceedings of the 7th. (San Antonio, TX, USA, Jan. 26–29, 1998). Vol. 98. 1998, pp. 63–78.

[36] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. "Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel". In: *Computer & Communications Security*. Proceedings of the 2013 ACM SIGSAC Conference. (Berlin, Germany, Nov. 4–8, 2013). New York, NY, USA: ACM, Nov. 2013, pp. 223–234. DOI: 10.1145/2508859.2516702.

[37] Mads Dam, Roberto Guanciale, and Hamed Nemati. "Machine Code Verification of a Tiny ARM Hypervisor". In: *Trustworthy Embedded Devices*. Proceedings of the 3rd International Workshop. (Berlin, Germany, Nov. 4, 2013). New York, NY, USA: ACM, Nov. 2013, pp. 3–12. ISBN: 978-1-4503-2486-1. DOI: 10.1145/2517300.2517302.

[38] Werner Damm and Holger Hermanns, eds. *Computer Aided Verification*. Proceedings of the 19th International Conference. (Berlin, Germany, July 3–7, 2007). Lecture Notes in Computer Science 4590. Berlin Heidelberg: Springer-Verlag, 2007.

[39] Jeremy Dawson. "Isabelle Theories for Machine Words". In: *Electronic Notes in Theoretical Computer Science* 250.1 (Sept. 1, 2009), pp. 55–70. DOI: 10.1016/j.entcs.2009.08.005.

[40] Jeremy Dawson, Paul Graunke, Brian Huffman, Gerwin Klein, and John Matthews. *Machine Words in Isabelle/HOL*. Version 2018. Aug. 15, 2018. URL: https://isabelle.in.tum.de/website-Isabelle2018/dist/library/HOL/HOL-Word/document.pdf (visited on 08/22/2019).

[41] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 14th International Conference. (Budapest, Hungary, Mar. 29–Apr. 6, 2008). Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science 4963. Berlin Heidelberg: Springer-Verlag, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[42] "Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics". In: *Automated Reasoning Workshop 2018*. (Cambridge, UK). Apr. 2018. URL: https://alastairreid.github.io/papers/ARW_18/ (visited on 08/25/2019).

[43] Morris J. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/-Counter Mode (GCM) and GMAC*. Tech. rep. 800-38D. Gaithersburg, MD, USA, Nov. 28, 2007. URL: https://www.nist.gov/publications/recommendation-block-cipher-modes-operation-galoiscounter-mode-gcm-and-gmac?pub_id=51288 (visited on 08/26/2019).

[44] Manuel Eberl, Gerwin Klein, Tobias Nipkow, Larry Paulson, and René Thiemann, eds. *Archive of Formal Proofs*. Aug. 19, 2019. URL: https://www.isa-afp.org/ (visited on 08/22/2019).

[45] David Evans and David Larochelle. "Improving Security using Extensible Lightweight Static Analysis". In: *IEEE Software* 19.1 (Jan.–Feb. 2002), pp. 42–51. ISSN: 0740-7459. DOI: 10.1109/52.976940.

[46] *F*: A Higher-Order Effectful Language Designed for Program Verification*. URL: https://www.fstar-lang.org/ (visited on 08/23/2019).

[47] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. *Modular Verification of Assembly Code with Stack-Based Control Abstractions*. Tech. rep. YALEU/DCS/TR-1336. New Haven, CT, USA: Department of Computer Science, Yale University, Nov. 2005. 24 pp. URL: http://flint.cs.yale.edu/publications/sbca.html.

[48] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. "Modular Verification of Assembly Code with Stack-Based Control Abstractions". In: *Programming Language Design and Implementation*. Proceedings of the 27th ACM SIGPLAN Conference. (Ottawa, Ontario, Canada, June 11–14, 2006). Vol. 41. PLDI 2006 6. New York, NY, USA: ACM, June 2006, pp. 401–414. DOI: 10.1145/1133981.1134028.

[49] Edward A. Feustel. "On the Advantages of Tagged Architecture". In: *IEEE Transactions on Computers* C-22.7 (July 1973), pp. 644–656. DOI: 10.1109/TC.1973.5009130.

[50] Edward A. Feustel. "The Rice Research Computer—A Tagged Architecture". In: *Proceedings of the 1972 Spring Joint Computer Conference*. (Atlantic City, NJ, USA, May 16–18, 1972). Vol. 40. New York, NY, USA: ACM, May 1972, pp. 369–377. DOI: 10.1145/1478873.1478920.

[51] Robert W. Floyd. "Assigning Meanings to Programs". In: *Mathematical Aspects of Computer Science* 19.1 (1967), pp. 19–32.

[52] Anthony Fox. "Improved Tool Support for Machine-Code Decompilation in HOL4". In: *Interactive Theorem Proving*. Proceedings of the 6th International Conference. (Nanjing, China, Aug. 24–27, 2015). Ed. by Christian Urban and Xingyuan Zhang. Lecture Notes in Computer Science 9236. Springer-Verlag. Cham, Aug. 19, 2015, pp. 187–202. DOI: 10.1007/978-3-319-22102-1_12.

[53] Anthony Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *Interactive Theorem Proving*. Proceedings of the First International Conference. (Edinburgh, UK, July 11–14, 2010). Ed. by Matt Kaufmann and Lawrence C. Paulson. Lecture Notes in Computer Science 6172. Berlin Heidelberg: Springer-Verlag, 2010, pp. 243–258. ISBN: 978-3-642-14052-5. DOI: 10.1007/978-3-642-14052-5_18.

[54] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. "A Verified, Efficient Embedding of a Verifiable Assembly Language". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 63:1–63:30. ISSN: 2475-1421. DOI: 10.1145/3290376.

[55] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification*. Proceedings of the 19th International Conference. (Berlin, Germany, July 3–7, 2007). Ed. by Werner Damm and Holger Hermanns. Lecture Notes in Computer Science 4590. Berlin Heidelberg: Springer-Verlag, 2007, pp. 519–531. ISBN: 978-3-540-73368-3. DOI: 10.1007/978-3-540-73368-3_52.

[56] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. "An Abstract Domain of Uninterpreted Functions". In: *Verification, Model Checking, and Abstract Interpretation*. Proceedings of the 17th International Conference. (St. Petersburg, FL, USA, Jan. 17–19, 2016). Ed. by Barbara Jobstmann and K. Rustan M. Leino. Lecture Notes in Computer Science 9583. Berlin Heidelberg: Springer-Verlag, Dec. 25, 2015, pp. 85–103. ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_4.

[57] Shilpi Goel. "Formal Verification of Application and System Programs Based on a Validated x86 ISA Model". PhD thesis. Dec. 2016. HDL: 2152/46437.

[58] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls". In: *2014 Formal Methods in Computer-Aided Design*. (Lausanne, Switzerland, Oct. 21–24, 2014). Ed. by Koen Claessen and Viktor Kuncak. Piscataway, NJ, US: IEEE, Dec. 18, 2014, pp. 91–98. DOI: 10.1109/FMCAD.2014.6987600.

[59] Joseph Amadee Goguen and José Meseguer. “Security Policies and Security Models”. In: *Security and Privacy*. Proceedings of the 1982 IEEE Symposium. (Oakland, CA, USA, Apr. 26–28, 1982). Piscataway, NJ, US: IEEE, Dec. 15, 2014, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[60] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. *Report on Gypsy 2.05*. Tech. rep. CLI-I. Oct. 1986.

[61] David Greenaway, June Andronick, and Gerwin Klein. “Bridging the Gap: Automatic Verified Abstraction of C”. In: *Interactive Theorem Proving*. Proceedings of the Third International Conference. (Princeton, NJ, USA, Aug. 13–15, 2012). Ed. by Lennart Beringer and Amy Felty. Lecture Notes in Computer Science 7406. Berlin Heidelberg: Springer-Verlag, Aug. 2012, pp. 99–115. DOI: 10.1007/978-3-642-32347-8_8.

[62] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. “Verified Characteristic Formulae for CakeML”. In: *Programming Languages and Systems*. Proceedings of the 26th European Symposium on Programming. (Uppsala, Sweden, Apr. 22–29, 2017). Ed. by Hongseok Yang. Lecture Notes in Computer Science 10201. Berlin Heidelberg: Springer-Verlag, Mar. 19, 2017, pp. 584–610. ISBN: 978-3-662-54434-1.

[63] *Heartbleed Bug*. Apr. 29, 2014. URL: http://heartbleed.com/ (visited on 08/22/2019).

[64] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stratified Synthesis: Automatically Learning the x86-64 Instruction Set”. In: *Programming Language Design and Implementation*. Proceedings of the 37th ACM SIGPLAN Conference. (Santa Barbara, CA, USA, June 13–17, 2016). New York, NY, USA: ACM, 2016, pp. 237–250. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908121.

[65] Charles Antony Richard Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969). Ed. by M. Stuart Lynn, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[66] Charles Antony Richard Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (Aug. 1978). Ed. by Robert L. Ashenhurst, pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[67] David Hovemeyer and William Pugh. “Finding Bugs is Easy”. In: *ACM SIGPLAN Notices* 39.12 (Dec. 2004), pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: http://findbugs.sourceforge.net/.

[68] Warren A. Hunt Jr. “Microprocessor Design Verification”. In: *Journal of Automated Reasoning* 5.4 (Dec. 1989), pp. 429–460. ISSN: 1573-0670. DOI: 10.1007/BF00243132.

[69] *Intel 64 and IA-32 Architectures. Software Developer’s Manual*. 4 vols. Intel Corporation, May 21, 2019. URL: https://software.intel.com/en-us/articles/intel-sdm (visited on 08/25/2019).

[70] Tariq Jamil. “RISC versus CISC. Why less is more”. In: *IEEE Potentials* 14.3 (Aug.–Sept. 1995), pp. 13–16. ISSN: 0278-6648. DOI: 10.1109/45.464688.

[71] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Berlin Heidelberg: Kluwer Academic Publishers, 2000.

[72] James C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[73] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–2:70. DOI: 10.1145/2560537.

[74] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal Verification of an OS Kernel". In: *Operating Systems Principles*. Proceedings of the 22nd ACM SIGOPS Symposium. (Big Sky, MT, USA, Oct. 11–14, 2009). New York, NY, USA: ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[75] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic". In: *Programming*. Proceedings of the 26th European Symposium. (Uppsala, Sweden, Apr. 22–29, 2017). Ed. by Hongseok Yang. Lecture Notes in Computer Science 10201. Berlin Heidelberg: Springer-Verlag, Mar. 19, 2017, pp. 696–723.

[76] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. "Automating Mimicry Attacks Using Static Binary Analysis". In: *USENIX Security Symposium*. Vol. 14. USENIX Association. 2005, pp. 161–176.

[77] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages*. Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium. (San Diego, CA, USA, Jan. 22–24, 2014). New York, NY, USA: ACM, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/ (visited on 08/25/2019).

[78] Stefan Lankes, Simon Pickartz, and Jens Breitbart. "HermitCore: A Unikernel for Extreme Scale Computing". In: *Runtime and Operating Systems for Supercomputers*. Proceedings of the 6th International Workshop. (Kyoto, Japan, June 1, 2016). New York, NY, USA: ACM, 2016, 4:1–4:8. ISBN: 978-1-4503-4387-9. DOI: 10.1145/2931088.2931093.

[79] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert - A Formally Verified Optimizing Compiler". In: *Embedded Real Time Software and Systems*. Proceedings of the 8th European Congress. ERTS 2016. SEE. Toulouse, France: HAL, Jan. 2016. HAL: hal-01238879.

[80] *LLVM Language Reference Manual – LLVM 10 documentation*. Functions. Aug. 23, 2019. URL: https://llvm.org/docs/LangRef.html#functions (visited on 08/24/2019).

[81] *LLVM Language Reference Manual – LLVM 10 documentation*. Terminators. Aug. 23, 2019. URL: https://llvm.org/docs/LangRef.html#terminators (visited on 08/24/2019).

[82] Anil Madhavapeddy and David J. Scott. "Unikernels: The Rise of the Virtual Library Operating System". In: *Communications of the ACM* 57.1 (Jan. 2014). Ed. by Moshe Y. Vardi, pp. 61–69. ISSN: 0001-0782.

[83] Daniel Matichuk, Toby Murray, and Makarius Wenzel. "Eisbach: A Proof Method Language for Isabelle". In: *Journal of Automated Reasoning* 56.3 (Mar. 1, 2016), pp. 261–282. DOI: 10.1007/s10817-015-9360-2.

[84] Daniel Matichuk, Makarius Wenzel, and Toby Murray. *The Eisbach User Manual*. Version 2018. Aug. 15, 2018.

[85] John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon. "Verification Condition Generation via Theorem Proving". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Proceedings of the 13th International Conference. (Phnom Penh, Cambodia, Nov. 13–17, 2006). Ed. by Miki Hermann and Andrei Voronkov. Lecture Notes in Computer Science 4246. Berlin Heidelberg: Springer-Verlag, 2006, pp. 362–376. DOI: 10.1007/11916277_25.

[86] J Strother Moore. "A Mechanically Verified Language Implementation". In: *Journal of Automated Reasoning* 5.4 (Dec. 1989), pp. 461–492. ISSN: 1573-0670. DOI: 10.1007/BF00243133.

[87] J Strother Moore. *Piton: A Verified Assembly Level Language*. Tech. rep. 1988.

[88] Magnus O. Myreen and Michael J. C. Gordon. "Hoare Logic for Realistically Modelled Machine Code". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Proceedings of the 13th International Conference. (Braga, Portugal, Mar. 24–Apr. 1, 2007). Ed. by Orna Grumberg and Michael Huth. Lecture Notes in Computer Science 4424. Berlin Heidelberg: Springer-Verlag, 2007, pp. 568–582. DOI: 10.1007/978-3-540-71209-1_44.

[89] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. "Decompilation into Logic—Improved". In: *2012 Formal Methods in Computer-Aided Design*. (Cambridge, UK, Oct. 22–25, 2012). Piscataway, NJ, US: IEEE, Feb. 19, 2013, pp. 78–81.

[90] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. "Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic". In: *2008 Formal Methods in Computer-Aided Design*. (Portland, OR, USA, Nov. 17–20, 2008). Piscataway, NJ, US: IEEE, Nov. 25, 2008, pp. 1–8. DOI: 10.1109/FMCAD.2008.ECP.24.

[91] George C. Necula. "Proof-Carrying Code". In: *Principles of Programming Languages*. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium. (Paris, France, Jan. 15–17, 1997). New York, NY, USA: ACM, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[92] Nguyen Anh Quynh. *Capstone: Next-Gen Disassembly Framework*. Aug. 7, 2014. URL: https://www.capstone-engine.org/ (visited on 07/27/2019).

[93] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Berlin Heidelberg: Springer-Verlag, Jan. 2002. DOI: 10.1007/3-540-45949-9. URL: https://isabelle.in.tum.de/.

[94] Peter O'Hearn, John Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic*. Proceedings of the 15th International Workshop. (Paris, France, Sept. 10–13, 2001). Ed. by Laurent Fribourg. Lecture Notes in Computer Science 2142. Berlin Heidelberg: Springer-Verlag, Aug. 30, 2001, pp. 1–19. ISBN: 978-3-540-44802-0. DOI: doi.org/10.1007/3-540-44802-0.

[95] I. M. O'Neill, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. "The Formal Verification of Safety-critical Assembly Code". In: *IFAC Symposium on Safety of Computer Control Systems 1988*. (Fulda, FRG, Nov. 9–11, 1988). Vol. 21. SAFECOMP '88 18. Amsterdam, The Netherlands: Elsevier Ltd., Nov. 1988, pp. 115–120. DOI: 10.1016/S1474-6670(17)54540-1.

[96] Jan Obdržálek and Marek Trtík. "Efficient Loop Navigation for Symbolic Execution". In: *Automated Technology for Verification and Analysis*. Proceedings of the 9th International Symposium. (Taipei, Taiwan, Oct. 11–14, 2001). Ed. by Tevfik Bultan and Pao-Ann Hsiung. Lecture Notes in Computer Science 6996. Berlin Heidelberg: Springer-Verlag, 2011, pp. 453–462. DOI: doi.org/10.1007/978-3-642-24372-1_34.

[97] Martin Ouimet and Kristina Lundqvist. *Formal Software Verification: Model Checking and Theorem Proving. Embedded Systems Laboratory Technical Report ESL-TIK-00214*. Tech. rep. Cambridge, MA, USA, 2008.

[98] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics*. Proceedings of the 22nd International Conference. (Munich, Germany, Aug. 17–20, 2009). Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Lecture Notes in Computer Science 5674. Berlin Heidelberg: Springer-Verlag, 2009, pp. 391–407. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_27.

[99] Scott Owens, Susmit Sarkar, and Peter Sewell. *A Better x86 Memory Model: x86-TSO*. Extended Version. Tech. rep. UCAM-CL-TR-745. Version 1746. University of Cambridge, Computer Laboratory, Mar. 2009. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-745.html (visited on 08/26/2019).

[100] Mark Probst. "Proper Tail Recursion in C". Institute of Computer Languages, TU Wien, Feb. 2, 2001.

[101] *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019). ISSN: 2475-1421.

[102] Silvio Ranise, Cesare Tinelli, and Clark Barrett. *FixedSizeBitVectors | SMT-LIB The Satisfiability Modulo Theories Library*. The SMT-LIB Initiative. June 13, 2017. URL: http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml (visited on 08/22/2019).

[103] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. "End-to-End Verification of Processors with ISA-Formal". In: *Computer Aided Verification*. Proceedings Part II of the 28th International Conference. (Toronto, Ontario, Canada, July 17–23, 2016). Ed. by Swarat Chaudhuri and Azadeh Farzan. Lecture Notes in Computer Science book 9780. Cham: Springer International Publishing, July 13, 2016, pp. 42–58. ISBN: 978-3-319-41540-6.

[104] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Logic in Computer Science*. Proceedings of the 17th Annual IEEE Symposium. (Copenhagen, Denmark, July 22–25, 2002). Ed. by Anne Jacobs. Piscataway, NJ, US: IEEE, Nov. 7, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[105] Henry Gordon Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (Mar. 1953), pp. 358–366. ISSN: 00029947. DOI: 10.2307/1990888. JSTOR: 1990888.

[106] Ian Roessle, Freek Verbeek, and Binoy Ravindran. "Formally Verified Big Step Semantics out of x86-64 Binaries". In: *Certified Programs and Proofs*. Proceedings of the 8th ACM SIGPLAN International Conference. (Cascais, Portugal, Jan. 14–15, 2019). New York, NY, USA: ACM, 2019, pp. 181–195. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294102.

[107] John M. Rushby. "Design and Verification of Secure Systems". In: *Operating Systems Principles*. Proceedings of the Eighth ACM Symposium. (Pacific Grove, CA, USA, Dec. 14–16, 1981). SOSP '81. New York, NY, USA: ACM, 1981, pp. 12–21. ISBN: 0-89791-062-1. DOI: 10.1145/800216.806586.

[108] John M. Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Tech. rep. Computer Science Laboratory at SRI International, 1992.

[109] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. "Loop-Extended Symbolic Execution on Binary Programs". In: *Software Testing and Analysis*. Proceedings of the Eighteenth International Symposium. (Chicago, IL, USA, July 19–23, 2009). New York, NY, USA: ACM, 2009, pp. 225–236. ISBN: 978-1-60558-338-9.

[110] Bastian Schlich. "Model Checking of Software for Microcontrollers". PhD thesis. 2008.

[111] Bastian Schlich, Falk Salewski, and Stefan Kowalewski. "Applying Model Checking to an Automotive Microcontroller Application". In: *Industrial Embedded Systems*. 2007 International Symposium. (Lisbon, Portugal, July 4–6, 2007). Piscataway, NJ, US: IEEE, Sept. 4, 2007, pp. 209–216. DOI: 10.1109/SIES.2007.4297337.

[112] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors". In: *Communications of the ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[113] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Programming Language Design and Implementation*. Proceedings of the 34th ACM SIGPLAN Conference. (Seattle, WA, USA, June 16–22, 2013). New York, NY, USA: ACM, 2013, pp. 471–482. ISBN: 978-1-4503-2014-6.

[114] Jianqi Shi, Jifeng He, Huibiao Zhu, Huixing Fang, Yanhong Huang, and Xiaoxian Zhang. "ORIENTAIS: Formal Verified OSEK/VDX Real-Time Operating System". In: *Engineering of Complex Computer Systems*. Proceedings of the IEEE 17th International Conference. (Paris, France, July 18–20, 2012). Piscataway, NJ, US: IEEE, Sept. 13, 2012, pp. 293–301. ISBN: 978-2-9541-8100-4.

[115] Jianqi Shi, Longfei Zhu, Huixing Fang, Jian Guo, Huibiao Zhu, and Xin Ye. "xBIL – A Hardware Resource Oriented Binary Intermediate Language". In: *Engineering of Complex Computer Systems*. Proceedings of the IEEE 17th International Conference. (Paris, France, July 18–20, 2012). Piscataway, NJ, US: IEEE, Sept. 13, 2012, pp. 211–219.

[116] Jianqi Shi, Longfei Zhu, Yanhong Huang, Jian Guo, Huibiao Zhu, Huixing Fang, and Xin Ye. "Binary Code Level Verification for Interrupt Safety Properties of Real-Time Operating System". In: *Theoretical Aspects of Software Engineering*. Proceedings of the Sixth International Symposium. (Beijing, China, July 4–6, 2012). Piscataway, NJ, US: IEEE, Aug. 16, 2012, pp. 223–226. DOI: 10.1109/TASE.2012.46.

[117] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy*. (San Jose, CA, USA, May 22–26, 2016). Piscataway, NJ, US: IEEE, Aug. 18, 2016. DOI: 10.1109/SP.2016.17. URL: https://angr.io/.

[118] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics*. Proceedings of the 21st International Conference. (Montreal, Canada, Aug. 18–21, 2008). Ed. by Otmane Ait Mohamed, César Muñoz, and Tahar Sofiène. Lecture Notes in Computer Science 5170. Berlin Heidelberg: Springer-Verlag, 2008, pp. 28–32. DOI: 10.1007/978-3-540-71067-7_6.

[119] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Information Systems Security*. Proceedings of the 4th International Conference. (Hyderabad, India, Dec. 16–20, 2008). Ed. by R. Sekar and Arun K. Pujari. Lecture Notes in Computer Science 5352. Keynote invited paper. Berlin Heidelberg: Springer-Verlag, Dec. 2008. ISBN: 978-3-540-89861-0.

[120] Jiaqi Tan, Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan. "AUSPICE: Automatic Safety Property Verification for Unmodified Executables". In: *Verified Software: Theories, Tools and Experiments*. Revised Selected Papers from the 7th International Conference. (San Francisco, CA, USA, July 18–19, 2015). Ed. by Arie Gurfinkel and

Sanjit A. Seshia. Lecture Notes in Computer Science 9593. Cham: Springer-Verlag, July 2015, pp. 202–222. DOI: 10.1007/978-3-319-29613-5_12.

[121] Adam Thornton. *A Brief History of the Rice Computer 1959-1971*. Feb. 24, 2008. URL: https://web.archive.org/web/20080224035658/http://www.princeton.edu/~adam/R1/r1rpt.html (visited on 08/22/2019).

[122] Cesare Tinelli. *QF_UFBV | SMT-LIB The Satisfiability Modulo Theories Library*. The SMT-LIB Initiative. July 18, 2017. URL: http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_UFBV (visited on 08/22/2019).

[123] Frank Tip. "A Survey of Program Slicing Techniques". In: *Journal of Programming Languages* 3 (1995), pp. 121–189.

[124] "Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture". In: *Formal Methods in Computer-Aided Design*. Proceedings of the 16th International Conference. (Mountain View, CA, USA, Oct. 3–6, 2016). Austin, TX, USA: FMCAD Inc, Oct. 2016, pp. 161–168. ISBN: 978-0-9835678-6-8. URL: https://alastairreid.github.io/papers/FMCAD_16/.

[125] Hans van Kranenburg. *Xen CPUID masking*. Aug. 18, 2016. URL: https://tech.mendix.com/linux/2016/08/18/xen-cpuid-masking/ (visited on 07/03/2019).

[126] Freek Verbeek, Joshua A. Bockenek, Abhijith Bharadwaj, Ian Roessle, and Binoy Ravindran. "Establishing a Refinement Relation between Binaries and Abstract Code". In: *Formal Methods and Models for System Design*. Proceedings of the 17th ACM-IEEE International Conference. (San Diego, CA, USA, Oct. 9–11, 2019). MEMOCODE'19. 2019.

[127] Freek Verbeek, Joshua A. Bockenek, and Binoy Ravindran. "Highly Automated Formal Proofs over Memory Usage of Assembly Code". Under review. 2019.

[128] Fish Wang and Yan Shoshitaishvili. "Angr – The Next Generation of Binary Analysis". In: *2017 IEEE Cybersecurity Development*. (Cambridge, MA, USA, Sept. 24–26, 2017). Piscataway, NJ, US: IEEE, Oct. 23, 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14.

[129] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. "Ramblr: Making Reassembly Great Again". In: *Network and Distributed System Security*. Proceedings of the 24th Annual Symposium. 2017.

[130] Mark Weiser. "Program Slicing". In: *Software Engineering*. Proceedings of the 5th International Conference. (San Diego, CA, USA, Mar. 9–12, 1981). ICSE '81. Piscataway, NJ, US: IEEE, 1981, pp. 439–449. ISBN: 0-89791-146-6. ACM: 800078.802557.

[131] Makarius Wenzel. "Isabelle/Isar—A Generic Framework for Human-Readable Proof Documents". In: *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec* 10.23 (Jan. 2007), pp. 277–298.

[132] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. Version 2018. Aug. 15, 2018.

[133] *X86 psABI · hjl-tools/x86-psABI Wiki*. URL: https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI (visited on 08/25/2019).

[134] William D. Young. "A mechanically verified code generator". In: *Journal of Automated Reasoning* 5.4 (Dec. 1989), pp. 493–518. ISSN: 1573-0670. DOI: 10.1007/BF00243134.

[135] Yuan Yu. "Automated Proofs of Object Code for a Widely Used Microprocessor". PhD thesis. University of Texas at Austin, Oct. 5, 1993.

Index