Low-Level Static Analysis for Memory Usage and Control Flow Recovery

Joshua Alexander Bockenek

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Freek Verbeek
Paul Plassmann
Michael S. Hsiao
Changhee Jung

3 February 2023
Blacksburg, Virginia

Keywords: Formal Verification, x86-64 Assembly, Interactive Theorem Proving, Static Binary Analysis, Memory Usage, Control Flow Recovery, Exception Handling

Low-Level Static Analysis for Memory Usage and Control Flow Recovery

Joshua Alexander Bockenek

(ABSTRACT)

Formal characterization of *the memory used by a program* is an important basis for security analyses, compositional verification, and identification of noninterference. However, soundly proving memory usage requires operating on the assembly level due to the semantic gap between high-level languages and the code that processors actually execute. Automated methods, such as model checking, would not be able to handle many interesting functions due to the undecidability of memory usage. Fully-interactive methods do not scale well either.

Sound *control flow recovery* is also important for binary decompilation, verification, patching, and security analysis. It lifts raw unstructured data into a form that allows reasoning over behavior and semantics. However, doing so requires interpreting the behavior of the program when indirect or dynamic control flow exists, creating a recursive dependency.

This dissertation tackles the first property with two contributions that perform proof generation combined with interactive theorem proving in a *semi-automated manner*: an untrusted tool extracts as much information as it can from the functions under test and then generates all the necessary proofs to be completed in a theorem prover. The first, Floyd-style approach still requires significant manual effort but provides good flexibility and ensures no paths are analyzed more than once. In contrast, the second, Hoare-style approach sacrifices some flexibility and avoidance of repeated path evaluation in order to achieve much greater automation. However, neither approach can handle the dynamic control flow caused by indirect branching.

The second property is handled by the second set of contributions of this dissertation. These two contributions provide *fully-automated* methods of recovering control flow from binaries even in the presence of indirect branching. When such dynamic control flow cannot be overapproximatively resolved, it is clearly noted in the resultant output. In the first approach to control flow recovery, a structured memory representation allows for general analysis of control flow in the presence of indirection, gaining scalability by utilizing context-free function analysis. It supports various aliasing conditions via the usage of nondeterminism, with multiple output states potentially being produced from a given input state. The second approach adds function context and abstract interpretation-inspired modeling of the C++ exception handling (EH) application binary interface (ABI), allowing for the discovery of previously-unknown paths while maintaining or increasing automation.

Low-Level Static Analysis for Memory Usage and Control Flow Recovery

Joshua Alexander Bockenek

(GENERAL AUDIENCE ABSTRACT)

Modern computer programs are so complicated that individual humans cannot manually check all but the smallest programs to make sure they are correct and secure. This is even worse if you want to reduce the trusted computing base (TCB), the stuff that you have to assume is working right in order to say a program will execute correctly. The TCB includes your computer itself, but also whatever tools were used to take the programs written by programmers and transform them into a form suitable for running on a computer. Such tools are often called *compilers*.

One method of reducing the TCB is to examine the lowest-level representation of that program, the assembly or even machine code that is actually run by your computer. This poses unique challenges, because operating on such a low level means you do not have a lot of the structure that a more abstract, higher-level representation provides. Also, sometimes you want to *formally* state things about a program's behavior; that is, say things about what it does with a high degree of confidence based on mathematical principles. You may also want to *verify* that one or more of those statements are true. If you want to be detailed about that behavior, you may need to know all of the chunks, or *regions*, in random-access memory (RAM) that are used by that program. RAM, henceforth referred to as just "memory", is your computer's first place of storage for the information used by running programs. This is distinct from long-term storage devices like hard disk drives (HDDs) or solid-state drives (SSDs), which programs do not normally have direct access to.

Unfortunately, there is no one single approach that can automatically determine with absolute certainty for all cases the exact regions of memory that are read or written. This is called *undecidability*, and means that you need to *approximate* those memory regions a lot of the time if you want to have a significant degree of automation. An *underapproximation*, an approach that only gives you some of the regions, is not useful for formal statements as it might miss out on some behavior; it is *unsound*. This means that you need an *overapproximation*, an approach that is guaranteed to give you *at least* the regions read or written.

Therefore, the first contribution of this dissertation is a preliminary approach to such an overapproximation. This approach is based on the work of Robert L. Floyd, focusing on the direct *control flow*[1] in an individual *function*.[2] It still requires a lot of user effort, including having to manually specify the regions in memory that were possibly used and do a lot of work to prove that those regions are (overapproximatively) correct, so our tests were limited in scope.

The second contribution automated a lot of the manual work done for the first approach. It is based on the work of Charles Antony Richard Hoare, who developed a verification approach

[1] where the steps of a program go

[2] structured program component

focusing on the *syntax*[3] of programs. This contribution produces what we call *formal memory usage certificates (FMUCs)*, which are formal statements that the regions of memory they describe are the only ones possibly affected by the functions under test. These statements also come with *proofs*, which for our work are like scripts used to verify that the things the FMUCs assert about the corresponding functions can be shown to be true given the assumptions our FMUCs have. Sometimes those proofs are incomplete, though, such as when there is a *loop*[4] in a function under test or one function *calls*[5] another. In those cases, a user has to finish the proof, in the first case by *weakening*[6] the FMUC's statements about the loop and in the second by *composing*, or combining, the FMUCs of the two functions.

Additionally, this second approach cannot handle *dynamic control flow*. Such control flow occurs when the low-level instructions a program uses to move to another place in that program do not have a pre-stored location to go to. Instead, that location is supplied as the program is running. This is opposed to *direct control flow*, where the place to go to is hard-coded into the program when it is compiled. The tool also cannot not deal with *aliasing*, which is when different *state parts*[7] of a program contain the same value and that value is used as the numeric address or identifier of a location in memory. Specifically, it cannot deal with *potential* aliasing, when there is not enough information available to determine if the state parts alias or not. Because of that, we had to add extra assumptions to the FMUCs that limited them to those cases where ambiguous memory-referencing state parts referred to *separate* memory locations. Finally, it specifically requires assembly as input; you cannot directly supply a binary to it. This is also true of the first contribution. Because of this, we were able to test on more functions than before, but not a lot more.

Not being able to deal with dynamic control flow is a big problem, as almost all programs use it. For example, when a function reaches its end, it has to figure out where to return to based on the current state of the program (in the previous contribution, this was done manually). This means that *control flow recovery* is very important for many applications, including *decompilation*,[8] *patching*,[9] and low-level analysis or verification in general. However, as you may have noticed from earlier in this paragraph, in order to deal with such dynamic control flow you need to figure out what the possible destinations are for the individual control flow transfers. That can require knowing where you came from in the program, which means that analysis of dynamic control flow requires *context*.[10] Even worse, it is another undecidable problem that requires overapproximation.

To soundly recover control flow, we developed *Hoare graphs (HGs)*, the third contribution of this dissertation. HGs use *memory models* that take the form of *forests*, or collections of tree data structures. A single tree represents a region in memory that may have multiple *symbolic* references, or abstract representations of a value. The children of the tree represent regions

[3] the textual form
[4] repeated bit of code
[5] executes
[6] removing information from
[7] value-holding components
[8] converting a program back into a higher-level form
[9] updating a program in place without modifying the original code and recompiling it
[10] in this context, information previously obtained in the program

used in the program that are *enclosed* within their parent tree elements. Now, instead of assuming that all ambiguous memory regions are separate, we can use them under various aliasing conditions. We have also implemented support for some forms of dynamic control flow. Those that are not supported are clearly marked in the resultant HG. No user interaction is required even when loops are present thanks to a methodology that automatically reduces the amount of information present at a re-executed instruction until the information stabilizes. Function composition is also automatic now thanks to a method that treats each function as its own context in a safe and automated way, reducing memory consumption of our tool and allowing larger programs to be examined. In the process we did lose the ability to deal with *recursion*,[11] though. Lastly, we provided the ability to directly load binaries into the tool, no external *disassembly*[12] needed. This all allowed much greater testing than before, with applications to multiple programs and program libraries.

The fourth and final contribution of this dissertation iterates on the HG work by narrowing focus to the concept of *exceptional control flow*. Specifically, it models the kind of exception handling used by C++ programs. This is important as, if you want to explore a program's behavior, you need to know all the places it goes to. If you use a tool that does not model exception handling, you may end up missing paths of execution caused by *unwinding*. This is when an exception is thrown and propagates up through the program's current *stack* of function calls, potentially reaching programmer-supplied handling for that exception. Despite this, commonplace tools for static, low-level program analysis do not model such unwinding. The control flow graphs (CFGs) produced by our exception-aware tool are called *exceptional interprocedural control flow graphs (EICFGs)*. These provide information about the exceptions being thrown and what paths they take in the program when they are thrown. Additional improvements are a better methodology for handling dynamic control flow as well adding back in support for recursion. All told, this allowed us to explore even more programs than ever before.

[11] functions that call themselves or call other functions that call back to the original

[12] converting machine code into human-readable instructions

This work is dedicated to my dearly departed cat, Abby, who lasted through my Master's but was not able to make it to the end of my PhD.

Acknowledgments

I cannot express in words how grateful I am to those who have assisted me on the long and harrowing journey to obtain my doctorate, which did not even start off as such a journey. However, I will attempt to do so here.

First, my family and family friends, who have been incredibly supportive on this quest. I am eternally grateful to them for that support, both the tangible and intangible.

Next, my coworkers and collaborators and fellow students in and outside the Systems Software Research Group (SSRG), both those who have moved on to other things and those who have yet to do so, who I have helped and been helped by in turn. I wish them all the best in their endeavors. I also treasure the guidance I have received from my advisor and the postdocs and professors I have worked with and learned from both at Virginia Tech (VT) and previously at North Carolina State University (NCSU).

Finally, my friends throughout the years, both in school and outside of it. Those I have hung out with, played games with, gone to movies or watched TV shows with. Though academic efforts make maintaining such ties difficult, I hope to maintain or even strengthen them moving into the future.

The bulk of this work was supported by a mix of the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028, DARPA under Agreement No. HR.00112090028, the Office of Naval Research (ONR) under grant N00014-17-1-2297, the Naval Sea Systems Command (NAVSEA)/the Naval Engineering Education Consortium (NEEC) under grant N00174-16-C-0018, and the Naval Surface Warfare Center Dahlgren Division (NSWCDD) under grant N00174-20-1-0009.

Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of DARPA, NIWC Pacific, ONR, NAVSEA/NEEC, or NSWCDD.

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

List of Theorems

Abbreviations

ABI application binary interface i, 3, 9, 15, 16, 98, 104, 110, 121, 126, 131, 134, 138
AES Advanced Encryption Standard 21, 23
AFP Archive of Formal Proofs 56
Amazon EC2 Amazon Elastic Compute Cloud 74
API application programming interface 30
APT Advanced Package Tool 130
ATP automated theorem prover 23

BAP the Binary Analysis Platform 11, 22

CBC cipher block chaining 19, 23
CFG control flow graph iv, xxiv, xxv, 4, 11, 12, 14, 16, 17, 24, 27, 28, 30, 41, 44, 48, 54, 58, 59, 62, 65, 79, 110, 111, 137, 138, 141
CFI control-flow integrity 4, 30, 57
CFR control flow recovery xxvii, xxviii, 4–6, 9, 11, 12, 14, 27
CG certificate generation 23
CIE Common Information Entry 118
CISC complex instruction set computer 10, 163
COTS commercial off-the-shelf 5, 16, 25, 134, 138
CPU central processing unit 7, 10, 15, 100, 131
CSP communicating sequential processes 22

DARPA the Defense Advanced Research Projects Agency vi
DFS depth-first search 97
DiL decompilation into logic 10, 20, 22–24, 28
DM device model 75
DSL domain-specific language 163
DVR dead variable reduction 21

EH exception handling i, xxviii, 9, 12, 15–17, 24, 29, 58, 110, 130, 131, 134, 138
EICFG exceptional interprocedural control flow graph iv, xxiv, xxv, xxviii, 11, 12, 14, 15, 17, 29, 30, 110–112, 118, 121, 127, 130–134, 138, 140–142
ELF Executable and Linkable Format 16, 26, 30, 101, 103, 125, 132

FDE Frame Description Entry 118
FDL functional description language 18, 19
FIFO first in, first out xxv

FMUC formal memory usage certificate iii, xxvii, 12, 57, 58, 60, 64–66, 68–70, 72, 76, 137

GCC the GNU Compiler Collection 47, 49, 74
GCM Galois/Counter Mode 21
GDB the GNU Project debugger 11, 29, 129, 130, 133
GPF general protection fault 140

HDD hard disk drive ii
HG Hoare graph iii, iv, xxiii–xxv, xxvii, xxviii, 12, 14, 17, 23, 25, 28, 79, 80, 82–85, 93, 96, 97, 100–102, 104, 105, 107, 108, 110, 129, 137, 138, 140–142
HOL higher-order logic 10, 20, 163

IECFG interprocedural exception control-flow graph 30
IH induction hypothesis 91
IPC inter-process communication 5
IR intermediate representation 5, 6, 142
ISA instruction set architecture 2, 3, 7, 9, 10, 13, 15, 18–23, 26, 27, 33, 35, 53, 74, 104, 121, 131, 163
ITP interactive theorem proving 2, 20, 23, 56, 136, 142

JIT just-in-time 30
JTUB jump table upper bound 132, 133
JVM Java virtual machine 19, 23, 27

LFP least fixed point 25, 42, 65, 85
LHS left-hand side xxiv
LIFO last in, first out xxv
LSDA language-specific data area 118
LTS long-term support 131

MC model checking 23
MIPS Microprocessor without Interlocked Pipelined Stages 23, 35
ML Meta Language 163
MRR memory region relation 13, 57, 58, 61, 72, 74, 76, 137

NAVSEA the Naval Sea Systems Command vi
NEEC the Naval Engineering Education Consortium vi
NIWC Pacific Naval Information Warfare Center Pacific vi
Nqthm the Boyer-Moore theorem prover 19, 23
NSWCDD the Naval Surface Warfare Center Dahlgren Division vi

ONR the Office of Naval Research vi
OS operating system 8, 23, 53, 56, 100, 131, 139

PowerPC Performance Optimization With Enhanced RISC – Performance Computing 35

QEMU Quick Emulator 75

RAM random-access memory ii, 100, 131
RHS right-hand side xxiv
RISC reduced instruction set computer 10, 163
RNN recurrent neural network 24
ROP return-oriented programming 4, 8, 14
RTOS real-time operating system 22

SA static analysis 23
SCF syntactic control flow 14, 16, 58–60, 64–67, 69, 72, 74, 76, 79, 137
SIMD single instruction, multiple data 75, 108
SLOC source lines of code 13, 23, 54
SMT satisfiability modulo theories 21, 22, 58, 61, 88, 141, 163
SO shared object 101
SPADE the Southampton Program Analysis Development Environment 9, 18, 19
SPARC Scalable Processor Architecture 27, 35
SSD solid-state drive ii, 100
SSE Streaming SIMD Extensions 34
STP Simple Theorem Prover 22

TCB trusted computing base ii, 2, 3, 10, 11, 15, 28, 47, 58, 107, 108, 136
TV translation validation 23

UC user contract 23

VC verification condition 19, 21, 22, 33, 137
VCG verification condition generator xxvii, 18, 21–23, 58, 69
VM virtual machine 53, 74
VMM virtual machine monitor 74

Abbreviations with Descriptions

Controller Area Network (CAN)

A standard network for electronic communications in automotive applications. 21

North Carolina State University (NCSU)

The university I attended for undergrad. vi

Systems Software Research Group (SSRG)

The research group at Virginia Tech I have been a part of for almost eight years. vi, *see also* VT

Virginia Tech (VT)

The university I attended for graduate school. vi, *see also* SSRG

Symbols

Elements

⊥ The "bottom" element of a set. Often used 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 as the `None` value of an optional type. In the HG work specifically, it is used as the always-false symbolic state predicate or as an unknown/undefined ℂ. 16, 25, 44, 86, 94, 95, 98, 119, 120, 123, 124, 126–128, *see also* ℂ

⊤ The "top" element of a set; used as the always-true symbolic state predicate for HGs. 86

Functions

eval Expression evaluation function; maps a possibly-state-dependent expression to a constant expression. 94, 96

EXPLORE The base function of the HG extraction algorithm. 93, 95

fetch Disassembles the instruction at the supplied address. 16, 84, 94, 97

memory model

ins Inserts the first memory tree argument into the second. 89–91, 94

pred Returns the predicate for the supplied state. 96

prop Invariant propagation function for Hoare-style work. 62

RUNUNTIL A function that performs symbolic execution until the supplied halting condition holds. 42–44, 65

sound The supplied HG is sound with respect to the supplied binary. 97

Stack Operations

peek Returns the element at the top of the supplied stack. 120, 123, 124, 126

pop Removes the element at the top of the supplied stack. 120, 121

push Places the supplied element on the top of the supplied stack. 120, 121

STEP General form for a step function (used by the Floyd-style and Hoare-style works). 42, 65

STEP_Σ The symbolic state step function for HG generation. 94, 96–98

sxtnd Bitvector sign extension. 73

τ The instruction-semantics-modeling inner step function for HG generation. 94, 96–98

Relations

+= In-place addition. 96

C Indicates the two memory model arguments are comparable. 92

Types

Variables

Attribution

The work presented in this dissertation is not solely my own, as the four contributions it contains were significant collaborative efforts, previously published in paper form. The sections below describe which components were primarily my work and which were mainly those of my compatriots. Joint efforts are documented as well.

Symbolic Execution

The initial work on formulating properties for memory regions as presented in Section 3.2, which is shared between the two works presented in this document, was provided by Dr. Freek Verbeek.

The symbolic execution engine used in those two contributions was initially provided and developed by Dr. Peter Lammich. While the machine model we ended up using in the work was provided to us as well, the semantics was not fully suitable for efficient formal verification in Isabelle. This was due to its status as bitvector formulas [150], which Isabelle did not have strong support for at the time. Therefore I also assisted in the development of simplification rules to enable higher-level reasoning on individual and multiple instructions. Many were also provided by a fellow student working on a tangential project [181]. As with the symbolic execution engine, the rules were implemented in Isabelle and formally proven correct (Section 3.2).

Floyd-Style Verification

The first main contribution to cover is the Floyd-style verification work described in Chapter 4 [22]. I provided three main contributions to that work, previously published as "Formal Verification of Memory Preservation of x86-64 Binaries".

The first was a Python package developed to interface with `angr` [165] for cutpoint identification and skeleton proof generation, described in Section 4.1. My second contribution was the development of structured proof strategies to flesh out and verify the skeleton proofs. This included developing the necessary preconditions and postconditions to ensure the possibility of function-level composition (Section 4.3).

My third contribution was the analysis of more than seventy functions that I selected as suitable case studies. Section 4.4.1 presents one such function, and the rest of the verification

effort is described in Section 4.5. These functions were non-recursive (but still potentially looping). The two recursive functions handled in the process of validating the work, including the factorial function presented in Section 4.4.2, were primarily verified by Dr. Verbeek.

Hoare-Style Verification

Following on from that work is a paper that built on our experiences from the previous verification work [180], "Highly Automated Formal Proofs over Memory Usage of Assembly Code". This work, which introduces the generation and verification of FMUCs, is described in Chapter 5.

Most of my primary contributions to this work were on the verification side of things. The first two big ones were application and development of syntactic rules for axiomatic reasoning over memory usage as applied to structured control flow (Section 5.2.4) as well as a method to automate the process of doing so, a verification condition generator (VCG) as described in Section 5.2.4. The methodology for function-level compositionality in Section 5.2.5 was also a primary contribution of mine, as was the case study analysis in Section 5.4.

On the FMUC generation side of things, I did most of the work on translating the Haskell FMUCs into the theorems and data types required to verify the FMUCs in Isabelle, which included adapting the syntactic control flow used in FMUC generation into a suitable form for the Isabelle work (Section 5.2.1). I also made some significant contributions to invariant generation in Section 5.1.3 and contributed some entries in the implemented Haskell semantics for additional instructions encountered in the process of FMUC generation.

Dr. Verbeek's contributions to the FMUC generation are as follows. In order to properly generate FMUC preconditions and postconditions, he formulated a symbolic execution machine model in Haskell for many common x86-64 instructions as well as additional ones encountered in the course of our case studies. Dr. Verbeek also developed the Z3 [55] interface for handling the memory region separation and enclosure decision problems, mentioned in Section 5.1, and did most of the work for control flow extraction (Section 5.1.1).

Hoare Graphs

The first of two control flow recovery (CFR) papers, "Formally Verified Lifting of C-Compiled x86-64 Binaries" was again a collaboration with Dr. Verbeek and others, with him and me being the major contributors to the work. It introduced the concept of HGs and provided formalisms, pen-and-paper proofs, a case study, and formal verification of some produced HGs [179].

The initial codebase and basic step function as well as the memory model framework were developed by Dr. Verbeek, but I took ownership of it after that. While Dr. Verbeek did contribute further adjustments during the process, most of the development from that point on was done by me. This includes fleshing out the step function with the semantics of more instructions as we encountered them in the case study, as well as larger decisions

about implementation structure. For example, I made the decision to switch the individual instruction parsing over to use Capstone [4] via the `hapstone` package [6] when a collaborator uncovered a bug in the parsing library that was originally chosen. I also transitioned the work from relying on unbounded integers in the Z3 interface to utilizing Z3's bitvector formulas, providing more accuracy and precision. Additionally, development of instrumentation for and application of the tool to the real-world programs and libraries of the case study was all done by me.

Finally, the formal proofs of HG in Isabelle were performed entirely by Dr. Verbeek. They have been documented in this dissertation for completion's sake. Most of the pen-and-paper proofs were also refined or completed by Dr. Verbeek, but I did assist with those and provided some of the initial ones.

Exceptional Interprocedural Control Flow Graphs

The second of the CFR papers and the final work in this dissertation, this paper presents the concept of EICFGs and provides formalism, pen-and-paper proofs, a case study, and fuzzed validation of some of the relevant EH library functions.

While this work was also done in collaboration with Dr. Verbeek, I did most of the actual development. This includes the implementation, formalisms, selection and analysis of the case study, and validation of components. Some of the basic structure of the implementation was pulled from the previous contribution. This includes such aspects as the instruction fetching and the initial version of the step function. However, everything built on top of that was all new, and a significant amount of refactoring occurred even for the shared components.

Part I

Prologue

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. Of course, software bugs have existed ever since the creation of non-trivial programs. For at least as long, researchers have been working on methods of bug-free programming and sound program analysis.

1.1 Formal Verification of Software

Many have dipped into the field of *formal methods* [31] to do this. For our purposes, the subfield of *formal verification* specifically. Formal verification allows reasoning over programs with a high degree of assurance. This assurance is provided by sound mathematical representations and logical reasoning. Even pen-and-paper proofs can provide guarantees that purely informal analysis does not.

Unfortunately, formal verification of software is still a difficult task. Many useful properties are *undecidable* [89, 138, 149]. There are no single, fully-automated methods that are guaranteed to determine them for all possible programs [23]. One alternative is to use interactive theorem proving (ITP) [81, 119, 158]. However, that does not scale well due to the amount of intricate user interaction and development involved.

1.2 Assembly and Binary Analysis

Source-code-level formal verification has another issue. It requires a large *trusted computing base (TCB)* [113, p. 270, 115, 153, p. 13]. Without a verified compiler [116], one must trust the semantics of a binary reflect its original source. Even then, one must assume that the instruction set architecture (ISA) in use has been correctly specified, that all libraries work correctly, and even that there are no firmware or hardware flaws in the physical computer being used to execute the program. Ultimately, it is not possible to eliminate the TCB

completely. However, it is possible to *reduce* it, and not just via the usage of non-mainstream compilers that can only handle a subset of programs for their given language.

This is where the field of *low-level analysis* comes into play. We focus here on assembly and binary analysis done *statically*. That is, analyzing programs "offline" with the usage of external tools rather than instrumenting or hooking into a running program. This automatically eliminates the compiler and any optimizations it may do from the TCB.

An additional benefit is that it allows for the analysis of programs when source code is not available. This is useful for reverse engineers and legacy code maintainers. For example, when a contractor is hired to develop a program, it can often cost a lot more money to get the source code than just a compiled version of the program. In such cases it may be more cost-efficient to just get the compiled version and figure out how to patch it for updates locally.

However, there is no such thing as a free lunch. The tradeoff for lower-level analysis is less abstraction. This is because even software written in a relatively low-level language like C has abstractions on memory for local variables and function calls. How and where memory is allocated may be compiler, application binary interface (ABI), and ISA-specific. It can even depend on what compiler options are in use, including the level of optimization. As a further illustration, consider formulating a property that a function cannot overwrite its own return address (something done in this dissertation). Doing so requires knowledge of the layout of the stack, including the values of the stack and frame pointers, thus making it an *assembly-level* or lower property. On the flip side, this also means that *such low-level properties can be precisely stated even if they are not evident on the source level.*

One such property is *memory usage*. Informally, memory usage describes the memory a program writes and reads (in other words, *uses*) in terms of prespecified regions. If verified correct, all memory utilized by the program is bounded to those regions.

Another property is *actual control flow recovered from the binary*. The control flow one can identify from source code is not necessarily the actual control flow a program will take due to the binary-source semantic gap. When a program does something not intended by the programmer due to the lack of abstraction enforcements on a low level, it exhibits "weird" behavior [58, 161]. In order to have a sound and thus *overapproximative* approach, both "normal" and weird edges must be recovered.

The above two properties are the ones targeted in this dissertation. We provide two contributions for each, summarized here and expanded on later in this chapter.

Memory Usage of Assembly Programs via Formal Verification

- A methodology based on Floyd-style verification [67]. It requires manual interaction during specification of the used memory regions as well as during proof completion.
- A methodology based on Hoare-style verification [87] that adds verification of *return address integrity*. The generation of assumptions made during verification as well as the proofs themselves are mostly automated.

Control Flow Recovery

- A methodology for formally verified disassembly and control flow recovery (CFR) for binaries compiled from C source code.
- A methodology or recovering *exceptional*, *interprocedural* control flow from C++ binaries.

1.3 Motivation

Now that we have introduced the concepts of memory usage and CFR, we here provide some motivation for their usage.

1.3.1 Memory Usage

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

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 [82]. 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 usage analysis could serve as a foundation for formal security analyses that could be used to expose vulnerabilities involving malicious writes.

Another important property that memory usage could help with is control-flow integrity (CFI) [2]. 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 part of what the property of memory usage states. This can aid in avoiding return-oriented programming (ROP) 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 first group's actions have no effect on what the second group of users can see [78, 154]. 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. Remember that memory usage is specifically about showing that all memory outside of specific regions is not modified or read by the function or functions associated with

those regions. This means that proving the region sets for two functions are disjoint would essentially prove noninterference for those two functions.[1]

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 [108, 135, 148]. Put into 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 usage allows discharging the most involved part of the frame rule, at least in terms of individual assembly functions. That is, it shows that the operations on memory in those functions are constrained to specific regions. This could then serve as a basis for a larger proof effort over multi-function assembly programs.

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 (IPC), 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 usage could be adapted to show the absence of such interactions by proving that multiple threads only write to specifically-allowed regions of shared memory.

1.3.2 Control Flow Recovery

The process of CFR is a vital component of many binary analyses and transformations. It has applications in the field of decompilation, verification, patching, and security. This is because all of those applications, discussed below, require *lifting* raw unstructured data to a form that allows reasoning over behavior and semantics.

Decompilation

The process of *decompilation* is the reverse of compilation [36]. It attempts to *lift* low-level code, possibly even machine code, to a higher-level representation. That representation can be something like LLVM's intermediate representation (IR)/bitcode [56], C [30], or even C++ [68]. Commercial off-the-shelf (COTS) tools such as Ghidra [134], IDA Pro [86], and Binary Ninja [189] often integrate or offer as a plugin such a decompiler as well, or are used to

[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 to prove as we do not currently differentiate between regions that are read and written.

provide the control flow for one [175]. Those decompilers generally target a source language such as C.

Decompilation is most important to the field of *reverse engineering*. This often occurs when source code is not available, either because of legacy programs where the source has been lost or because of proprietary restrictions. Under ideal circumstances, it produces human- or machine-readable source code that can be used to intuit the behavior and semantics of the program being reverse engineered. It also allows for easier reconstruction or modification of the program, if that is the intent.

Verification

As seen later on in Chapters 4 and 5, CFR is also very important for the process of verification. Many, if not all, binary verification efforts require an understanding of program control flow. Without in-depth CFR, you end up with tools that cannot handle complex control flow such compiled `switch` statements [128, 169]. This is also evident in the aforementioned chapters.

Multiple verification efforts utilize the above-mentioned decompilation approach as well rather than directly validating or verifying assembly semantics. Generally they lift to an IR that is then verified [29, 126, 127]. This allows for more abstraction, but does require a lot of groundwork.

Patching

Non-trivial, smart binary patching or rewriting requires control flow information as well. If you are instrumenting binaries to dynamically randomize their own basic blocks, you cannot do so without knowing the bounds those blocks must abide by [100, 185].

Duck, Gao, and Roychoudhury [57] did recently provide a way to patch binaries without control flow information. However, that work requires potentially inefficient program modification such as extra instruction padding to accomplish that task. It is also restricted specifically to insertion of code trampolines [7]. That is useful if program size and usage of additional functions is not a concern but may not help for more restrictive tasks.

Security

General, non-formal security analysis also benefits from control flow information. Sometimes this is due to the tool targeting control flow itself [52, 109]. At other times, it is because that information about control flow is necessary to reason over program behavior and semantics. For example, dead code removal, the elimination of code that is never executed, is sometimes used to lessen the attack surface of a program. However, knowing what code is dead requires reachability analysis, which inherently requires control flow.

In fact, platforms for binary analysis often provide CFR themselves [167, 183]. Some even implement their own decompilation or use existing tools in order to simplify those analyses. This allows developers to focus on the actual analyses and not deal with all of the low-level details. However, that does require a deep understanding of what information can

be abstracted away and what cannot. The choice of model for analysis, whether formal or informal, affects what properties of the concrete item can be analyzed.

1.4 Challenges

As mentioned, the biggest challenge in assembly-level verification specifically 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 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 abstraction. Instead, all control flow on the assembly level is performed using conditional or unconditional branches (jumps and calls), either to a predetermined location or to a calculated label. Some lower-level languages such as C do allow for direct jumps to explicit labels via statements such as `goto`, or even indirect versions with stored labels via compiler extensions. However, modern best practices discourage the usage of those constructs as they do indeed complicate control flow analysis and can often be transformed into a more structured style without significant code growth.

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 units (CPUs) for embedded systems lack support for hardware division as efficient division algorithms require a lot of logic gates. For such processors, runtime division must be calculated using an algorithm implemented in assembly rather than via a specific instruction.

1.4.1 Types and Memory

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 [64, 65, 173], but we do not have that luxury on typical modern systems. Tagged memory could also serve that purpose [27, 168, 195], but it is not yet widely in use.

An additional issue with assembly, and the one most significant for the memory-related properties in this dissertation, lies in the simplicity of the user-exposed memory model. The vast majority of high-level, structured languages with scoping, including C, prevent functions from accessing the local variables of other functions without significant effort or explicit notation or argument passing, 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 runtime libraries and operating systems (OSes) do provide some form of memory protection, but those generally rely on runtime detection of invalid accesses for unallocated memory and are often not fine-grained enough for reasoning about individual stack frames or local variables. They also do not work if the program manually manages its memory in large chunks, as happened with Heartbleed. Any verification effort that wishes to reason about low-level memory properties therefore must provide its own abstractions and assumptions on layout.

1.4.2 Disassembly and Binary Lifting

Switching to an approach that integrates disassembly into the process of extracting information from a low-level program further requires gaining insight into program control flow. Typically, such binary lifting requires answering *at least* the following base questions:

- *Which* instructions are potentially executed within the binary?
- *In what order* can those instructions be executed?

Those two questions are mutually recursive; they cannot be isolated from each other. This is the "chicken-and-egg" problem of disassembly [159].

Put another way, once disassembling more than one instruction, disassembly[2] requires knowledge of which instructions are to be disassembled next. However, in order to determine those instructions, you have to know how the one that was just disassembled was reached. In other words, you need the *control flow* of the program. Such information is not necessarily statically available. For example, jump targets may need to be dynamically computed, the stored return addresses for `ret` instructions change based on context, and even the bounds on jump table indices themselves may not be fixed. Thrown exceptions and callbacks supplied to external functions are also non-obvious sources of control flow. Determining the target of a `ret` can be non-trivial even when the call graph is nominally static as well, because there is always the possibility of an instruction within the returning function having overwritten the return address.[3]

At a minimum, a disassembler that supports calls and indirect jump traversals in a structured setting needs to ensure the following properties:

Return Address Integrity Functions cannot overwrite their own return addresses, or if they do the target must be known.[4] This requires the absence of stack overflows or similar inappropriate stack manipulation.

Bounded Control Flow All indirect, or non-immediate, branches transfer control flow to fixed, statically-calculated, bounded sets of addresses. This requires the ability to determine upper bounds on array indices.

[2] Specifically for our case, recursive descent disassembly.
[3] When used purposely, this is the core of the previously-mentioned ROP.
[4] This assumes a standard structured programming methodology.

Calling Convention Adherence All called functions properly restore the set of registers the 64-bit System V ABI considers non-volatile.

This problem is exacerbated when dealing with *exceptional control flow*, such as that induced by the C++ `throw` and `try`...`catch` statements. Such control flow does not respect traditional structured programming paradigms. Modern methods also require a significant amount of auxiliary knowledge that may not be directly present in the compiled code itself. Debugging builds have it to provide support for stack unwinding during debugging. Programs with structured exception handling (EH) require it even when debugging information is excluded.

Furthermore, that control flow is *dynamic*. The target of a throw, which instruction address the process of unwinding ends up at, is decided at runtime by standard library functions [32]. These include `__cxa_throw`, `__cxa_begin_catch`, and `__cxa_end_catch`. Determining that target requires knowing the current function call stack, the current caught exception stack, and various other details stored within the exception objects themselves. It also requires modeling the semantics of those library functions. Notably, that need for the current call stack means CFR with exceptions is inherently *interprocedural*. Interprocedural analysis is challenging, as one cannot simply isolate individual functions or blocks of machine code but must instead consider the binary as a whole.

1.5 State of the Art

Initial formalisms developed by Floyd [67] and Hoare [87] have provided a basis for the formal verification of software for over fifty years. Many works have built on those core structures over the years to model additional program features, such as *separation logic* [148] for composability and various ways of representing concurrent code [139, 191]. Despite this, to the best of our knowledge, no works successfully tackle the specific problems tackled in this dissertation.

1.5.1 Assembly Verification

As there are currently no state-of-the-art methods that specifically aim to formally verify the property of memory usage as described here, we focus on other important works in the field that lead up to our contributions for memory usage.

One of the first major efforts in assembly verification was that of Clutterbuck [41] and Clutterbuck and Carré [42]. That work involved analysis of a subset of the Intel 8080 ISA, SPACE-8080, using the Southampton Program Analysis Development Environment (SPADE) [33]. SPADE was an early software suite for the development of high-integrity software. Their work utilized an interactive Floyd-style verification approach to prove functional correctness of very small functions. It was quickly followed by a real-world application of the approach in the field of aeronautics [136]. That application involved the verification of the firmware of a jet engine's fuel control unit.

Another big breakthrough was the development by Myreen et al. of an initial approach to

formal binary lifting. Known as *decompilation into logic (DiL) [126–128]*, it converts machine code into a higher-order logic (HOL) representation. This representation can then be used for verification efforts in a theorem prover. As a specific example, it was utilized in the verification of the seL4 microkernel [105, 106, 172]. However, though the usage of DiL enabled automatic binary lifting for much of seL4, that verification effort still required significant manual effort.

In Goel [76] and Goel et al. [77] 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 [98]. Further previous and later efforts for a variety of assembly-level analyses and targets are documented later in Section 2.1. The ones mentioned here are also described in more detail.

Tackling the issue of instruction semantics from another angle, the STRATA tool by Heule et al. [84] provides a technique using machine learning to derive bitvector semantics from actual CPUs. This allows for more accurate semantics than previous approaches relying on human-interpreted specification documents even when when working with complex instruction set computer (CISC) ISAs. As evidence, the Heule team identified mistakes and bugs in the human-readable Intel documentation of the instructions they derived semantics for. Of course, even reduced instruction set computer (RISC) specifications, despite ostensibly being simpler, are not always implemented accurately in emulators or other tools not synthesized directly from a structured specification [94]. This indicates that TCB reduction is desirable for all architectures if the goal is to maximize reliability and trustworthiness of the final product.

Despite the accomplishments of the Heule work, it still has some drawbacks. Most importantly, their approach was not formally verified. This means that there is still potential for bugs in the specification or synthesis of semantics to go unnoticed. That gap was filled by the work of Roessle, Verbeek, and Ravindran [150]. That work establishes a methodology for generating formal equivalence theorems between assembly code and big-step semantics in an interactive theorem prover (specifically, Isabelle). This was accomplished by using STRATA to extract per-instruction small-step semantics and then performing tests in a formal framework against actual hardware to validate them. The validated semantics were then used in DiL methodology to lift full programs into a form that could be easily reasoned over, even in the presence of floating-point operations. Though we did not directly build on the works of Heule et al. [84] or Roessle, Verbeek, and Ravindran [150], they served as inspiration and influence in developing our instruction semantics as well as for the validation performed in Chapter 12.

1.5.2 Control Flow Recovery

At the time this dissertation was written, no other tool operating on binaries could provide scalable, formally overapproximative assurance between a binary and its lifted representation. The bulk of existing methods are either known to be unsound (they either misidentify code as data or are underapproximative) [159] or are speculative or learning-based [13, 97, 187]. The strength of those tools is their universality: they typically provide output for any binary, even in cases where guesses and non-validated assumptions have to be made. Their weakness is that

their outputs are untrustworthy; thus, any analyses built on top of them are untrustworthy as well.

There have been some efforts to resolve this issue. For example, Brumley et al. [30] developed Phoenix, a tool for decompilation to C. This tool ensures the preservation of semantics of binaries even in the presence of unstructured control flow. However, it relies on an external tool, the Binary Analysis Platform (BAP) [29], to perform the CFR and they encountered multiple situations where it did not work due to its lack of support for floating point and other "exotic" instructions. Kinder [101, 102] and Kinder and Kravchenko [103] also provided the tool Jakstab, which performs formal binary lifting using abstract interpretation. However, it is targeted at a different, more limited set of applications from our works; specifically, small obfuscated programs such as Windows device drivers and malware. It also often requires an external harness in order to validate those drivers which must be included in the TCB if it is not itself validated. Furthermore, we argue that it is not actually an overapproximative tool later in Section 2.3.1.

Additionally, existing state-of-the-art disassemblers/decompilers, such as IDA Pro [85], Ghidra [134], and Binary Ninja [178], do not document exceptional control flow between procedures. They are able to extract the exception information statically available in binaries, including landing pad locations. They can even provide interprocedural CFGs. However, they do not perform the interprocedural static analysis required for reconstructing *exceptional* control flow. That is, they cannot trace the path from an exception throw site to the landing pad instructions the thrown exception goes to in the process of unwinding. CFGs under such analyses do not have any outgoing edges from throw sites. They are portrayed as non-returning, or terminating, functions.

Therefore, in order to analyze the stack unwinding caused by an exception being thrown, one must typically utilize a runtime debugger such as the GNU Project debugger (GDB). In fact, many state-of-the-art disassembly tools, IDA Pro and Ghidra included, rely on GDB to perform dynamic analysis via debugging. That is, they require runtime information in order to trace exceptional state. In contrast, our exceptional interprocedural control flow graph (EICFG) generator does not.

1.6 Contributions

This dissertation consists of four main contributions. Two of them are formal approaches to per-function verification of the assembly-level property we call memory usage. The other two are methods for CFR from binaries.

The first two contributions are the memory usage approaches: *Floyd-style* verification and *Hoare-style* 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 [131] and completed there. The proof strategies for both approaches involve *symbolic execution* of the underlying assembly code [104], 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, Floyd-style verification, requires significantly more user user input but has the potential for much stronger invariants. Meanwhile, the second approach, Hoare-style verification, has a significantly higher level of proof automation via the generation of formal memory usage certificates (FMUCs) but is not as suited for stronger invariant production. Symbolic execution is also more efficient in the Floyd-style approach as it more closely follows the structure of the function's CFG. In contrast, the Hoare-style approach must deal with operating on a restricted set of control flow constructs, which can result in extra symbolic execution.

We also provide two trustworthy approaches to sound CFR: Hoare graph (HG) generation and EICFG generation. Both approaches target binaries specifically rather than relying on external tools for assembly extraction. They also both provide *assurance* that *if* unannotated output is produced, that output is a sound representation of the binary.

However, they differ in their degree of scalability, their degree of context-sensitivity, and the sort of information lifted for their individual CFGs. For example, the HG work generates proofs of output assurance while the EICFG work uses informal proofs combined with concrete validation of its EH-related abstract transition rules.

1.6.1 Floyd-Style Memory Usage

This methodology for verification of memory usage 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. This is why we focused on the aforementioned property of memory usage.

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 usage 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 and complete proofs for memory usage 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 functions. This is achieved using off-the-shelf tools.

That proof skeleton 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 necessary for successfully completing symbolic

execution and fleshing out the annotations on state so that the symbolic execution of later blocks can continue from that of earlier ones.

This methodology was applied to 63 functions extracted from the HermitCore [114] unikernel library [118], 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. [121], 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 usage, 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.

1.6.2 Hoare-Style Memory Usage

Taking our experiences from the Floyd-style verification work into account, we chose a slightly different path for the other memory usage 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* [87] over the syntactic control flow. 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 usage.

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 [34]. Of the 352 functions present in those binaries, 251 or Successful Xen percent 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 each of the 65 loops

handled.

To the best of our knowledge, this is the first work to achieve that degree of coverage for optimized x86-64 binaries produced by production code. While the aforementioned methodology produced by Tan et al. [169] is fully automated, it was much slower than our approach here. This means it would take longer to cover the same amount of functions we did even though it technically has 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 (SCF) with respect to minimizing symbolic execution, but those were atypical.

1.6.3 Hoare-Logic-Based Control Flow Recovery

Changing focus to the contributions for CFR, we have an algorithm (and implementation) for lifting an HG out of an x86-64 binary. HGs are Hoare logic-based CFGs. The vertices of those HGs are symbolic states that consist of the following:

1. *predicates* containing information on registers, memory locations and flags and
2. *memory models* that provide pointer aliasing information.

Each HG edge is labeled with the corresponding assembly instruction. Our key intuition here is that those edges are one-step *inductive*: each edge forms a *Hoare triple* [87]. Every vertex/state contains enough information to prove that its outgoing edges are overapproximative, even in the case of non-trivial control flow. This includes indirect branches, jump tables, and function calls/returns. The soundness of this overapproximation is ultimately shown with pen-and-paper proofs in Chapter 7.

Additionally, Section 6.1 contains an example where instructions are potentially overlapping, which is often an arrangement found in obfuscated code. That example exhibits an ROP gadget that depends on whether two pointers alias or not, with the aliasing case resulting in an unexpected `ret` being executed. That aspect of the example is discussed further in Section 6.1.4.

This work proved to be more scalable than Jakstab, recovering the control flow for 399 771 instructions.

1.6.4 Exceptional Interprocedural Control Flow Recovery

The final contribution of this dissertation is a tool for static, interprocedural, automated C++-exception-aware [32] binary-level control flow analysis. That tool produces *EICFGs* using an abstract interpretation-inspired [46, 47], recursive-descent methodology [129]. It can do this even in the presence of recursion and some forms of indirect control flow. To accomplish this, the EICFGs document program state in a restricted domain that only concerns itself with the components of program state necessary to perform function calls and

EH. This includes exception objects currently allocated, number of uncaught exceptions, and which exceptions are currently in a caught state.

Symbol-stripped binaries are supported, but may not perform well if they contain unresolvable indirections or unmodeled callbacks. Non-C++ binaries, or C++ ones that do not use EH, are supported as well for flexibility.

As with the other contributions in this dissertation, the produced EICFGs are *overapproximative*. Every concrete path in a program, every path possible during dynamic execution, is included in its corresponding EICFGs. We informally argue that the abstract semantics that are symbolically executed overapproximate their corresponding concrete semantics [45]. To strengthen this claim, we validated our abstract transition rules for EH against the concrete implementations of the corresponding library functions. This was done by *fuzzing* dynamically instrumented test binaries using generated abstract start and end states. In this fuzzing process, abstract start states are first concretized to actual CPU states, then run against the function under test in an instrumented binary. Observation of the resultant CPU state allows verifying whether or not the end state from abstract execution corresponds to that of actual concrete execution. The only case where overapproximation is not guaranteed is when an indirection cannot be resolved. Such cases are clearly marked in the EICFG.

We applied the tool to 341 off-the-shelf binaries compiled from C++, C, and Fortran source code. The implemented tool was able to identify 3350 unique throws and successfully trace the exceptional control flow for every one of them. On average, dealing with exceptional control flow can increase coverage by 13 instructions per unique throw, with each throw averaging 379 unwind edges. Those edges are ones tools such as Ghidra [134] do not produce.

1.7 Limitations and Scope

None of the contributions in this dissertation cover the usage of concurrency or multithreaded code. We have scoped it out due to its complicated nature. Our ISA coverage is restricted to x86-64, as targeting multiple ISA would require additional state modeling or further levels of abstraction. Our ABI coverage is restricted to System V, as that requires assumptions about register invalidation during/after function calls. We also assume that the per-instruction semantics in use and the state changes they express are sound (for example, that they are semantics that have been machine-learned from actual hardware [84, 150]). This addition to the TCB is required as for the most part we do not directly validate those semantics ourselves. Completely eliminating semantics from the TCB would require testing against actual hardware, as done for some library functions in Section 12.1.

Additionally, due to the aforementioned undecidability, none of these approaches are universal. They may fail on certain functions/binaries or need to annotate certain instructions with unsoundness warnings.

1.7.1 Memory Usage

Due to the amount of user effort required, the memory-usage-focused contributions in this dissertation only target individual functions or small collections of functions. They also require an external tool to generate the disassembly under test. This disassembly must be loaded into the theorem prover from an external file or manually copied for the CFG-driven work. In the case of the Hoare-style work, it is directly copied into the generated theory files. Neither approach has support for EH either, though the relevant library functions can be modeled as no-ops or terminators as needed. For modeled functions, we assume

Additionally, for the Hoare-style work specifically, we had to rule out *recursion* as it was not a feature well-suited to automation in that framework. Usage of |goto| is also not supported due to not easily fitting into SCF structure. On the plus side, recursion seems to be uncommon in systems code, as we encountered only one function with it in the case study for that contribution. Further limitations of the Hoare-style approach can be found in Section 5.4.

1.7.2 Control Flow Recovery

In order to reduce state space complexity, provide focus, and provide scalability when recovering control flow, we also:

1. target potentially-stripped COTS Executable and Linkable Format (ELF) binaries compiled with various levels of optimization;
2. do not deal with any functions executed after an exit or termination; and
3. assume that all memory regions accessed by the binary are either aliasing, separate or enclosed [8, 9].

Lastly, we assume the existence of a fetch function that, given an address, soundly retrieves the corresponding instruction from the binary.

Experimental results show that the majority of unsoundness annotations concern function callbacks. In order to gain scalability, we treated function calls as context free. That allowed us to reduce the state space and also reuse previous executions. However, it also means that if a function pointer is passed as a parameter, its concrete value will be unknown.

External functions also require some assumptions. Specifically, we assume that external calls properly follow the System V ABI and do not interfere with their parent stack frames. This means that those registers considered volatile are invalidated when we encounter such calls, but we do not have to invalidate most other components of the state. An in-depth exploration of function call handling comes later in Section 7.4.2.

Finally, to deal with the final assumption regarding aliasing/separation/enclosure, we again utilize overapproximation. In cases where we cannot obtain clean arrangements of regions and would instead require overlapping ones, we merely invalidate the regions under consideration. This ensures that the result of accessing such regions will just be $\bot$, allowing us to avoid engaging in generating innumerable states for all possible overlapping arrangements. From our

experiments, this did not result in significant loss of the information necessary to successfully complete our analyses.

For EICFGs specifically, we assume that no external calls throw exceptions themselves. We also do not model `setjmp`/`longjmp`, as they do not interact well with structured EH [44]. Of course, the HG work does not support EH, though it can still analyze binaries containing the corresponding library calls.

1.8 Organization of Dissertation

Following this introduction in Chapter 2 is a review of tools and work related to the field of assembly-level verification, control flow lifting, and software correctness in general. 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 3.

After that, the Floyd-style approach to verification of memory usage mentioned above is presented in Chapter 4 while the Hoare-style approach is presented in Chapter 5. I then have the aforementioned work for the lifting of formal control flow graphs in Part III followed by the interprocedural, exception-aware CFG extractor in Part IV. Finally, this dissertation wraps up in Chapter 13.

Chapter 2

Related Work

Verification of assembly has been an active field of research for decades. Binary analysis in general, both static and dynamic, has a rich history as well. 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 in which assembly verification played a role in a larger verification context, Section 2.2. Table 2.1 provides an overview of the assembly and integrated projects. It also includes the first two works presented in this dissertation.

Switching tracks to a more structural organization for the second half of this dissertation, we have the following two sections. First, a set of works covering the lifting or extraction of control flow and state machines in Section 2.3, followed by a coverage of tools that perform exception analysis in Section 2.4 (with bottom-up approaches summarized in Table 2.3).

2.1 Assembly-Level Verification

Clutterbuck [41] and Clutterbuck and Carré [42] 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) [33], 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 verification condition generator (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 Floyd-style approach presented in Chapter 4 does, with additional assertions on state to avoid errors and stronger conditions in order to prove functional correctness. They also provided *rewrite rules* that described the semantics of the formal models in SPADE. Unlike the work detailed in Chapter 4, 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 [136]. 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, Boyer and Yu [26] and Yu [194] presented operational semantics and mechanized reasoning for approximately 80 %, or around 85, of the instructions of the MC68020 microprocessor ISA. They implemented those semantics and mechanized their approach in the Boyer-Moore theorem prover (Nqthm) [25], a precursor to the theorem prover ACL2 [98]. 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 usage 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 that is a significant difference.

Building on the rely-guarantee[1] scheme for verification of concurrent systems introduced by Xu, Roever, and He [191] some years before, Yu and Shao [193] developed a logic-based type system for analysis of concurrency on the assembly level. It received a fully mechanized implementation verified for soundness in Coq. This mechanization allows for the semi-automated verification of undecidable safety properties such as mutual exclusion, deadlock freedom, and partial correctness, much like the usage of semi-interactive proving to deal with undecidable properties in Chapters 4 and 5 and Part III.

Following that, Matthews et al. [121] targeted a simple machine model called TINY as well as the M5 operational model of Java virtual machine (JVM) 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 conditions (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 cipher block chaining (CBC)-mode encryption and decryption. In total, they covered 631 assembly instructions, less than that handled by any of the methodologies presented in this dissertation. Of course, they were targeting a stronger property than 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 Floyd-style approach presented in Chapter 4 of any of the works presented in this chapter, as they even implemented a version in Isabelle. Their Isabelle version did not support compositionality,

[1] or "assume-guarantee", as the paper calls it

however.

Additionally, Goel [76] and Goel et al. [77] presented an approach for modeling and verifying non-deterministic programs on the binary level. As with Matthews et al. [121], 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 Part II of 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 automated approach AUSPICE provided by Tan et al. [169] takes about six 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 5, they too used a full set of Hoare rules in their analysis.

Though it is not a verification methodology by itself, DiL provides the means by which a program can be verified in a theorem prover. Developed by Myreen et al. in the HOL4 theorem prover [166], 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 [126]. It formally covers the gap between machine code and an 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 [160].

Also, Feng et al. [62, 63] presented stack abstractions for modular verification of assembly code in the Coq theorem prover [35]. Their work allows for integration of various proof-carrying code systems [130]. As with the work presented in Chapter 5, 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 5, though not that in Chapter 4, manual annotations are required to provide information regarding invariants and memory layout.

Schlich [156] worked on the development of a model checker for analysis of microcontroller assembly, [MC]SQUARE. Though not designed formally from the ground up (it was implemented in Java rather than a theorem prover), it supports several microcontroller ISAs and 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 [157]. The three programs Schlich, Salewski, and Kowalewski 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. 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.

Remark 2.1 (Function isolation in our works). The case study on HermitCore in Section 4.5 did involve isolating the functions under test before compiling them and covered less instructions. By contrast, the analyzed Xen functions in Section 5.4 were handled without any modification whatsoever to the Xen build process and covered even more instructions. Building on that, Part III had a combination of full-binary and per-function analyses with even more instructions for its Xen case study (Section 8.1), while Part IV was primarily per-binary but had unsound additional coverage via per-function fallbacks and went far beyond just Xen for its case study (Section 12.2).

Brauer et al. [28] 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 (DVR), which [mc]square uses for state space reduction. While this stack safety approach is similar in focus to the memory usage works I present here, the scope is much smaller. Their focus was specifically on stack memory rather than memory in general.

Several years ago, Fromherz et al. [72] embedded a subset of the x86-64 ISA in the functional, verification-oriented language F* [61]. 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) [14] and AES-Galois/Counter Mode (GCM) [59]. 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 [24]. This was done in the style of *proof by reflection* [15]. 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 4 and 5, 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 5 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 Efforts Involving Assembly-Level Verification

A major verification effort based on DiL was the verification of the seL4 kernel [105, 106]. The seL4 project provides a microkernel written in formally proven correct C code. The tool AutoCorres is used for C code verification [80]. Sewell, Myreen, and Klein [160] 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. 5.7a): their approach requires a static separation of stack and heap instead.

Shi et al. [162] 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 (CSP) model analysis [88]. Binary verification was done by lifting the RTOS binary to xBIL, a related hardware verification language [163]. 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 [164]. 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 lines of x86-64 assembly 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 5, Dam et al. [49] and Dam, Guanciale, and Nemati [50] 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 [153]. Their methodology integrated HOL4 with the Binary Analysis Platform (BAP) [29]. 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 [71] 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) [74] 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. [12], who did a similar proof for the ARMv8-A model provided by Fox [70].

Finally, Bevier et al. [19] presented a systems approach to software verification that targeted correctness all the way down to the hardware level. All of their work was implemented

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 4	x86-64	ITP+VCG	HermitCore	2379+ insts
Chapter 5	x86-64	CG,ITP,VCG	Xen	12 252 insts
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+UCs	PROSPER	3000 insts
Baumann et al.	ARMv8-A	ATP+UCs	HAPSOC	8000 SLOC
Bevier et al.	PDP-11-like	ITP+TV	Full system	3k+ SLOC/insts

in Nqthm. Hunt [90] developed a general-purpose, 32-bit microprocessor, FM8502, and proved that its gate-level specification was an implementation of its formal ISA. Bevier [16–18] 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 [192] designed and proved the correctness of a code generator, a major compiler component, for a subset of the Gypsy 2.05 programming language [79]. That code generator's output was the verified, high-level assembly language Piton [124]. Moore [123] then proved the correctness of that language's FM8502 implementation.

2.3 Control Flow Recovery

We relate our work for HG lifting to existing approaches for disassembly (Section 2.3.2), binary decompilation (Section 2.3.3), and binary verification (Section 2.3.4). To the best of our knowledge, the only existing work that has a similar focus on disassembly based on formal methods is Jakstab [101–103], which we therefore discuss in more detail.

Table 2.2: Overview of disassembly and decompilation approaches

Approach	Formal?	Overapprox.?	Methodology	Case study
Jakstab	Yes	No	AbsInt	Windows drivers
`objdump`	No	No	linear sweep	
BYTEWEIGHT	No	No	Probabilistic	Various bins
Wartell et al.	No	No	Probabilistic	Handpicked bins
Miller et al.	No	No	Probabilistic	Various bins
Spedi	No	No	Probabilistic	Benchmarks
`dcc`	No	No	Dataflow+CFG+idiom	Handcrafted bins
`asm2c`	No	No	Dataflow+CFG+idiom	Benchmarks
Mocha	No	No	Graph manip.	
Krakatoa	No	No	Graph manip.+rewrite	Handpicked files
RetDec	No	No	Recursive traversal	
Phoenix	No	No	Semantic preservation	Coreutils
FoxDec	Yes	Semi	Rewriting	recompilation
SmartDec	No	No	C++-specific	
McSema	No	No	CFG analysis	
Ramblr	No	No	CFG analysis	
CodeSurfer/x86	No	No	Interactive	
IDA Pro	No	No	Heuristics	
Binary Ninja	No	No	Heuristics	
Ghidra	No	No	Heuristics	
Katz et al.	No	No	RNNs	
DiL	Yes	No	Rewriting	
Part III	Yes	Yes	Non-det. mem+join	Xen Project
Part IV	Semi	Yes	EH handling	Various bins

2.3.1 Jakstab

Jakstab performs binary analysis and control flow reconstruction by utilizing *abstract interpretation* [46, 47]. Its main analysis was designed for binaries with potentially handcrafted, obfuscated behavior (such as Windows device drivers and malware), and much like our HG lifting, it uses a form of at-will disassembly. This means that only specific series of bytes queried from specific locations in the binary are disassembled.

Unlike our lifting work, Jakstab often requires usage of manually-coded harnesses for binaries. A harness provides property specification and additional intermediate operations not found in the actual binary, and possibly external call modeling. They may additionally be used to provide pointer initialization for library/driver code. However, an imprecise harness may lead to false positives requiring manual investigation, and that kind of harness is impossible to create precisely for COTS binaries, such as the binaries used in our case studies. Because of this, Jakstab offers a variety of heuristics to improve scalability and precision at the cost of making the results possibly unsound [101, p. 129].

Additionally, we argue that Jakstab is not overapproximative even for those sound cases. That is, even when no indirections occur, it will not always reach all instructions that are actually reachable in a binary. The instructions and states that are reached by Jakstab are *relative to the harness*; that is, relative to some initialization and external information. In contrast, we make much fewer assumptions about the initial state and conditions. Furthermore, Jakstab's average coverage is only 15 % of the instructions that are present in a binary [101, Table 6.2]. This percentage is computed exclusively over the case studies where it reports a complete and successful result (no counterexamples found for the property being checked). In general, programs do not normally exhibit that much dead code. Therefore Jakstab appears to underapproximate rather than overapproximate.

Also, Jakstab takes a slightly different approach to calculating least fixed points (LFPs). Rather than performing a join operation that may produce range bounds, it instead keeps track of a certain number of values for each "variable" (register, memory location). Once that number is exceeded, the possible values are widened to an unknown (akin to $\bot$), though pointer type is preserved when possible. This operation does not appear to be utilized when control flow recombines after conditional statement divergence, however. Because of that, it results in a larger state space overall than our HG lifting work, though potentially a more precise one.

Related to this is Jakstab's memory handling approach. Unlike our more fine-grained memory model approach (described later in Section 7.3), it is all-or-nothing. This means it cannot handle writes to fully-unknown (or on-the-stack-but-exact-location-unknown) memory regions, as it will end up with overwritten return addresses in all such cases. While Jakstab can be configured to continue execution in such scenarios, the developers considered the results too imprecise for practical usage. Our lifting tool, meanwhile, is able to handle most such situations as it provides memory models for standard aliasing conditions.

Remark 2.2 (Issues using Jakstab). We were not able to directly compare the behavior of Jakstab with our lifting mechanism. Jakstab on its own did not work due to unsound

assumptions on the structure of ELF binaries [133]. This is likely due to its development being primarily focused on the handling of Windows programs. Even so, while were able to successfully test with a modified version [142, 143], we were not able to do a comparison as Jakstab does not support 64-bit binaries.

2.3.2 Disassembly

Disassemblers are tools that take a binary and lift it to an assembly language. Traditionally, there are two main methods of disassembly: *linear sweep* and *recursive traversal* [159]. Modern disassemblers may combine the two or use other techniques like probabilistic [122, 186, 187] or conflict analyses [13].

Linear sweep disassemblers, such as GNU's `objdump`, scan through a binary linearly in areas where code is typically encountered to extract instructions [159]. Linear sweep algorithms are easy to implement but well-known to be unsound [159]. ISAs with variable-length instructions/non-strict alignment requirements, such as x86, pose problems to such disassemblers when complex indirect jumps are involved. Intermixed code and data also pose a problem. The lack of reachability analysis can also cause problems for code after function calls even when they are calls to internal functions. This is because, unless the disassembler does multiple sweeps, it cannot identify non-returning functions, though heuristics may be able to help with making reasonable assumptions.

Recursive traversal algorithms are more complex than linear sweep. They operate by starting from some initial instruction and then trace the possible paths of execution, interpreting instructions as they proceed [110, 159]. Our work in Part III is an example of a recursive traversal disassembler, with Part IV using a similar methodology. This allows higher accuracy than linear sweep. For example, such disassemblers support cases where parts of instructions can be interpreted as other instructions, possible with ISAs having unaligned/variable-length instructions. They will also be more likely to exclude code that is not executed. The major challenge of recursive traversal is properly dealing with indirect calls and jumps. A tool that primarily uses recursive traversal is IDA Pro [85], intended for interactive debugging and reverse engineering. Typically, existing approaches to recursive traversal use heuristics or guesses to approximate indirect branches.

Probabilistic approaches to disassembly [13, 122] include machine learning techniques, such as BYTEWEIGHT [10] or the works of Wartell et al. [186, 187]. In general, these techniques attempt to identify sequences of bytes as instructions based on their context, using large amounts of *training sets* as a guide. The approach of Miller et al. [122] identifies the probability of byte sequences representing instructions based on the context. It uses manually derived *hints* rather than training sets. Spedi by Ben Khadra, Stoffel, and Kunz [13] provides a speculative, conflict analysis-based approach. This approach involves scanning through a binary and speculatively extracting basic blocks of instructions and then eliminating infeasible possibilities via a conflict analysis process. While it is able to handle switch statements (jump tables), it does not handle other types of indirection. The main disadvantage of probabilistic/speculative techniques is that they inherently cannot provably overapproximate the behavior of the binary. Although they typically have very few cases of underapproximation,

such cases are not impossible.

The key differences between our lifting work and existing disassemblers are that:

1. no existing disassembler aims at providing a guarantee that the lifted representation is a sound overapproximation of the binary; and
2. our approach goes beyond disassembly, providing both control flow and invariants that are sufficiently strong enough to prove control flow.

The cost of our approach is that it may fail, whereas other approaches are able to guess or use heuristics to continue.

2.3.3 Binary Decompilation

As mentioned in the introduction to this dissertation, a decompiler takes a binary or other low-level source as input and lifts it to a higher-level representation. An early attempt at such lifting to full source code level was that of Cifuentes and Gough [37]. The `dcc` tool resulting from that work lifts 80286 assembly to C code, though its internal representation can support multiple higher-level languages. In fact, it performs a similar sort of syntactic control flow reconstruction as the Hoare-style work presented in this dissertation. That does leave it with similar limitations to that work, however. Cifuentes, Simon, and Fraboulet [38] also provided a similar translator for Scalable Processor Architecture (SPARC) assembly, `asm2c`, soon after.

Decompiling Java bytecode to source code was of interest starting in the 90s as well, with tools such as Mocha [177] and Krakatoa [146]. Both tools utilize graph transformations to extract information, with Krakatoa providing additional rewriting rules and greater coverage. These tools are interesting as, unlike many register-based ISAs, including the one we target here (x86-64), the JVM is a stack-based architecture with pseudo-register local variables for method arguments and other purposes.

More recent works such as RetDec [5], Phoenix [30], and FoxDec [182] all aim at lifting a binary to C code as well. Meanwhile, the SmartDec tool [68] lifts to C++ code, whereas McSema [56] lifts to LLVM. Ramblr [184] lifts a binary to *symbolized* assembly, where concrete addresses are replaced with symbolic labels. CodeSurfer/x86 [8, 9] provides a graphical interface for lifting binaries to an intermediate representation and interactively analyzing them. Additionally, decompilers are often integrated into reverse engineering and program exploration tools such as IDA Pro [86], Binary Ninja [189], and Ghidra [134].

Some decompilers even rely on techniques of machine learning, much like disassemblers. The field of natural language processing in particular has provided inspiration for the processing of formal languages. For example, Katz, Ruchti, and Schulte [97] applied recurrent neural networks, which are often used for text processing and generation, to the problem of decompilation.

Despite all this, a key factor in many decompilation approaches, and also in other approaches that aim at producing CFGs or dataflow analyses, is that they rely on the disassembly and CFR being done by an external tool that is assumed to be sound. Our algorithm thus

complements these works. For example, McSema requires an external source of control flow information, which could be generated from our solution. Many loop-identifying algorithms used by decompilers require a CFG as well [83, 188]. `dcc` notably performs its own disassembly, but that is not the prevailing trend.

For a more formal approach, we return to DiL [126, 127]. However, DiL does not deal with indirect branching and *assumes* that return addresses are not overwritten. In their own words, "[its] heuristic is easily confused by computed branches" [128]. In contrast, our HG lifting approach supports various forms of indirect branches and can detect potentially-overwritten return addresses.

2.3.4 Relation to Binary Verification

As a reminder, binary verification techniques aim to prove properties on the machine code level [111]. Typically, binary verification aims at proving that the binary is correct with respect to some higher-level artifact (source code or a specification). Klein et al. [106], Klein, Sewell, and Winwood [107], and Sewell, Myreen, and Klein [160] used a refinement-based approach to verify the seL4 microkernel binary. Kamkin et al. [96] developed a methodology for verifying that the machine code of RISC-V binaries satisfy annotations in the binaries' source code. For a top-down approach, proof-carrying code [130] integrates a proof of correctness into the binary that is verified at runtime. If the proof fails, the binary cannot be executed. This does require some additional functionality, with both the compiler and the program host needing to support it.

In contrast, our HG lifting approach is targeted at the scenario where a higher-level artifact such as source code or a specification is not available. In such contexts, most approaches are interactive [76, 77, 181]. Even our Hoare-style work presented in Chapter 5, tailored to memory usage properties, has a "manual effort vs. instruction count ratio" of roughly 1 to 11 [180]. While the earlier-mentioned AUSPICE is fully automated, it is also very slow Tan et al. [169]. Our HG lifting approach therefore *complements* formal verifiers that generate invariants for proving functional correctness.

Remark 2.3 (Lifting in the TCB). The invariants this lifting approach generates do not contain enough information to prove full functional correctness; they are tailored for control flow and pointer relations. The tool instead serves to remove the lifting process from the TCB. This is useful because verification based on untrustworthy disassemblers and control flow reconstruction is itself untrustworthy.

2.4 Exception Handling Analysis

The two main approaches to analyzing exceptional behavior for C++ programs, and behavioral program analysis in general, are bottom-up and top-down. Bottom-up tools include decompilers and disassemblers, as well as this work. Top-down tools focus on analysis of source code or other higher-level representations.

Table 2.3: Bottom-up exceptional analysis comparisons

Program	Intraprocedural†	Interprocedural‡	Academic Evals
EICFG work	**Statically**	**Statically**	
Binary Ninja	Statically*	Dynamically	
IDA Pro	Statically	Dynamically	[73, 117]
Ghidra	Statically	Dynamically	[117, 140, 151]
McSema [112, 175]	Statically	No	[51, 140]
RetDec	Unknown	No	[117]

† Can it identify the landing pads in a function?
‡ Can it trace from (re)throw to landing pad?
* Via https://github.com/EliseZeroTwo/SEH-Helper

2.4.1 Bottom-Up Approaches: Decompilers and Disassemblers

We provide a summary of such works in Table 2.3. The table describes whether or not the works do intra- or interprocedural exception analysis and if so, if it is static or dynamic. Static approaches, such as the one presented in this paper, typically aim for overapproximation. They utilize abstraction or other methods to model paths symbolically. In contrast, dynamic approaches are inherently underapproximative. This is because they rely on concrete runtime behavior and evaluating all possible concrete paths is infeasible. Even works that hijack runtime control flow to force specific path execution must do it in a sampled fashion [190].

While Binary Ninja [178], IDA Pro [85], and Ghidra [134] all support some form of intraprocedural EH analysis, they can only perform interprocedural EH analysis dynamically via debugging. For example, Ghidra provides default, platform-dependent analyses that extract try-catch block information and other landing pad information for EH. However, it does not provide unwinding control flow in the generated block/call graphs, only cross-references for the LPT. This means that it can identify landing pads and analyze the code following them, but it cannot identify the exceptions that will reach them. Its built-in debugger that can perform unwinding is also ultimately just an interface to external, dynamic debugging tools such as GDB or LLDB [171], which perform dynamic analysis and instrumentation. Traces are also supported, but those require the program to have been run previously.

McSema [175] is back too! It is relevant because its decompilation process lifts machine code to LLVM bitcode. This allows providing intraprocedural exception analysis by generating the LLVM representation for exception landing pads [112]. However, it does not perform interprocedural analysis as it merely lifts to LLVM bitcode rather than tracing the execution of exceptions from throw site to landing pad.

RetDec [5] functions similarly to McSema as a decompiler-to-LLVM, though it also supports C output. However, we could not find information about its ability, or lack thereof, to deal with EH. We also were unable to successfully apply it to programs using C++ exceptions. Furthermore, even when it does work it suffers from accuracy issues and produces non-

semantically-equivalent results, many of which do not even run [69]. As with McSema, it does not perfom interprocedural exception analysis.

2.4.2 Top-Down Approaches

By contrast, there are tools that analyze exceptions from the source-code side. This prevents the analysis of legacy code without source but allows for better static analysis during development, or even formal proofs of correctness of the exceptional semantics.

Hutton and Wright [91] provided basic formal semantics for source-level C++-like exceptions. They accompanied this with a compiler for a small language with exceptions and a proof of correctness of that compilation. This is different from our approach as we do not aim to verify the correctness of exceptional behavior due to our lack of ground truth (source code or some program specification).

Prabhu et al. [144] provide source-level generation of interprocedural exception control-flow graphs (IECFGs) for C++ exceptions. These IECFGs are much like our EICFGs, but for source code. Unlike our work, however, IECFGs are used to *eliminate* exceptions when compiling to a binary to make static analysis easier. They cannot be used to analyze already-compiled programs with exceptions.

Java programs use exceptions as well. Kechagia et al. [99] provide a tool for identifying unhandled application programming interface (API) misuses via static exception propagation and test case generation. This differs from our work as we detect in-program exceptional behavior, not exceptions produced by API or other external calls.

2.4.3 Tools that use `libunwind`

A standard library for instrumented/dynamic unwinding is `libunwind` [125]. It has been utilized by tools such as RockJIT [132], which protects just-in-time (JIT) compilers by enforcing CFI. It can function without source code. However, as an instrumenting library, it requires a running program and thus must operate dynamically.

Our final tool, as well as all of the other bottom-up analysis tools mentioned above, assumes both the existence and correctness of the `.eh_frame` and `.gcc_except_table` ELF sections. However, neither of those cases are guaranteed. To deal with such scenarios, Bastian, Kell, and Zappa Nardelli [11] provide tools for the validation as well as synthesis of exception-handling-related table-based unwinding information.

2.5 Summary

This section covered some of the work related to that presented in this dissertation. Previous assembly-level formal verification efforts as well as verification efforts containing assembly or binary analysis components were discussed. We also covered various tools and approaches for lifting or extracting CFGs/state machines from binaries as well as ones that perform exception-aware analyses.

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 4, none appear to have achieved the 12 252 verified instructions covered in Chapter 5. We then built further on that with our approaches in Part III, covering 399 771 instructions, and Part IV, covering 4 715 806 instructions.

Part II

Methods of Analyzing Memory Usage

Chapter 3

Symbolic Execution

This chapter covers the methodology used in Chapters 4 and 5 to formally determine the state changes caused by individual basic blocks. That methodology relies on a formal big-step semantics of the x86-64 ISA provided by Roessle, Verbeek, and Ravindran [150], described in Section 3.1. We then extended those semantics with additional rewrite rules to increase efficiency and properly reason about memory. Those rules are documented in Section 3.2. The rules involving reading and writing from memory form the basis for the memory usage methodologies in Chapters 4 and 5. Essentially, they generate memory region VCs that must be discharged in order to prove memory usage.

Example 3.1 (Aggregation)**.** Consider the following two instructions:

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

These instructions write to the 64-bit register `rax`. Registers `ax` and `al` refer to the low 16 and 8 bits of that register respectively. 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.

3.1 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 [150]. They embedded a bitvector-based, big-step semantics machine-learned from a modern version of the x86-64 ISA in Isabelle/HOL.

That semantics included instruction set extensions such as the Streaming SIMD Extensions (SSE) 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. [84], 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 4 and 5 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 [150]. 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`.

3.1.1 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 part, 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.

3.1.2 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 3.2 (Reading part of a region)**.** 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, Performance Optimization With Enhanced RISC – Performance Computing (PowerPC), SPARC, and Microprocessor without Interlocked Pipelined Stages (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.

3.2 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 [150] 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 3.1.

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 [181]. 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.

3.2.1 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 3.3 (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 3.4 (Enclosure)**.** Region r_0 is *enclosed* by r_1, notation $r_0 \preceq r_1$, 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.

3.2.2 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 3.5 (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

$$r = [a, s] \coloneqq b_0 \bullet b_1 \bullet b_2,$$

[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 into the right, not the right into the left as the Isabelle version does.

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 3.6 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. (3.1). 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.

$$
\sigma(r_0 \coloneqq v_0)(r_1 \coloneqq v_1) \equiv \begin{cases} \underline{\sigma(r_1 \coloneqq v_1)}(r_0 \coloneqq v_0) & \text{if } r_0 \bowtie r_1 \\ \underline{\sigma((r_0 \coloneqq v_0) \text{ merge } (r_1 \coloneqq v_1))} & \text{otherwise} \end{cases} \tag{3.1}
$$

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 3.1 as our guide. If an assignment to the exact region $[a, s]$ exists in the current 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

Algorithm 3.1 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 $\ldots$ merge ovl_1 merge ovl_0
    $\alpha' \leftarrow \{[a', s'] \coloneqq v'\} \cup sep$
    $a'' \leftarrow 8(a - a') - 1$
    return $(\alpha', \langle s + a'', a'' \rangle v')$

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 3.6 (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 $\texttt{rsp} = \texttt{rsp}_0$, the set M of memory regions required for the given block of assembly is ultimately

$$M = \{[\texttt{rsp}_0 - 8, 2], [\texttt{rsp}_0 - 4, 4], [\texttt{rsp}_0 - 7, 2], [\texttt{rsp}_0 - 6, 4], [\texttt{rsp}_0 - 8, 8]\}.$$

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 3.1, 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 3.7 (Separation)**.** 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:

$$\bigotimes M \equiv \forall (i_0, r_0), (i_1, r_1) \in M \cdot \text{if } i_0 = i_1 \text{ then } r_0 = r_1 \text{ else } r_0 \bowtie r_1 \tag{3.2}$$

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.

$$r \preceq r \tag{3.3a}$$
$$r_0 \bowtie r_1 = r_1 \bowtie r_0 \tag{3.3b}$$
$$r_0 \preceq r_2 \wedge r_1 \preceq r_3 \wedge r_2 \bowtie r_3 \longrightarrow r_0 \bowtie r_1 \tag{3.3c}$$
$$r_0 \preceq r_1 \wedge r_1 \preceq r_0 \longrightarrow r_0 = r_1 \tag{3.3d}$$
$$r_0 \preceq r_1 \wedge r_1 \preceq r_2 \longrightarrow r_0 \preceq r_2 \tag{3.3e}$$
$$r_0 \bowtie r_1 \wedge \operatorname{snd} r_0 \neq 0 \wedge \operatorname{snd} r_1 \neq 0 \longrightarrow r_0 \not\preceq r_1 \tag{3.3f}$$
$$\bigotimes(M) \wedge (i_0, r_0), (i_1, r_1) \in M \wedge i_0 \neq i_1 \longrightarrow r_0 \bowtie r_1 \tag{3.3g}$$

Equation (3.3) 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 3.4. Such inequalities can be solved using the Isabelle/HOL tool `unat_arith`, which is an arithmetic equation solver for bitvectors [53, 54]. 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 4, 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. (3.3).

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 5 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.

3.3 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 4

Floyd-Style 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 usage over individual assembly functions. That property ensures that only documented regions of memory are read and written; anything outside of those regions remains unchanged.

This Floyd-style 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 4.1, with formal definitions of the needed constructs presented in Section 4.2. Following that, Section 4.3 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 4.4, while a real-world application to the HermitCore unikernel library [114] is presented in Section 4.5. Our observations regarding usage of the methodology on that case study are given in Section 4.6.

4.1 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 [184] of the binary analysis tool `angr` [165, 183]. That modified version was provided by Roessle, Verbeek, and Ravindran [150] 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 4.2.3. Basic starting predicates for those preconditions and postconditions as well

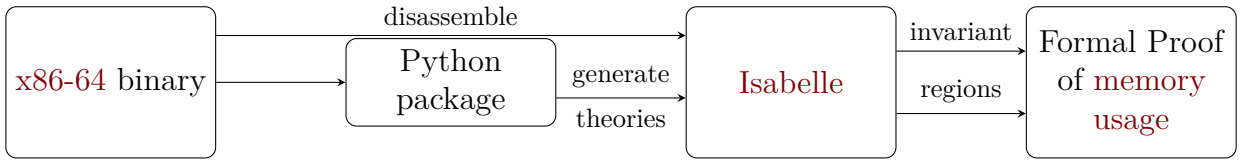


Figure 4.1: Overview of Floyd-style memory usage verification

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 usage theories for every function analyzed. The theory files can then be opened in the interactive theorem prover Isabelle and the assembly loaded using the parser of Roessle, Verbeek, and Ravindran [150]. Once that is done, a user can flesh out the invariants (Section 4.2.3) and add the necessary sets of memory regions that the functions write to in order to complete the proofs of memory usage. Defining the necessary invariants for functions with complex control flow is generally a hard task, but targeting a property such as memory usage does reduce the amount of work required as seen in Sections 4.4 and 4.5. The work is still not trivial, however.

4.2 Formal Definitions

A formal definition of memory usage requires a formal basis to work with, and that basis is the machine model from Section 3.1.

4.2.1 Symbolic Execution for Floyd-Style Verification

While Chapter 3 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 $\bot_{\mathrm{E}}$ instead.

From the machine model, we manually derived a run function $\textsc{runUntil} : (S \to \mathbb{B}) \times S \to (S \mid \bot_{\mathrm{E}} \mid \bot_{\mathrm{NT}})$. 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 $\bot_{\mathrm{E}}$. 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 $\bot_{\mathrm{NT}}$. Formally, this is achieved by a standard LFP construction.

4.2.2 Hoare Triples for Memory Usage

Unlike the usual formulation of Hoare logic [87, 126], 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 4.1 (Hoare triple for memory usage)**.** $\{P\}H\{Q\}$ 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:

$$\{P\}H\{Q\} \equiv \forall\sigma \cdot P(\sigma) \wedge \sigma' \neq \bot_{\text{NT}} \longrightarrow \sigma' \neq \bot_{\text{E}} \wedge Q(\sigma'), \tag{4.1}$$

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

4.2.3 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 [137, 155], 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 our control-flow-based approach akin to the aforementioned Floyd verification [67]. 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 4.1 can be inferred for the program as a whole (Theorem 4.3).

More formally, the *Floyd invariant* for a function is a partial function that takes the form $I : \mathbb{A} \rightharpoonup (S \to \mathbb{B})$. 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 4.2 (Floyd invariant)**.** A Floyd invariant I *holds* if and only if, for any state σ,

$$I(\text{loc}\,\sigma)(\sigma) \longrightarrow \sigma' \neq \bot_{\text{E}} \wedge (\sigma' = \bot_{\text{NT}} \vee I(\text{loc}\,\sigma')(\sigma')), \tag{4.2}$$

where $\sigma' = \text{RUNUNTIL}((\lambda\sigma_r \cdot I(\text{loc}\,\sigma_r) \neq \bot), \sigma)$and $\text{loc}\,\sigma_r$ 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 4.3 (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, $H(\sigma) \longrightarrow \text{loc}\,\sigma = l_f$. Then $\{I(l_0)\}H\{I(l_f)\}$.*

Proof. Remember from Definition 4.1 that

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

Though there could be any number of additional annotations between l_0 and l_f, Floyd [67] 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:

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

As we have already assumed that I holds, we can substitute in the right side of the implication from Definition 4.2 to obtain

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

This then simplifies to

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

which is trivial. □

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

4.2.4 Definition of Memory Usage

The formal definition of memory usage takes the form of a Hoare triple from Definition 4.1. 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 usage 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 4.4 (Memory usage)**.** 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 usage* if and only if, for any address a and byte value v_0, the following implication holds:

$$(\forall r \in M \cdot r \bowtie [a, 1]) \longrightarrow \{P \wedge *[a, 1] = v_0\} H \{Q \wedge *[a, 1] = v_0\} \tag{4.3}$$

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.

4.3 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.

4.3.1 Intra-Function

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

Listing 4.1: 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 4.5 (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 4.6 (Compositionality)**.** Now consider the block of assembly that could be generated for Listing 4.1. Let l_f denote the final location of the block while l_P denotes the initial location of program P. Theorem 4.5 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.

4.3.2 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 $\texttt{rsp} = \texttt{rsp}_0 + 8$. 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 4.7 (Function calls)**.** 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 $\texttt{rsp}_0$. 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`): $*[\texttt{rsp}_0, 8] = l_{call} + 5 \wedge \texttt{rsp} = \texttt{rsp}_0$. 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: $\texttt{rsp} = \texttt{rsp}_0 + 8 \wedge \text{loc} = l_{call} + 5$. 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

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

Listing 4.2: `pow2` in C

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

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

    return a;
}
```

used as the target of a write after a function call. In order to prove memory usage, **rbp** must be shown to be preserved. Generally, this is easy to prove by strengthening the pre- and postcondition with a conjunct $\texttt{rbp} = \texttt{rbp}_0$. 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.

4.4 Examples

The following sections present some basic explanation of the procedure used in this chapter via two simple functions. The first, in Section 4.4.1, is a non-recursive function that features a loop. The second, in Section 4.4.2, gives an example of a recursive function and shows why those are difficult to reason about.

4.4.1 Non-Recursive Loop Example: `pow2`

This simple loop-based function, shown in Listing 4.2, raises two to the power of its argument. The assembly code was obtained by compiling a C program containing the function with the GNU Compiler Collection (GCC) 7.2.0 and disassembling it using the modified `reassembly` analysis mentioned in Section 4.1. 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 $\texttt{rsp}_0$: 1. the caller's **rbp** at $\texttt{rbp} = \texttt{rsp}_0 - 8$ (eight bytes); 2. the argument to the function at $\texttt{rbp} - \texttt{0x14}$ (four bytes); 3. the accumulation variable and return value at $\texttt{rbp} - 8$

Listing 4.3: 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
```

(eight bytes); 4. the counter variable at $\texttt{rbp} - \texttt{0xc}$ (four bytes); 5. and the address of the location to return to at $\texttt{rsp}_0$. The memory region for this function is thus $r_s = [\texttt{rsp}_0 - 28, 36]$. 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:

$$\text{parent}([\texttt{rsp}_0, 8], i_s, r_s) \qquad \text{parent}([\texttt{rsp}_0 - 20, 4], i_s, r_s)$$
$$\text{parent}([\texttt{rsp}_0 - 8, 8], i_s, r_s) \qquad \text{parent}([\texttt{rsp}_0 - 28, 4], i_s, r_s)$$
$$\text{parent}([\texttt{rsp}_0 - 16, 8], i_s, r_s) \qquad \text{parent}([a, 1], i_a, [a, 1])$$

For the memory usage 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 4.2 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 4.3, 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 3.

4.4.2 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

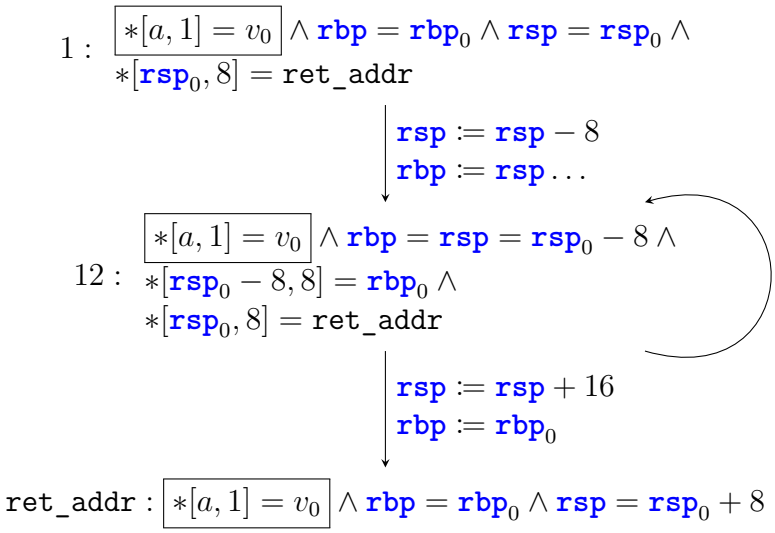


Figure 4.2: Floyd invariant for `pow2` in CFG form

Listing 4.4: Factorial in C

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

to n. Expressed in recursive form, that definition is:

$$n! = \begin{cases} n * (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases} \tag{4.4}$$

The C equivalent of that function is shown in Listing 4.4. The assembly snippet shown in Listing 4.5 is again the result of a function compiled with GCC 7.2.0 and disassembled by the tweaked `reassembly` analysis [184]. 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 $\mathtt{rsp}_0$, 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 $\mathtt{rsp}_0$ due to the function's recursive nature. The set M of separated parent regions is characterized by the following assumptions:

Listing 4.5: 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
```

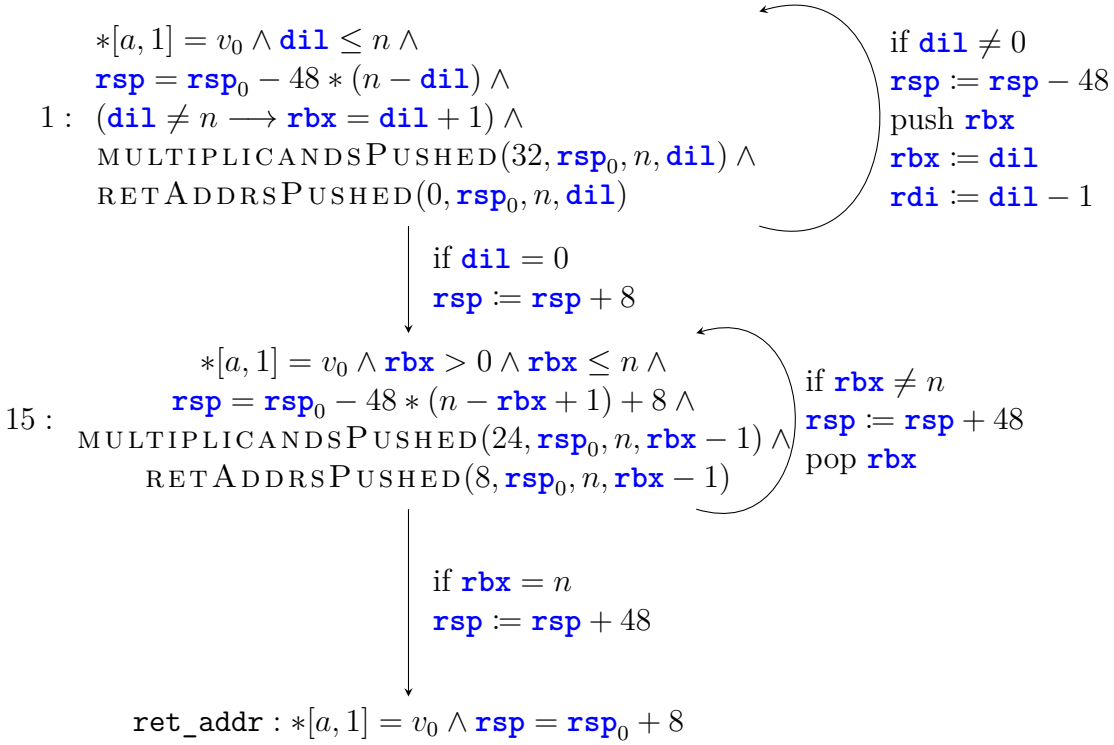


Figure 4.3: Floyd invariant for factorial in CFG form

$$\forall m \leq n \cdot (i_m, [\mathtt{rsp}_0 + 48 * m - (n * 48) - 40, 40]) \in M \tag{4.5a}$$
$$\forall m \leq n \cdot (i'_m, [\mathtt{rsp}_0 + 48 * m - (n * 48), 8]) \in M \tag{4.5b}$$
$$(i_a, [a, 1]) \in M \tag{4.5c}$$

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.

$$\text{MULTIPLICANDSPUSHED}(\mathit{offset}, \mathtt{rsp}_0, n, x) \equiv$$
$$\forall i \cdot n > x \wedge i < n - x - 1 \longrightarrow *[\mathtt{rsp} + i * 48 + \mathit{offset}, 8] = i + 1 + \mathtt{rbx} \tag{4.6}$$

$$\text{RETADDRESS}(\mathit{rsp}, \mathtt{rsp}_0, \mathit{offset}, n, x, i) \equiv$$
$$\text{if } \mathit{rsp} + (n - x - i) * 48 - \mathit{offset} = \mathtt{rsp}_0 \text{ then } \mathtt{ret\_addr} \text{ else } 15 \tag{4.7}$$

$$\text{RETADDRSPUSHED}(\mathit{offset}, \mathtt{rsp}_0, n, x) \equiv$$
$$\forall i \leq n - x \cdot *[\mathtt{rsp} + (n - x - i) * 48 - \mathit{offset}, 8] = \text{RETADDRESS}(\mathtt{rsp}, \mathtt{rsp}_0, \mathit{offset}, n, x, i) \tag{4.8}$$

The Floyd invariant for the factorial function is shown in Fig. 4.3 and Eqs. (4.6) to (4.8). 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 $\mathtt{dil} = 0$, the function has executed its final recursion and control reaches location 15. The second loop then pops values of the stack until $n = \mathtt{rbx}$, 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:

$$\begin{array}{l} \{*[a, 1] = v_0 \wedge \mathtt{rsp} = \mathtt{rsp}_0 \wedge *[\mathtt{rsp}_0, 8] = \mathtt{ret\_addr}\} \\ H \\ \{*[a, 1] = v_0 \wedge \mathtt{rsp} = \mathtt{rsp}_0 + 8 \wedge \mathrm{loc} = \mathtt{ret\_addr}\}. \end{array} \tag{4.9}$$

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

4.5 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 [118], 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 machines (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 [114] 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 [20].

4.5.1 Functions Analyzed

The functions from Hermitcore that were selected for analysis are summarized in Table 4.1. 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 memory model used in this dissertation does not support detection of access violations for

Table 4.1: 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

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.

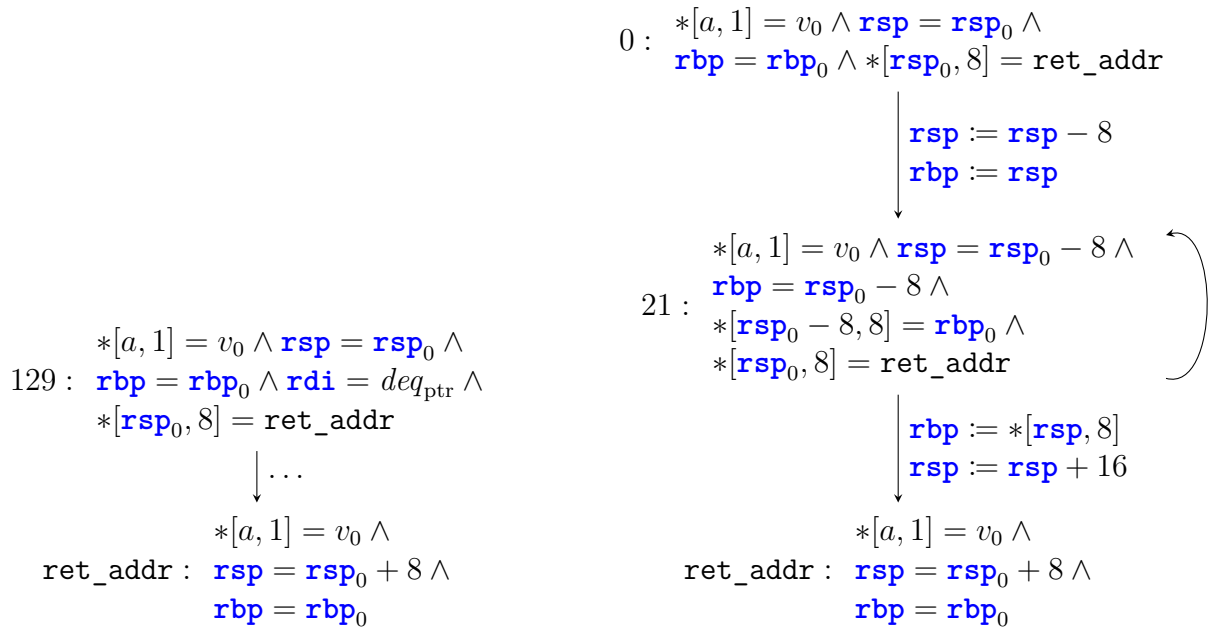
All of these functions were isolated and then compiled into binaries. Because of this, functions marked as `static inline` had those qualifiers removed. This prevented them from being eliminated when compiled with optimizations, as most of the functions would otherwise have their bodies inlined.

Figures 4.4a and 4.4b 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.

4.6 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 usage. 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 4.4: Floyd invariants for the described case study functions in CFG form

4.6.1 Defining the Invariant

While restricting the verification effort to memory usage 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 [145], 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.

4.6.2 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.

4.6.3 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 (AFP) [60] (though not necessarily efficiently). Recursion is the primary exception, with the proofs of stack and frame pointer preservation requiring significant ITP over word arithmetic.

4.7 Summary

This chapter covered one method for formal verification of memory usage 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 usage 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 usage 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 5

Hoare-Style Verification

While the methodology presented in the previous chapter for verifying memory usage 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 certificates (FMUCs) generated by untrusted, informal tools. FMUCs consist of two main components: theorems on memory usage 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 5.1, while the process of verification in Isabelle is documented in Section 5.2. A full example of FMUC usage can then be found in Section 5.3. 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 5.4 is an in-depth case study on the Xen Project hypervisor [34]. In total, FMUCs were generated and proofs discharged in Isabelle for 251 Xen functions.

5.1 FMUC Generation

FMUCs require the assembly code of a program as input. That source assembly could be obtained from a binary using a disassembler, such as `objdump`, IDA [85], Ghidra's decompiler [134], or Capstone [4]. If source code is available, it could be generated directly by a compiler

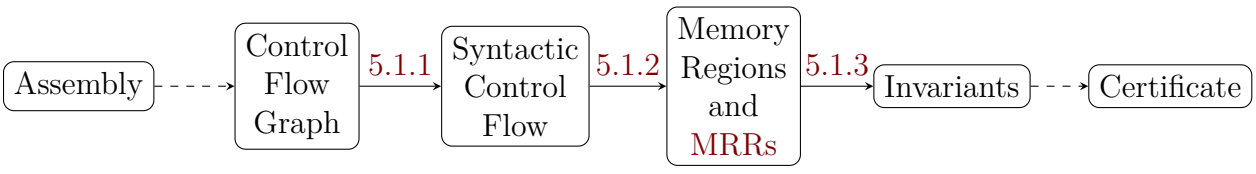


Figure 5.1: Overview of FMUC generation

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 5.2.5.

The general procedure for generating FMUCs, laid out in Fig. 5.1, 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 [165], ultimately producing an SCF (the details of which are presented in Section 5.1.1). 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 3. 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 [55]. Symbolic execution is also used in the process of generating the pre- and postconditions for each basic block, elaborated on in Section 5.1.3.

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 5.2) 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.

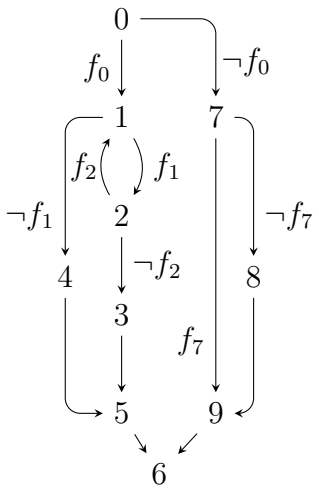
5.1.1 Control Flow Extraction

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 represented using the polymorphic type Φ.

$$\begin{aligned} \langle \text{scf} \rangle &\models \langle \text{scf} \rangle \texttt{ ; } \langle \text{scf} \rangle \mid \texttt{Block}\ \beta \mid \texttt{Skip} \mid \texttt{Continue} \mid \texttt{Break}\ \langle \text{br} \rangle \\ &\quad \mid \texttt{If}\ \Phi\ \texttt{Then}\ \langle \text{scf} \rangle\ \texttt{Else}\ \langle \text{scf} \rangle\ \texttt{Fi} \mid \texttt{Loop}\ \langle \text{scf} \rangle\ \texttt{Pool}\ \langle \text{res} \rangle \quad (5.1) \\ \langle \text{br} \rangle &\models \mathit{ID} \mid \epsilon \quad (5.2) \\ \langle \text{res} \rangle &\models \texttt{Resume \{(}\mathit{ID}\texttt{,}\langle \text{scf} \rangle\texttt{),...\}} \mid \epsilon \quad (5.3) \end{aligned}$$

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 EH as used in C++ is generally provided by external function calls.



(a) Example CFG

```
Block 0;
If f0 Then
    Loop
        Block 1;
        If f1 Then
            Block 2;
            If f2 Then Continue Else Break 2 Fi
        Else
            Break 1
        Fi
    Pool Resume {(2, Block 3), (1, Block 4)};
    Block 5
Else
    Block 7;
    If f7 Then Skip Else Block 8 Fi;
    Block 9
Fi;
Block 6
```

(b) Syntactic Control Flow

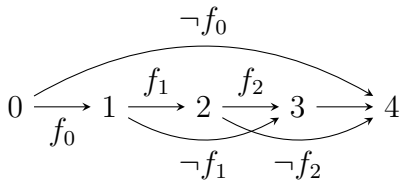
Figure 5.2: Example of control flow extraction

Example 5.1 (SCF extraction)**.** Figure 5.2 provides an example of SCF extracted from a CFG. The CFG in Fig. 5.2a 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. 5.2b 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 `Breaks` 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



(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 5.3: Example of code duplication

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.

SCF Explosion

As stated, the algorithm is not optimal in terms of generated SCF size due to the possibility of certain basic blocks being duplicated. There are two situations where such 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 5.2 (SCF explosion)**.** Figure 5.3 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.

5.1.2 Symbolic Execution for Generation

In Section 5.1.1, 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 $\mathrm{scf}(\mathcal{P}(A_{\mathrm{SP}}), E_{\mathrm{SP}})$, which is formulated over sets of assignments. It keeps track of all used memory regions, both the actual regions used 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 3.2.1, 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 3.3 and 3.4; the result states the property holds if the resultant problem is *unsatisfiable*. These SMT problems can be solved by Z3 [55] for a wide range of expressions over bitvectors using the `QF_UFBV` logic [147, 174]. Z3 is also used in this work for determining the sign of two values in the region merge algorithm, originally presented in Definition 3.5. Additionally, reads of overlapping regions may require merging and separation analysis as described in Section 3.2.1, 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 3.2.2.

5.1.3 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 5.3 (Initial invariant)**.** To reuse Example 3.6, its initial precondition would be:

$$\phi = \{\texttt{rip} \coloneqq \texttt{a0}, \texttt{rsp} \coloneqq \texttt{rsp}_0, [\texttt{rsp} - 8, 8] \coloneqq v_0, [\texttt{rsp}, 8] \coloneqq \texttt{ret\_addr}\}. \tag{5.4}$$

Propagation requires performing *substitution*, which is defined over assignments, state parts,

and expressions, all with respect to invariant ϕ.

$$\begin{aligned}
\text{subst}(\phi, sp \coloneqq v) &= \text{subst}(\phi, sp) \coloneqq \text{subst}(\phi, v) && \text{(5.5a)}\\
\text{subst}(\phi, sp) &= \text{if } \exists v \cdot (sp, v) \in \phi \text{ then } v \text{ else } sp && \text{(5.5b)}\\
\text{subst}(\phi, e_0 \bigcirc e_1) &= \text{if } \exists v \cdot (e_0 \bigcirc e_1, v) \in \phi \text{ then } v \text{ else } \text{subst}(\phi, e_0) \bigcirc \text{subst}(\phi, e_1) && \text{(5.5c)}\\
\text{unary ops} &= \dots\\
\text{ternary ops} &= \dots
\end{aligned}$$

Algorithm 5.2 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:

$$\text{post}(\phi, \alpha)^1 \equiv \{(v, e) \mid (v \coloneqq e \in \alpha \wedge (v, \_) \in \phi) \vee ((v, e) \in \phi \wedge (v, e) \text{ is unmodified by } \alpha)\}. \tag{5.6}$$

Example 5.4 (Invariant propagation)**.** Once again consider Example 3.6. Propagation of the initial precondition through the single basic block produces the following postcondition:

$$\begin{aligned}
\phi = \{&\texttt{rip} \coloneqq \texttt{ret_addr},\\
&\texttt{rsp} \coloneqq \texttt{rsp}_0 + 8,\\
&[\texttt{rsp}_0 - 8, 8] \coloneqq \texttt{0xAABBCCDD} \bullet \langle 31, 16\rangle v_0 \bullet \texttt{0xEEFF},\\
&[\texttt{rsp}_0, 8] \coloneqq \texttt{ret_addr}\}.
\end{aligned} \tag{5.7}$$

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.

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

Algorithm 5.2 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))
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')
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`)
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`)
function PROP(ϕ, `Resume` *resumes*)
  for all $\alpha_i \in$ *resumes* **do**
    $(\phi_i', \alpha_i') \leftarrow$ PROP(ϕ, α_i)
  $\phi'' \leftarrow \bigcap \phi'$
  return (ϕ'', `Resume` ZIP(i, α'))
function PROP(ϕ, α) ▷ *Default case*
  return (ϕ, α)

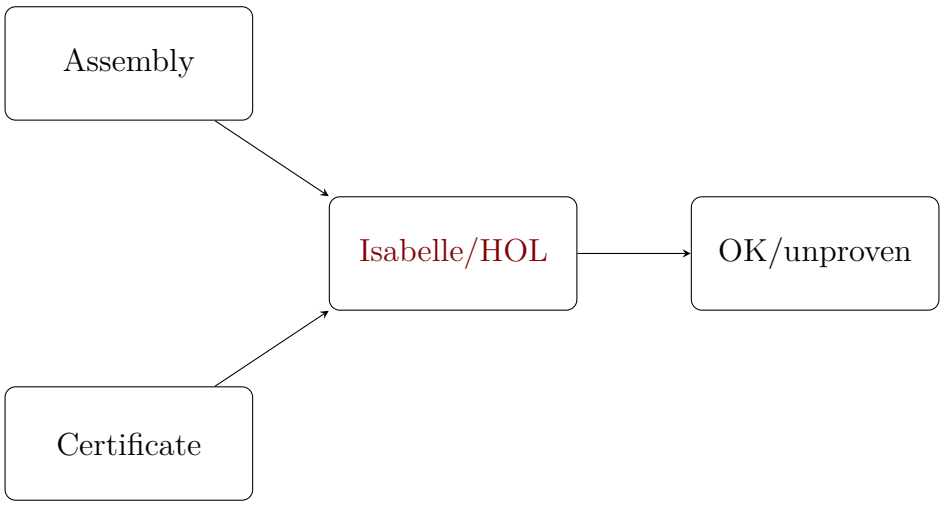


Figure 5.4: Overview of FMUC verification

5.2 FMUC Verification

This section presents verification of an FMUC as shown in Fig. 5.4, 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 usage 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 usage Hoare triple can be verified.

5.2.1 Syntactic Control Flow in Isabelle/HOL

As described previously, SCF 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 5.2.4 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 5.4 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 5.1.3.

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 5.2.2.

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.

5.2.2 Symbolic Execution for Verification

Section 4.2.1 previously presented a formal symbolic execution engine based on the machine model described in Section 3.1. 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

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

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 $\text{step} : A \times \mathbb{N} \times S \to S$, 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: $\text{execScf} : SCF \times S \times S \to \mathbb{B}$. 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 5.2.4.

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.

5.2.3 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 usage* with respect to state changes is required.

Definition 5.5 (Memory usage with respect to state changes)**.** The set of memory regions M' characterizes memory usage 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:

$$\text{preserve}(M', \sigma, \sigma') \equiv \forall a \cdot (\forall r \in M' \cdot [a, 1] \bowtie r) \longrightarrow \sigma : *[a, 1] = \sigma' : *[a, 1]. \tag{5.8}$$

Using Definition 5.5, each block gets a lemma of the form

$$P(\sigma) \longrightarrow \text{SYMBEXEC}(n, a, i, \sigma, \sigma') \wedge Q(\sigma') \wedge \text{preserve}(M(\sigma), \sigma, \sigma'). \tag{5.9}$$

Note that M is a state-dependent function. Every generated version of Eq. (5.9) is discharged with an Isabelle/HOL proof method written in Eisbach [120], 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 usage 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.

5.2.4 Function Body Verification

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

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 4, 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 Hoare-style approach.

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

(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

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

(d) While rule

$$\frac{M = \varnothing \qquad P \longrightarrow Q}{\{P\}\texttt{Skip}\{Q;M\}}$$

(e) Skip rule

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

(f) Resume rule

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

(g) Precondition weakening

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

(h) Postcondition strengthening

Figure 5.5: Hoare rules for memory usage

Definition 5.6 (Hoare triple for SCF)**.**

$$\{P\}f\{Q;M\} \equiv \forall \sigma\ \sigma' \cdot P(\sigma) \wedge \textsc{execScf}(f, \sigma, \sigma') \longrightarrow Q(\sigma') \wedge \text{preserve}(M, \sigma, \sigma') \quad (5.10)$$

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 5.6 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. 5.5a, 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.

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. 5.5 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 5.2.3.

The while rule is based on a loop invariant, I. If the memory usage of one iteration of loop body f is constrained to set of memory regions M, then so is the memory usage 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:

$$M' = \{r \mid \exists\sigma \cdot I(\sigma) \wedge r \in M(\sigma)\} \tag{5.11a}$$
$$= \{r \mid \exists\sigma \cdot i(\sigma) < 10 \wedge r = [a + i(\sigma), 1]\} \tag{5.11b}$$
$$= \{[a', 1] \mid a \leq a' \leq a + 10\} \tag{5.11c}$$

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 usage and of memory usage as a whole. A weaker invariant, such as $i < 20$, would result in a larger set of memory regions

Listing 5.1: 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 5.2: Main VCG method

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

by relaxing the constraints on symbolic addresses and, for other situations, symbolic region sizes.

Verification Condition Generation

The VCG presented here is a set of Eisbach proof methods, the entry point of which is shown in Listing 5.2. It is designed to automatically apply the proper Hoare logic rules as much as possible via the `vcg_step` method in Listing 5.1, driven by the formal SCF provided by the FMUC.

Internally, `vcg_step` repeatedly applies one of the Hoare rules from Fig. 5.5 (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 5.2.3. 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,

Listing 5.3: 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 5.4: 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   )?
```

shown in Listing 5.4 that relies on the loop rule presented in Fig. 5.5d. That loop rule is structured such that the majority of work required to support the loops is identifying the 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. 5.5g 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.

5.2.5 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:

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

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

Figure 5.6: Frame rule for composition of memory usage

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:

```
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 $*[\texttt{rsp}_0^g - 8, 8] = \texttt{rbp}_0^g$, 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: $\texttt{rbp} = \texttt{rbp}_0^g$. 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 $[\texttt{rsp}_0^g - 8, 8]$ is required. The proof must also show that f does not overwrite region $[\texttt{rsp}_0^g, 8]$, 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* [108, 135, 148] requires. As described in Section 1.3.1, 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 usage verification, we developed the frame rule presented in Fig. 5.6. 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 usage, 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 $[\texttt{rsp}_0^g - 8, 8] = \texttt{rbp}_0^g$. 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:

$$\{P\}\boxed{\mathrm{f}}\{Q;M_f\} \equiv \exists P_f\ Q_f\ P_{\text{sep}} \cdot \text{all four assumptions of the frame rule are satisfied.} \quad (5.12)$$

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.

5.3 Full Example

This section presents an execution of the entire toolchain on the example given in Fig. 5.7a as a summary of Sections 5.1 and 5.2. 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 5.7b 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. 5.7d 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 [48]. 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. 5.7d). 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 5.2.5, `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 5.7f 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 5.2.4. The first proof method applied is `vcg`, which initializes the proof with the

```c
int main(int argc, char** argv) {
  int* a = (int*)argv;
  int* b = (int*)(argv + 4);
  *(int*)(argv + 2) = *a + *b;
  *(char*)argv = 'a';

  int array[argc];
  for (int i = 0; i < argc; i++)
    array[i] = argv[i][0] * 2;

  if (is_even(argc))
    return array[argc];
  return array[0];
}
```

(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

$$M = \{r_0 = [\mathtt{rsp}_0, 8], r_1 = [\mathtt{fs}_0 + 40, 8], r_2 = [\mathtt{rsi}_0 + 36, 4], r_3 = [\mathtt{rsp}_0 - 8, 8], \dots\}$$
$$MRR = \{r_0, r_1, r_2, r_3, \dots, r_{12}\} \text{ are separate}$$

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

$$P_{\mathtt{123e}}(\sigma) = \begin{array}{r} \mathtt{rip} = \mathtt{0x123e} \\ \mathtt{rbp} = \mathtt{rsp}_0 - 8 \\ \mathtt{rdi} = \mathtt{rdi}_0 \\ \mathtt{rsp} = \mathtt{rsp}_0 - (88 + 16 * ((15 + 4 * \mathrm{sxtnd}(\langle 31, 0\rangle \mathtt{rdi}_0)/16)) \\ *[\mathtt{rsp}_0 - 40, 8] = \mathtt{rsp}_0 - (85 + 16 * ((15 + 4 * \mathrm{sxtnd}(\langle 31, 0\rangle \mathtt{rdi}_0))/16)) \gg 2 \ll 2 \\ *[\mathtt{rsp}_0 - 48, 8] = \mathrm{sxtnd}(\langle 31, 0\rangle \mathtt{rdi}_0) - 1 \\ *[\mathtt{rsp}_0 - 56, 8] = \mathtt{rsi}_0 + 32 \end{array}$$

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

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

(e) Assumption due to call of `is_even`

```
apply (vcg scf: main_scf)
apply (vcg_while ‹P123e  P1246›)
apply vcg_step+
```

(f) Isabelle proof code (manual effort)

Figure 5.7: Application of entire methodology on example

function's SCF, applies Hoare rules, and proves correctness of all memory usage up 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 $\texttt{rsi}_0$ on the assembly level) is not allowed to point to a region within the stack frame of the `main` function.

5.4 Application: Xen Project

The Xen Project [34] is a mature, widely-used virtual machine monitor (VMM), 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 (Amazon EC2), 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[2], 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[3]. 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 [176]. 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

[2] https://wiki.xen.org/wiki/XenStore
[3] https://linux.die.net/man/1/cpuid

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 5.1: Verified Xen functions

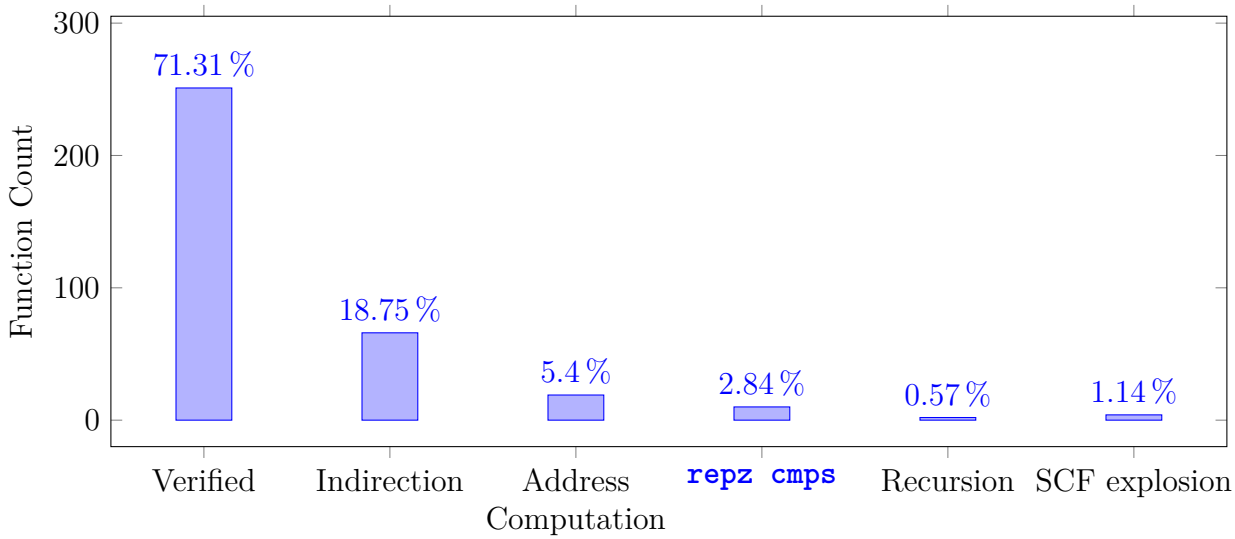


Figure 5.8: Analyzed Xen functions compared to unverified features

the functionality of Quick Emulator (QEMU). QEMU is a free, open-source emulator.[4] Xen uses it to emulate device models (DMs), which provide interfaces for hardware storage.

This methodology is currently capable of dealing with Successful Xen percent of the functions present in the aforementioned binaries (see Fig. 5.8). The supported features include (nested) loops, subcalls, variable argument lists, jumps into other function bodies, string instructions with the **`rep`** prefix, and single instruction, multiple data (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 5.1.1, 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

[4] https://www.qemu.org/

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.

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. 5.3 shows up extensively or when while loops have multiple entry points.

Table 5.1 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. 5.8.

5.5 Summary

This chapter presented an approach to formal verification of memory usage 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 Successful Xen percent 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.

Part III

Hoare Graphs

Chapter 6

Lattice-Based Formal Lifting and Hoare Graphs

The previous works in this dissertation provided ways of lifting binaries to some abstract representation using Floyd-style and Hoare-style methodologies. While the second method was an improvement on the first in terms of efficiency, automation, and coverage, there was plenty of room for improvement. Loops still required manual effort to deal with, memory regions that potentially aliased were not supported, and both recursion and indirect branches were scoped out. Furthermore, our SCFs were susceptible to explosion, resulting in significant overheads for certain functions. Additionally, while possible, function composition was difficult to accomplish without further manual effort. Finally, that work still relied on assembly dumps and assumed programs could not jump into instructions, which is not always the case.

To deal with those issues, we introduce the concept of *Hoare graphs (HGs)*. These HGs return to a CFG-style analysis, but with significant improvements. First, the analysis itself is fully automated, much like the Hoare-style approach. Furthermore, HGs provide structured, nondeterministic memory models to deal with the memory aliasing problem. Loops and similar control flow structures requiring fixed points utilize an improved stabilization process involving a *join-semilattice*, which also assists in reducing state space explosion. Support for some indirect calls and jumps was added in the form of overapproximative jump table calculations. Context-free function composition is included as well, reducing time and space consumption while allowing larger programs to be analyzed. The specific composition mechanism used also provides support for recursive functions. Lastly, the tool we have provided operates directly on binaries, reading instructions at specific addresses rather than operating on a parsed list of assembly. This allows for the detection of "weird" edges [58, 161] that previous analyses could miss. Identifying such edges is important for the detection of unexpected and unintended program behavior.

In short, this part of the dissertation contains the following contributions:

- a trustworthy, automated approach to binary lifting via HGs, which provide a formal overapproximative relation between the binary and the lifted output;

- a demonstration that overapproximative binary lifting can be used to find "weird" edges in binaries; and
- the application of that binary lifting approach to all non-concurrent x86-64 executables of the Xen hypervisor. In total, we successfully lifted 45 binaries and 2115 library functions to HGs. 399 771 assembly instructions were reached by the analyses.

We have made the associated code artifact for this contribution publicly available [21].

Below is an example of an HG and properties of interest within it. This chapter is followed by an in-depth exploration of the technical formulation of HGs and their generation (Chapter 7). A practical application follows in Chapter 8. Finally, we wrap up in Chapter 9.

6.1 Example

Listing 6.1 and Fig. 6.1 show a snippet of a binary in assembly form and a sample of its lifted HG, respectively. For the sake of presentation, the example uses 32-bit instructions. Additionally, the symbolic value `a` is used to represent the base address of some jump table. The operations of that assembly snippet are as follows.

1. The **`cmp`** and **`ja`** instructions on Lines 1 and 2 compare the current value of register **`eax`** to the constant value `0xc3`. If **`eax`** is less than or equal to `0xc3`, the **`mov`** at address `0xb` (Line 3) reads from a jump table with base address `a` and the value stored in register **`eax`** as the jump table index. The pointer read from the jump table (referred to as $a_{\texttt{jt}}$) is stored in register **`eax`**.
2. Two memory writes happen:
   (a) Pointer $a_{\texttt{jt}}$ is written to memory at the address stored in register **`edi`** (Line 4).
   (b) The immediate value 1 is written to memory at the address stored in register **`esi`** (Line 4).
3. On Line 6, pointer $a_{\texttt{jt}}$ is used as the target of an indirect branch.

In short, the expected behavior of this assembly is that it reads an address from a jump table containing `0xc3` addresses and jumps to that address.

However, the example is constructed as an example of "weird" control flow [58, 161]. At first glance, there are no **`ret`** instructions in Listing 6.1. Despite this, under specific circumstance, a **`ret`** instruction may be executed. That circumstance is if the pointers in registers **`esi`** and **`edi`** alias. If that is the case, one of the bytes of the first instruction (`0xc3`) is interpreted as *another* instruction, specifically **`ret`**. As this is a real concrete execution path, any overapproximative lifted representation must model such behavior.

We explain several of the points made in the introduction using this example.

Remark 6.1 (Notation). The notation at state `14` indicates that reading 4 B from address **`edi`** produces value $a_{\texttt{jt}}$. Additionally, $\equiv$ and $\bowtie$ denote aliasing and separation, respectively.

Listing 6.1: Example binary snippet for HG lifting

```
1 0x0 : 3dc3000000     cmp  eax,c3
2 0x5 : 0f8718000000   ja   1c
3 0xb : 8b0485__a___   mov  eax,DWORD PTR [eax*4+a]
4 0x12: 8907           mov  DWORD PTR [edi],eax
5 0x14: c70601000000   mov  DWORD PTR [esi],1
6 0x1a: ff27           jmp  DWORD PTR [edi]
```

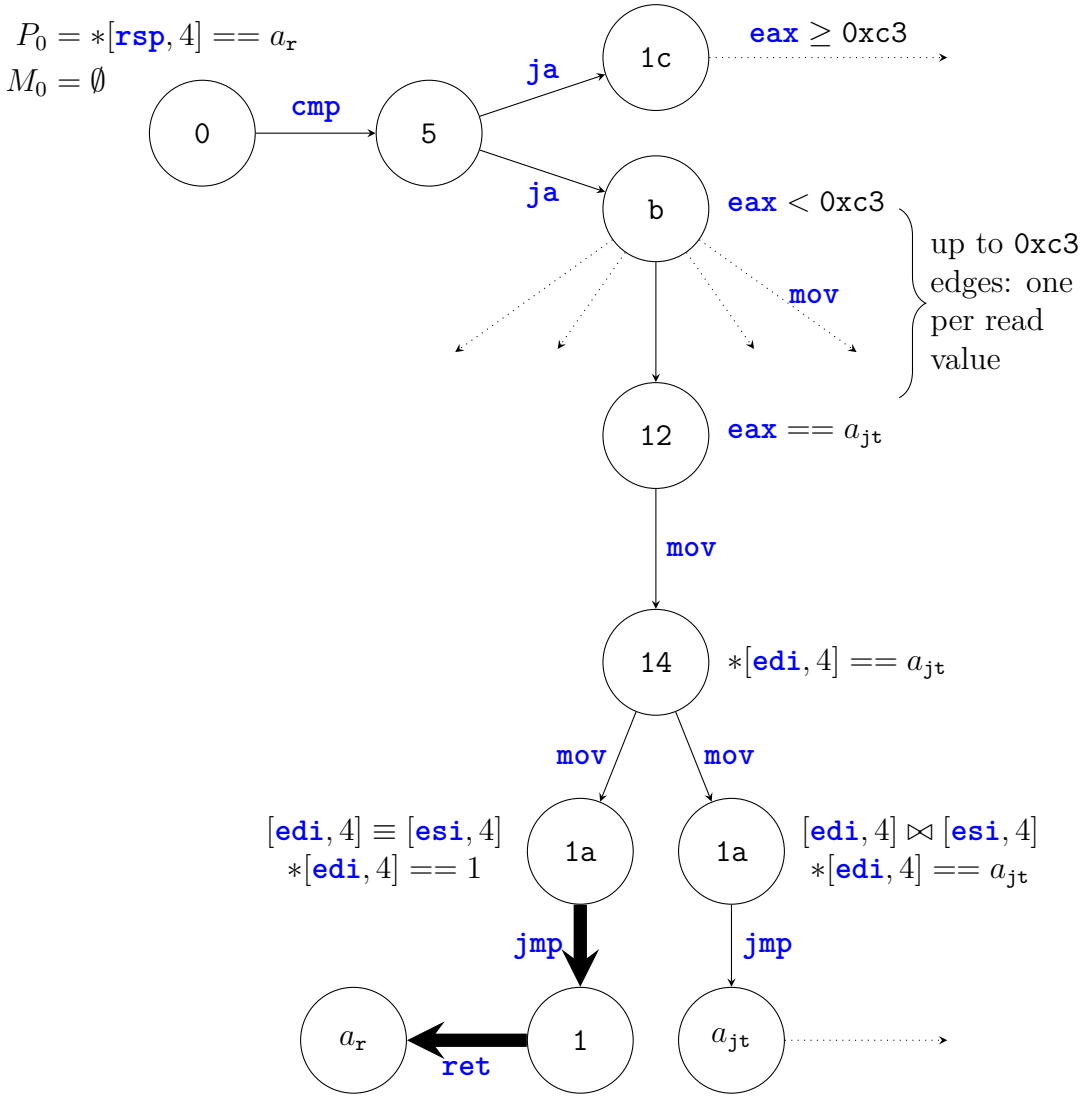


Figure 6.1: HG example

6.1.1 The Hoare Graph is Provably Overapproximative

Consider the set of outgoing edges from vertex `b`. The predicate associated with that vertex contains clauses indicating that register **`eax`** is bounded. That information is sufficient for proving that reading the jump table provides at most `0xc3` possible values for $a_{\texttt{jt}}$. In order to derive that information, the predicate for vertex `5` must contain information on the flags read by the **`ja`** instruction. Those flags are set by **`cmp`**. Looked at another way, the edges in the path `0` $\rightarrow$ `5` $\rightarrow$ `b` each form a Hoare triple with the predicates at each vertex as their pre- and postconditions.

6.1.2 Disassembly Requires Alias Analysis

The predicate for vertex `14` does not contain any information regarding the aliasing relationship between the pointers in registers **`edi`** and **`esi`**. Thus we must overapproximate nondeterministically by having one outgoing edge for each case. In the aliasing ($\equiv$) case, the **`mov`** on Line 5 overwrites the previous **`mov`** on Line 4. The program would then jump to address `1` instead of performing the intended jump to address $a_{\texttt{jt}}$.

6.1.3 Disassembly Requires Bounds Checking

Another symbolic value, $a_{\texttt{r}}$, represents the address initially stored at the top of the stack frame. The HG contains an edge to a final vertex[1] where the instruction pointer is set to that address. To obtain that result, every other vertex on the path from vertex `0` to that final vertex must contain enough information preserve the stack. Specifically, they must show that the return address has not been modified and that the frame and stack pointers (**`rbp`** and **`rsp`**) are managed properly throughout function execution.

6.1.4 Weird Edges are Found

A jump to address `1` jumps into the middle of an instruction. Since byte `c3` corresponds to the `ret` instruction, this is actually ROP gadget. An unexpected "weird" edge leading to unexpected control flow has been found. In Fig. 6.1, they are denoted with bold arrows.

Remark 6.2 (Memory regions are assumed to not partially overlap). The branch at vertex `14` produces two edges, one for aliasing and one for separation. Hypothetically, the two 4-byte regions [**`edi`**, 4] and [**`esi`**, 4] could also partially overlap. For example, there may be a case where **`edi`** = **`esi`** + 2. We exclude this case, because it rarely occurs in binaries compiled from source code, even at high optimization levels. We do support enclosure of regions within larger ones, however.

6.1.5 Hoare Graphs Facilitate Formal Verification

The HG is generated by the algorithm presented in Section 7.4. While that algorithm has been proven sound with pencil-and-paper proofs for Theorem 7.27, stronger guarantees can be

[1] state

provided by formal verification. To that end, HGs can be exported to the interactive theorem prover Isabelle/HOL. In that formulation, akin to our Floyd-style work from Chapter 4 or Hoare-style work from Chapter 5, each vertex becomes its own lemma.

Example 6.3 (Vertices as lemmas)**.** Vertex `14` is translated to a Hoare triple that states that the invariant associated with instruction address `14` ensures, as a postcondition, the disjunction of the invariants associated with address `1a`.

Essentially, this step removes the need to trust the implementation of the algorithm presented in this paper.

At first glance, it may seem that a small piece of code leads to an exorbitant number of states and edges. However, typically the state space is close to the number of instruction addresses (see Section 8.1.2), as we apply joining of states to reduce the state space whenever possible.

Chapter 7

Technical Formulation

This chapter contains the details of HG lifting and the necessary *symbolic states*. First with some introductory concepts for our symbolic states in Section 7.1, then detailed coverage of our state predicates and memory models in Sections 7.2 and 7.3, and finally the lifting algorithm itself in Section 7.4. These concepts are accompanied by explanations and proofs.

7.1 Starting Concepts

For this part, we have the following types:

`Inst` a structured representation of assembly instructions

Pred state predicates (Section 7.2)

Mem memory models (Section 7.3)

$\mathbb{W}_n$ words of bit length n, with a default of 64 bit

The vertices of an HG are represented by the aforementioned symbolic states, which are tuples of type

$$\mathsf{Pred} \times \mathsf{Mem}.$$

We use σ and s to respectively denote symbolic and concrete states. Additionally, notation $s \vdash X$ indicates that X holds in concrete state s, where X can be either a predicate or a memory model.

Definition 7.1 (Binaries)**.** A binary is defined by a tuple of the form

$$\langle a_e, \mathrm{fetch}, \mathbb{S}, \rightarrow_B \rangle,$$

where a_e of type $\mathbb{W}_{64}$ is the entry point of the binary and fetch of type $\mathbb{W}_{64} \mapsto$ `Inst` returns the instruction at the supplied address. The behavior of the binary is modeled by some set of concrete states s and some deterministic black-box transition relation $\rightarrow_B$ over concrete states.

Definition 7.2 (Hoare graphs)**.** An HG is defined as a tuple

$$\langle \mathbb{\Sigma}, \sigma_I, \rightarrow_\Sigma \rangle,$$

where symbolic states in $\mathbb{\Sigma} = \mathsf{Pred} \times \mathsf{Mem}$ consist of predicates and memory models, $\sigma_I \in \mathbb{\Sigma}$ is an initial symbolic state, and $\rightarrow_\Sigma$ of type $\mathbb{\Sigma} \times \texttt{Inst} \times \mathbb{\Sigma} \mapsto \mathbb{B}$ is a non-deterministic transition relation labeled with instructions.

Furthermore, the algorithm requires the definition of a *join* operation over symbolic states. This operation soundly merges the information stored in two such states. It is required because the algorithm explores the HG state space by recursively adding new vertices/states and edges on the fly. Joining serves two purposes:

1. to prevent state space explosion and
2. to ensure termination.

If an instruction address is visited more than once, the *supremum*[1] of all symbolic states associated with that address is computed until an LFP is reached.

We therefore define an algebraic *join-semilattice* over symbolic states. That is, we define our join operation such that it establishes a partial order over our symbolic states, allowing us to calculate a least upper bound for any two symbolic states. This join-semilattice is depicted as the tuple $\langle \mathbb{\Sigma}, \sqcup \rangle$, where $\mathbb{\Sigma}$ is the type of symbolic states and $\sqcup$ denotes the join operation. The desired partial ordering over $\mathbb{\Sigma}$, $\sqsubseteq$, is then derived by defining $\sigma_0 \sqsubseteq \sigma_1$ as $\sigma_1 = \sigma_0 \sqcup \sigma_1$. Intuitively, $\sigma_0 \sqsubseteq \sigma_1$ denotes that σ_0 is "less abstract" than σ_1.

In a typical setting, the join operation would be implemented by a disjunction. However, using plain disjunction as a join does not provide sufficient abstraction for our purposes; it is too precise. As an example, using disjunction with two predicates where one has $x = 1$ and the other has $x = 10$ would produce a predicate equivalent to $x \in \{1, 10\}$. We therefore provide a join operation that abstracts by *overapproximating* the disjunction; in other words, making it coarser. To reuse the previous example, the result would instead be $1 \leq x \leq 10$. The join must then satisfy the following soundness criterion for any concrete state s:

$$(s \vdash P \vee Q) \implies (s \vdash P \sqcup Q). \tag{7.1}$$

Furthermore, the partial ordering derived from this join is *coarser* than ordering based on implication. That is, $P \sqsubseteq Q \implies (s \vdash P \implies Q)$. As $\sqsubseteq$ is used as a termination condition during LFP computation, its coarseness is important both to ensure termination of the algorithm and to reduce running time. In other words, the join must be of sufficient coarseness to ensure that there are no infinitely descending chains of symbolic states $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \cdots$. However, too much coarseness is undesirable. An overly coarse join, such as one that just produces an always-true predicate, would result in an HG that is too abstract and over which no properties could be proven.

Finally, as a symbolic state consists of a predicate and a memory model, we define a subjoin for both.

[1] equivalent of summation for joining

7.2 Predicates

Predicates are assertions on state. A predicate P consists of a set of *clauses*. P holds in state s ($s \vdash P$) if and only if all clauses hold.

Breaking it down further, a clause consists of two symbolic expressions and their relation. A symbolic expression of type $\mathbb{E}$ consists of the some combination of the following:

- register references ($\mathbb{R}$),
- flag references ($\mathbb{F}$),
- 64-bit words ($\mathbb{W}$),
- symbolic values ($\mathbb{V}$),
- memory regions (modeled by an expression for the address and a natural number for the size), and
- the application of an operator to a list of expressions.

In formal notation, this is:

$$\mathbb{E} \coloneqq \mathbb{R} \mid \mathbb{F} \mid \mathbb{W} \mid \mathbb{V} \mid \mathbb{E} \times \mathbb{N} \mid \mathsf{Op} \times [\mathbb{E}] \tag{7.2}$$

We identify a subset of these symbolic expressions called *constant expressions* ($\mathbb{C}$). These expressions cannot contain state parts such as registers, flags, or memory regions. They represent constants or computations constructed using initial values. For example, $\texttt{rdi}_0$ denotes the initial symbolic valueof register `rdi`. This value does not change during symbolic execution.

In notation, clauses take the form $\mathbb{E} \square \mathbb{C}$,where $\square \in \{=, \neq, <, <_s, \geq, \geq_s\}$. The $\square_s$ relations treat their operands as signed, while the corresponding non-subscripted versions treat their operands as unsigned.

There are also two special clauseless predicates, $\top$ and $\bot$. Those special predicates respectively indicate always true (holds for any state) and always false (holds for no state). $\bot$ is also used to indicate an unknown $\mathbb{C}$.

Definition 7.3 (Predicate join)**.** The aforementioned join of two predicates P and Q, notation $P \sqcup Q$, is performed by doing a form of range abstraction for symbolic bit-vector values [152]. This is implemented as:

$$\begin{aligned}
P \sqcup Q &\stackrel{\text{def}}{=} \bigcup \{ \between(p, q) \mid \langle p, q \rangle \in P \times Q \} \\
\between(l = r_1, l = r_2) &\stackrel{\text{def}}{=} \{ l \geq \min(r_1, r_2), l \leq \max(r_1, r_2) \} \\
\between(l < r_1, l < r_2) &\stackrel{\text{def}}{=} \{ l < \max(r_1, r_2) \} \\
&\vdots \\
\between(a, b) &\stackrel{\text{def}}{=} \begin{cases} \{a\} & \text{if } a = b \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}$$

The operator presented in Definition 7.3 performs a *merge* for each clause pair $\langle p, q \rangle$ in the Cartesian product of its argument predicates. This merge produces a potentially empty set of clauses generated from the two clauses supplied to it. While only merge rules for $=$ and $<$ are shown, there are also rules for the other possible clause operations. The max and min functions used are partial; they do not have a result if the maximum/minimum of the expressions supplied to them cannot be determined. This can happen if those symbolic expressions contain unrestricted values. In such cases, no clause is produced.

Example 7.4 (Predicate join)**.** Let $P = \{a = 3, a < \texttt{rdi}_0\}$ and $Q = \{a = 4, a < \texttt{rsi}_0\}$. As both predicates have equality clauses for a, those clauses are merged to produce a pair of clauses denoting that the value of a lies in the range $[3, 4]$. Since no maximum can be established between $\texttt{rdi}_0$ and $\texttt{rsi}_0$, these clauses are dropped. Thus, $P \sqcup Q = \{a \geq 3, a \leq 4\}$.

As required for a lattice, the join is associative, commutative, and idempotent. Associativity is derived from the fact that set union and minimum/maximum are associative operations. The join is commutative and idempotent due to the commutativity and idempotency of the merge function. Finally, we have the following for any state s:

Lemma 7.5 (Soundness of the join)**.** $s \vdash P \vee Q \implies s \vdash P \sqcup Q$.

Proof. The proof of this lemma consists of two cases:

$$s \vdash P \implies s \vdash P \sqcup Q$$
$$s \vdash Q \implies s \vdash P \sqcup Q$$

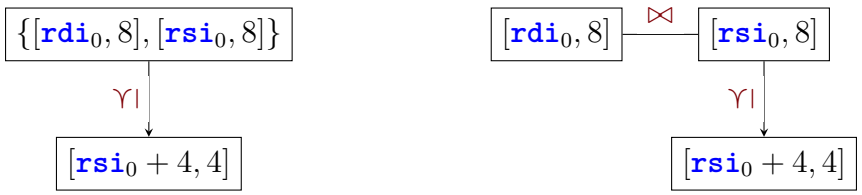
Only one of those cases needs to be proven; the other follows from the symmetry of the join operation. Assuming $s \vdash P$, we now show $s \vdash P \sqcup Q$. For $s \vdash P \sqcup Q$ to be true, all of its clauses must hold. Consider the first case, for equality.0 Since $c \stackrel{\text{def}}{=} l = r_1$ is a clause in P and $l = r_2$ is a clause in Q, we have $c'_1 \stackrel{\text{def}}{=} l \geq \min(r_1, r_2)$ and $c'_2 \stackrel{\text{def}}{=} l \leq \max(r_1, r_2)$. c implies both of those, and thus $s \vdash c \implies s \vdash c'_1 \wedge c'_2$. The other cases are similar. □

7.3 Memory Models

Program analysis in programs with pointers requires efficient alias identification and classification. When different variables (or in our case, state parts) point to the same memory region, those variables are *aliased*. This is an important issue that must be dealt with for proper predicate transformation. That is because alias information directs assignments to memory. Specifically, when two state parts are aliased,[2] writing to the symbolic location (memory region) specified by one of them will change the value that is read from the symbolic location (memory region) specified by the other. Unfortunately, it is not always possible to determine whether or not two state parts alias.

We thus keep track of memory regions read and written during a sequence of execution in structured *memory models*. These memory models store *aliasing*, *separation*, and *enclosure*

[2]Or rather, the symbolic expressions recorded as being stored in them alias.



(a) Aliasing (b) No aliasing

Figure 7.1: Memory model examples

relations for memory regions. This allows for efficient checks of those relations. Furthermore, constructing them in a nondeterministic fashion allows for dealing with multiple potential combinations of relations between read and written regions in memory. They are defined by the following data structures:

$$\mathsf{MemTree} \coloneqq \{\mathbb{C} \times \mathbb{N}\} \times \mathsf{Mem} \qquad\qquad \mathsf{Mem} \coloneqq \{\mathsf{MemTree}\}$$

That is, a memory model consists of a possibly empty *forest* of memory trees. Each memory tree has as a top-level node, a set of memory regions, and a possibly empty sub-forest that holds its child regions. Two regions in the same node set alias. The child regions are enclosed in their parents. Finally, regions in sibling nodes are separate.

Example 7.6 (Aliasing and non-aliasing)**.** Consider the two memory models presented in Fig. 7.1. These memory models involve three regions: $[\texttt{rdi}_0, 8]$, $[\texttt{rsi}_0, 8]$ and $[\texttt{rsi}_0 + 4, 4]$. The memory models depict the case where $\texttt{rdi}_0$ and $\texttt{rsi}_0$ alias and not alias. As stated above, sibling nodes on the same level are separate, while children are enclosed by their parents. Regions within the same node alias. Thus, Fig. 7.1a shows a situation with two top-level regions aliasing and the child region they share. Figure 7.1b, meanwhile, shows a situation where the two top-level regions do not alias, and thus only one of those regions contains an enclosed child.

Definition 7.7 (Memory relations)**.** Let s be a concrete state and let $r_0 = [e_0, n_0]$ and $r_1 = [e_1, n_1]$ be two regions in memory. Then, the properties of *aliasing*, *separation*, and *enclosure*, notations $\equiv$, $\bowtie$, and $\preceq$, respectively, are defined as:

$$r_0 \equiv r_1 \overset{\text{def}}{=} s \vdash e_0 = e_1 \wedge n_0 = n_1$$
$$r_0 \bowtie r_1 \overset{\text{def}}{=} s \vdash (e_0 + n_0 \leq e_1) \vee (e_1 + n_1 \leq e_0)$$
$$r_0 \preceq r_1 \overset{\text{def}}{=} s \vdash e_0 \geq e_1 \wedge e_0 + n_0 \leq e_1 + n_1$$

Such relations hold *necessarily* if and only if they hold in all concrete states s. For example, $[\texttt{rsi}_0, 4] \bowtie [\texttt{rsi}_0 + 4, 4]$ is a necessarily-separate relation as there are no values of $\texttt{rsi}_0$ for which those two regions are not separate. The SMT solver/theorem prover Z3 [55] is used to establish whether these "necessarily"-relations hold for symbolic addresses given the current state predicate. This is done via expression translation directly to Z3's bit-vector

representations, meaning no information is lost in the conversion and when querying the constructed logical formulas.

In our Z3 interface, each operation to determine aliasing/separation/enclosure also takes a predicate as an additional argument. This is used to provide additional assumptions for necessarily-calculations, as in isolation it is often not possible to determine the relationships between entirely symbolic expressions that do not share symbolic values. While the predicate may not help in all cases, there may be instances where a relation between two symbolic values or expressions was expressed previously in the program.

We further extend the notation in Definition 7.7 to memory trees. That is, $t_0 \bowtie t_1$ denotes that all regions in t_0 are necessarily separate from all regions in t_1. Notation $t_0 \equiv t_1$ ($t_0 \preceq t_1$) denotes that some region in the top node of t_0 and some region in the top node of t_1 necessarily alias (are enclosed).

7.3.1 Insertion

Construction of a memory model is performed using the recursive ins function shown below. It takes as input a memory tree t and the current memory model M. The current predicate P is also supplied to assist in the region relationship analysis. However, it is elided from the below presentation as it is a read-only value that is passed along through the function call chain. For output, function ins produces, nondeterministically, a set of new memory models based on all possible pointer relationships for the newly-inserted region. If no necessarily-relation can be established between t and any tree in M, then all trees possibly overlapping with t are destroyed (see Section 1.7.2). If a necessarily-relation can be established between tree t and some tree already in M, then only the relevant memory models need to be produced.

To ensure that the invariants for the memory model M being inserted into are not violated, the external interface to the function takes a single region, r, instead of a memory tree. This region is embedded in a minimal memory tree, $\langle \{r\}, \varnothing \rangle$, which is then supplied to the below function.

Definition 7.8 (Memory tree insertion)**.** Let $t_0 = \langle R_0, M_0 \rangle$ and $t_1 = \langle R_1, M_1 \rangle$ be two trees. Function ins of type $\mathsf{MemTree} \times \mathsf{Mem} \times \mathsf{Pred} \to \{\mathsf{Mem}\}$ is defined as follows:

$$(t_0, \varnothing) \stackrel{\text{def}}{=} \{t_0\}$$

$$\mathrm{ins}(t_0, t_1 : M) \stackrel{\text{def}}{=} \begin{cases} \mathrm{ins}_{\mathrm{AL}}(t_0, t_1, M) & \text{if } t_0 \equiv t_1 \\ \mathrm{ins}_{\mathrm{SEP}}(t_0, t_1, M) & \text{if } t_0 \bowtie t_1 \\ \mathrm{ins}_{\mathrm{ENC}}(t_0, t_1, M) & \text{if } t_0 \preceq t_1 \\ \mathrm{ins}_{\mathrm{CON}}(t_0, t_1, M) & \text{if } t_1 \preceq t_0 \\ \mathrm{destroy}(t_0, M) & \text{otherwise} \end{cases}$$

Notation $a : X$ denotes $\{a\} \cup X$

$$\begin{aligned}
\text{ins}_{\text{AL}}(t_0, t_1, M) &\stackrel{\text{def}}{=} \{(R_0 \cup R_1, M') : M \mid M' \in \mathsf{fold}(\text{ins}, M_0 \cup M_1)\} \\
\text{ins}_{\text{SEP}}(t_0, t_1, M) &\stackrel{\text{def}}{=} \{t_1 : M' \mid M' \in \text{ins}(t_0, M)\} \\
\text{ins}_{\text{ENC}}(t_0, t_1, M) &\stackrel{\text{def}}{=} \{\langle R_1, M' \rangle : M \mid M' \in \text{ins}(t_0, M_1)\} \\
\text{ins}_{\text{CON}}(t_0, t_1, M) &\stackrel{\text{def}}{=} \{\text{ins}(t', M) \mid t' \in \{\langle R_0, M' \rangle \mid M' \in \text{ins}(t_1, M_0)\}\}
\end{aligned}$$

As further explanation, let $t_0 = \langle R_0, M_0 \rangle$ be the tree to be inserted and $t_1 = \langle R_1, M_1 \rangle$ be a tree already in the memory model.

1. If t_0 and t_1 alias, then they are combined by taking the union of their nodes. Their child regions are not necessarily the same, however. Thus, subtrees are then reinserted using a fold.
2. If trees t_0 and t_1 are separate, then tree t_0 is recursively inserted into the remainder of the memory model and t_1 is added without modification.
3. If t_0 is enclosed in t_1, it is recursively inserted into the sub-forest of t_1. Each memory model M' thus obtained is wrapped in a tree $\langle R_1, M' \rangle$. The remainder of memory model M is unmodified.
4. Finally, if t_1 is enclosed in t_0, then t_1 is recursively inserted into the sub-forest of t_0. Each memory model M' thus obtained is wrapped in a tree $t' = \langle R_0, M' \rangle$. That tree is then recursively inserted into memory model M.

Example 7.9 (Potential aliasing in assembly)**.** Consider the three-instruction assembly snippet below. This snippet first stores the value 1000 in the eight-byte memory region pointed to by `rsi`, then stores the value 1001 in the four-byte region pointed to by `rsi` + 4. Finally, it stores the value 1002 in the eight-byte region pointed to be `rsi`. If the current state allows aliasing and separation between [`rdi`, 8] and [`rsi`, 8], then these three instructions will result in the two memory models in Fig. 7.1. Note that region [`rsi` + 4, 4] is necessarily enclosed in region [`rsi`, 8].

```
1 mov    qword ptr [rdi],   1000
2 mov    dword ptr [rsi+4], 1001
3 mov    qword ptr [rsi],   1002
```

A memory model M *holds* in concrete state s if all siblings are separate and all trees hold. A tree holds if its node contains only aliasing regions and all trees in its sub-forest are enclosed. Formally, this is expressed as:

Definition 7.10 (Memory model holding)**.** A memory model M *holds* in state s, notation $s \vdash M$, if and only if:

$$(\forall t_0, t_1 \in M \cdot t_0 \neq t_1 \implies s \vdash t_0 \bowtie t_1) \wedge (\forall t \in M \cdot s \vdash t)$$

A memory tree $t = \langle R, M \rangle$ *holds* in state s, notation $s \vdash t$, if and only if:

$$(\forall r_0, r_1 \in R \cdot s \vdash r_0 \equiv r_1) \wedge (\forall t' \in M \cdot s \vdash t' \preceq t) \wedge (s \vdash M)$$

Example 7.11 (Memory model holding)**.** Consider again the memory models in Fig. 7.1 for the assembly snippet in Example 7.9. The aliasing memory model in Fig. 7.1a holds only in states where $\texttt{rdi}_0 = \texttt{rsi}_0$. Meanwhile, the non-aliasing memory model in Fig. 7.1b holds only in states where $\texttt{rdi}_0 + 8 \leq \texttt{rsi}_0$ or $\texttt{rsi}_0 + 8 \leq \texttt{rdi}_0$.

An important quality our insertion function must possess is *completeness*. That is, the produced memory models should cover any possible relation between inserted region r and any region r' already present in the memory model. To formulate completeness, we use $R(M)$ to denote the set of regions in memory model M and $\mathcal{R}(M)$ to denote the set of relations. For example, we have $([\texttt{rdi}_0, 8] \equiv [\texttt{rsi}_0, 8]) \in \mathcal{R}(M)$ for the memory model in Fig. 7.1a.

Lemma 7.12 (Region insertion and memory model relations)**.** *Let r_0 and M be a region and a memory model, respectively. Also let f of type $\mathbb{C} \times \mathbb{N} \mapsto \{ \equiv , \bowtie , \preceq , \succeq \}$ be some mapping that provides a relation between r_0 and r' for any region r' currently in memory model M. Assume that f is possibly true:*

$$\exists s \cdot s \vdash M \wedge (\forall r' \in R(M) \cdot s \models r_0 \ f(r') \ r')$$

Then, insertion of region r_0 into M will produce at least a corresponding memory model:

$$\exists M' \in \text{ins}(\langle r_0, \varnothing \rangle, M) \cdot \{(r_0 \ f(r') \ r') \mid r' \in R(M)\} \subseteq \mathcal{R}(M')$$

In words, there exists some memory model that contains all relations of mapping f.

Proof. The proof is by induction. The base case is trivial. For the inductive case, we insert region r into $\{t_1\} \cup M$. Five cases are possible:

1. Region r_0 necessarily aliases with t_1. In this case, since mapping f is possibly true, it must assign $\equiv$ to all top-level regions of t_1, $\succeq$ to all other regions in t_1 and $\bowtie$ to all regions in M. The created memory model contains these relations.
2. Region r_0 is necessarily separate from t_1. In this case, since mapping f is possibly true, it must assign $\bowtie$ to any region in t_1. Thus, tree t_1 is not modified and region r_0 is recursively inserted into M. The induction hypothesis (IH) then finishes the proof.
3. Region r_0 is necessarily enclosed by a top-level region of t_1. Since mapping f is possibly true, it must assign $\bowtie$ to all regions of M. Therefore, the insertion function does not modify M. Since r_0 and r_1 do not alias, the top-level regions R_1 of tree t_1 can remain unmodified as well. Region r_0 is recursively inserted into a child of t_1, proof follows from the IH.
4. Tree r_1 is necessarily enclosed into region r_0. In this case, since mapping f is possibly true, it must assign $\succeq$ to all regions of t_1. Therefore, tree t_1 is recursively inserted as subtree of r_0, producing a set of trees. For the remaining regions not in t_1, f can hold arbitrary relations. Therefore any t' in the produced set is recursively inserted into M. Again, the IH then finishes the proof.

$\boxed{\{[\texttt{rdi}_0, 8], [\texttt{rdx}_0, 8]\}}$ (a) Alternate memory model

$\boxed{[\texttt{rdi}_0, 8]}$ (b) Join result

Figure 7.2: Join with alternate memory model

5. As an extra case, consider when no relation can necessarily be established. In that case, all of the above proofs apply, as memory models are generated for each possible case within this set.[3] □

7.3.2 Joining

As with state predicates, memory models have a join operation.

Definition 7.13 (Memory model join)**.** The join of two memory models M_0 and M_1, notation $M_0 \sqcup M_1$, is recursively defined as:

$$M_0 \sqcup M_1 \stackrel{\text{def}}{=} \left\{ \sqcup (T) \;\middle|\; T \in {}^{M_0 \cup M_1}\!/_{\mathbf{C}^+} \right\}$$

$$\langle R_0, . \rangle \; \mathbf{C} \; \langle R_1, . \rangle \stackrel{\text{def}}{=} R_0 \cap R_1 \neq \varnothing$$

$$\sqcup (T) \stackrel{\text{def}}{=} \left\langle \bigcap \{R \mid \langle R, . \rangle \in T\}, \bigsqcup \{M \mid \langle ., M \rangle \in T\} \right\rangle$$

This operation partitions the memory trees in M_0 and M_1 based on equivalence relation $\mathbf{C}^+$. This equivalence relation is the transitive closure of relation $\mathbf{C}$, which determines if its two memory trees have any top-level regions in common. In other words, all memory trees that have one or more top-level regions in common are put in an equivalence class and are thus joinable. The operator $\sqcup$ then performs the join operation for each equivalence class of memory trees, taking the intersection of all their region sets and the supremum of their child memory models.

Example 7.14 (Memory model join)**.** Consider two memory models M_0 and M_1 where both have the top node $[\texttt{rdi}_0, 8]$. We add the distinction that M_0 has an enclosed child $[\texttt{rdi}_0, 4]$ and M_1 has an enclosed child $[\texttt{rdi}_0 + 4, 4]$. The result of joining M_0 and M_1 is a single memory model with $[\texttt{rdi}_0, 8]$ as its top node and the two child regions in separate sibling nodes.

Example 7.15 (Memory model join cases)**.** First, consider the memory models shown in Fig. 7.1. Since the top nodes share a region, the two trees belong to the same equivalence class. The intersection of all of the region alias sets is:

$$\{[\texttt{rdi}_0, 8], [\texttt{rsi}_0, 8]\} \cap \{[\texttt{rdi}_0, 8]\} \cap \{[\texttt{rsi}_0, 8]\}.$$

As this intersection produces $\varnothing$, the result of the join is $\varnothing$, an empty memory model. Second, consider the join of the memory model shown in Fig. 7.1a with the one shown in Fig. 7.2a.

[3] This does *not* include partially overlapping cases, as described in the previous chapter.

The result of that join is the memory model shown in Fig. 7.2b. This is because all top-level memory trees involved in the merge share a region, $[\texttt{rdi}_0, 8]$, but have no comparable children.

To strengthen our arguments, we prove that the join over memory models we have provided is sound.

Lemma 7.16 (Soundness of the memory model join)**.** *Let s be a state and M_0 and M_1 be memory models. Then:*

$$(s \models M_0 \vee M_1) \implies (s \models M_0 \sqcup M_1)$$

Proof. Let $r_0 \square r_1$ be a relation in $\mathcal{R}(M_0 \sqcup M_1)$. If $\square$ is $\equiv$, then both regions r_0 and r_1 must have been present in all trees in the corresponding equivalence class. The relation thus held in either M_0 or M_1. If $\square$ is $\bowtie$, then the two regions are from trees generated from different equivalence classes. Since they are from trees that do not share a top-level region, the original trees in either M_0 or M_1 are separate as well. Similar reasoning applies for the other cases. □

Definition 7.17 (Symbolic state join)**.** The join of two symbolic states $\sigma_0 = \langle P_0, M_0 \rangle$ and $\sigma_1 = \langle P_1, M_1 \rangle$, notation $\sigma_0 \sqcup \sigma_1$, is:

$$\sigma_0 \sqcup \sigma_1 \stackrel{\text{def}}{=} \langle P_0 \sqcup P_1, M_0 \sqcup M_1 \rangle$$

Remark 7.18 (Loss of information). This join in its full form *loses* information. It can thus only be applied in a sound fashion for *postcondition weakening* [87]. In other words, dropping clauses and performing state cleanup serve only to reduce state constraints; they never add additional ones. In practice, this loss of information means that we may produce a state that would not actually be encountered during program execution. Alternatively, we may be unable to resolve some indirections or prove some return addresses, which would result in additional annotations generated or even tool failure. Despite this, in cases with successful completion and no annotations produced, we will always produce all states that would be encountered during concrete execution.

7.4 Algorithm

Algorithm 7.3 provides the base functionality for HG extraction. That base functionality is then extended in two ways:

1. the addition of a context-free approach to function calls (see Section 7.4.2) and
2. the prevention of states from being joined when they are incompatible (i.e. not matching Definition 7.23).

In order to do this, the algorithm maintains two objects. First, a *bag* of the produced symbolic states that have yet to be explored. Function EXPLORE is repeatedly called until that *bag* is empty. Second, the current HG *HG*. When the *bag* is empty, *HG* is the final HG.

Furthermore, the algorithm requires a function τ that models instruction semantics. Our implementation supports a wide range of x86-64 instructions, including (conditional) moves and jumps as well as arithmetic, logical and bit-vector operations (sufficient to deal with all the Xen Project binaries). Given a supported instruction and a suitable memory model, function τ transforms its supplied predicate into a set of predicates by symbolically executing the single instruction. The memory model allows τ to take into account information on pointer relations when performing symbolic execution using destination operands that reference memory locations.

The algorithm also requires an expression evaluation function $\text{eval} : \mathbb{E} \times \mathsf{Pred} \mapsto \mathbb{C}$, which maps an expression (containing registers, flags, and dereferenced memory regions) to a constant expression. To that end, it requires the predicate of the current symbolic state. No memory model information is required for this evaluation; that information is only required when writing to a location in memory.

Definition 7.19. Given predicate P, the evaluation of an expression e is defined as follows:

$$\text{eval}(e, P) = \begin{cases} v & \text{if } e = v \text{ is a clause in } P \\ \bot & \text{otherwise} \end{cases}$$

That evaluation function is then used by the algorithm in the process of executing steps according to the following step function:

Definition 7.20. The *symbolic state step function* for symbolic state $\sigma = \langle P, M \rangle$, notation $\textsc{step}_\Sigma(\sigma)$, is defined as:

$$\textsc{step}_\Sigma(\sigma) \stackrel{\text{def}}{=} \{\langle P', M' \rangle \mid P' \in \tau(P, M') \wedge M' \in \text{ins}(R, M)\}$$

where

$$R \stackrel{\text{def}}{=} \{[\text{eval}(a, P), s] \mid [a, s] \text{ used by instruction } i\} - \{\bot\}$$
$$i \stackrel{\text{def}}{=} \text{fetch}(\text{eval}(\texttt{rip}, P))$$

Given the current symbolic state σ, the set of next symbolic states is obtained by performing the following two actions:

- inserting relevant regions into the current memory model and
- applying predicate transformation to the current predicate.

The set of regions is obtained by investigating the operands of the current instruction.

Example 7.21 (Regions from instructions)**.** The instruction `mov qword ptr [rax + 4* rdi], rax` results in one region, $[\texttt{rax} + 4 * \texttt{rdi}, 8]$.

That region is—given the current predicate—evaluated to a constant.

Example 7.22 (Evaluation of region part)**.** If the current predicate contains `rax` = `0x100` and `rdi` = $\texttt{rsi}_0$, then evaluation of `rax` $+ 4 *$ `rdi` produces the constant $0x100 + 4 * \texttt{rsi}_0$.

If the current predicate does not contain enough information to evaluate a state-part-containing region to a constant region, evaluation for that region produces $\bot$ and the region is not inserted. Otherwise, the successfully-evaluated region is inserted. The latter case overapproximates any relation (e.g., aliasing, separation) the new region may have with the current memory model.

It is also possible for lack of information to result in inability to continue predicate execution. Specifically, if no bounded set of continuation states can be determined, we produce an annotation and stop further exploration from that state. This primarily occurs for unresolved indirect jumps and calls.

Definition 7.23 (Compatibility)**.** Two symbolic states σ and σ' are *compatible*, notation $\sigma_0 \cong \sigma_1$, if and only if their instruction pointers (`rip`) are equal. States will only be joined when they are compatible.

An extension to the base algorithm presented below in Algorithm 7.3 modifies this definition. Specifically, it adds the constraint that states are not considered compatible when shared registers are assigned different immediate values that directly influence control flow (for example, when the immediates are loaded from a jump table). In general, it is impossible to know whether or not a stored value will influence future control flow. However, it is sufficient to detect situations in which values will *likely* influence future control flow.

This modification to compatibility does not cause any soundness issues. If a value was erroneously deemed to influence future control flow, then we have unnecessarily explored paths that could have been joined earlier, but this only affects analysis time and not soundness. Joining states that contain immediate values that turn out to be necessary to assess future control flow is no issue either. It will merely lead to unresolved indirections or errors being reported by the analysis, and soundness will still not be affected.

Concretely, if two states have assignments of different immediate pointers to instructions[4] to the same state part, then we do not join those pointers to an abstract value. Instead, we continue exploration from both states. This is because those immediate values are highly likely to influence future control flow. Such an approach causes less abstraction, but does allow us to resolve more indirections. In very specific cases it even results in more preciseness.

7.4.1 Base Algorithm

Let σ be some symbolic state from the *bag* (Line 2). Function EXPLORE first searches for a current symbolic state σ_c already in the current *HG* that is compatible (Line 3). If such a state exists and it is more abstract (based on ordering $\sqsubseteq$) than state σ, no further exploration is necessary (Line 4). Otherwise, σ and σ_c are joined (Line 5). The *HG* is modified by replacing the current state with the joined one. This replacement maintains all current edges: only the state is modified. Symbolic state σ_j is the state to be explored further. If no

[4] concrete pointer values that fall within the range of the text section of the binary

Algorithm 7.3 Base version of HG extraction

```
1:  function EXPLORE
2:      pop σ from bag
3:      if ∃σ_c ∈ HG · σ_c ≅ σ then
4:          if σ ⊑ σ_c then return
5:          σ_j := σ ⊔ σ_c
6:          HG[σ_c := σ_j]
7:      else
8:          σ_j := σ
9:      for all σ' ∈ STEP_Σ(σ_j) do
10:         HG += (σ_j, σ')
11:         if eval(rip, pred(σ')) is not immediate then
12:             annotate, stop further exploration
13:         else
14:             bag += σ'
```

compatible state exists in the current HG, then σ is the state to be explored further (Line 8). Exploration occurs at Lines 9 to 14. For every successor σ' (possibly none), an edge is added to the HG. If, for some successor evaluation of the instruction pointer, `rip` does not produce an immediate concrete value, there are two possibilities. Either

1. a return statement has been encountered (after which `rip` is set to the symbol pushed to the top of the stack in the initial state), or
2. the current symbolic state does not provide sufficient information to resolve the computation of `rip` (because of an indirect branch, for example).

In the second case, the state is annotated with an unsoundness warning (Line 12) and the algorithm terminates early. Otherwise, the successor is added to the bag.

Soundness

To formulate soundness and present a proof, we first define a relation $\mathbf{R}$ between the concrete transition system and the HG. We then prove Lemma 7.25, which shows that this relation is a simulation. As a direct result of this lemma, any concrete path can be simulated by a path consisting of symbolic steps produced by function STEP_Σ.

Definition 7.24 (Related to)**.** A concrete state s is *related to* symbolic state $\sigma = \langle P, M \rangle$, notation $s \mathbf{R} \sigma$, if and only if:

$$s \mathbf{R} \sigma \stackrel{\text{def}}{=} (s \vdash P) \wedge (s \vdash M)$$

Lemma 7.25 ("Related to" is a simulation)**.** *Assume that predicate transformation τ is correct:*

$$\forall s\ s' \cdot s \rightarrow_B s' \wedge (s \vdash P) \implies \exists Q \in \tau(P, M) \cdot s' \vdash Q.$$

Then relation $\mathbf{R}$ *is a simulation between the concrete transition system and the transition system obtained by abstract step function* $\textsc{step}_\Sigma$*:*

$$\forall s\ s' \cdot s \to_B s' \wedge s\ \mathbf{R}\ \sigma \implies \exists \sigma' \in \textsc{step}_\Sigma(\sigma) \cdot s'\ \mathbf{R}\ \sigma'$$

Proof. Let s and σ be two related states. Hence $(s \vdash P) \wedge (s \vdash M)$. By correctness of τ, we obtain a predicate $Q \in \tau(P, M)$ such that $s' \vdash Q$. By Lemma 7.12 (completeness of the insertion function), the memory model that holds in state s' is generated. Since the step function overapproximates by taking any combination of predicates in $\tau(P, M)$ and generated memory models, there exists at least one symbolic state that is related to s'. □

Definition 7.26 (HG soundness)**.** The HG $H = \langle \mathbb{\Sigma}, \sigma_I, \to_\Sigma \rangle$ is *sound* with respect to some binary $B = \langle a_e, \text{fetch}, \mathbb{S}, \to_B \rangle$, notation $\text{sound}(H, B)$, if and only if:

$$\text{sound}(H, B) \equiv \forall s_0 \to_B^* s \to_B s' \cdot \exists \sigma \to_\Sigma \sigma' \cdot s\ \mathbf{R}\ \sigma \wedge s'\ \mathbf{R}\ \sigma'$$

In words, for every reachable transition from s to s' in the binary, there must exist a related transition in the HG.

Theorem 7.27 (Algorithm soundness)**.** *Algorithm 7.3 constructs a sound HG.*

Proof. The structure of the algorithm is close to a depth-first search (DFS). For that reason, the white-path lemma is used to prove soundness [43]. For a normal DFS, the white-path lemma states that the DFS will eventually explore some state s' if and only if there exists some state s currently in the bag *and* there exists a "white" path from s to s'. A key difference between Algorithm 7.3 and a normal DFS is that states are joined. For the sake of this proof, a state is therefore considered "white" if the current HG contains no compatible state that is equal or more abstract (under $\sqsubseteq$). We reformulate the white-path lemma as follows:

$$\begin{aligned}
&\text{sup}(\sigma') \text{ is explored} \iff \\
&\exists \sigma \in \mathit{bag} \cdot \exists \pi = [\sigma, \ldots, \sigma'] \cdot \mathsf{white}(\pi) \\
&\textbf{where} \\
&\text{sup}(\sigma') \equiv \bigsqcup \{\sigma'' \mid \sigma'' \cong \sigma' \wedge \exists \pi = [\sigma, \ldots, \sigma''] \cdot \mathsf{white}(\pi)\}
\end{aligned}$$

In words, $\text{sup}(\sigma')$, the supremum of all compatible states that are currently reachable through white paths, is explored by the algorithm if and only if there exists a white path from some σ currently in the bag to σ'. Given this version of the white-path lemma, it directly follows that if the bag initially contains the initial state only:

$$\text{sup}(\sigma') \text{ is explored} \iff \sigma' \text{ is reachable from } \sigma_0$$

Now let s be a reachable concrete state and s' be a successor. Lemma 7.25 shows that the path from s_0 to s can be simulated by a path of related symbolic states. Let σ be the symbolic state related to concrete state s, i.e., $s\ \mathbf{R}\ \sigma$. Since σ is reachable, $\text{sup}(\sigma)$ is explored. We thus have $s\ \mathbf{R}\ \sigma \implies s\ \mathbf{R}\ \text{sup}(\sigma)$. This is a direct implication of Lemma 7.16: since joining makes the states more abstract, it makes the set of related concrete states larger. Line 9 will then explore some state $\sigma' \in \textsc{step}_\Sigma(\sigma_j)$. By Lemma 7.25, we have $s'\ \mathbf{R}\ \sigma'$. □

7.4.2 Extension: Function Calls

The base algorithm as presented in Algorithm 7.3 does not treat function calls as special instructions. This is unsatisfactory for two reasons. First, for *external* function calls, a function τ that transforms the predicate may not be available. External function calls are dynamically linked and thus the assembly instructions are not available during static analysis. Second, even though *internal* function calls theoretically pose no problem, simply unfolding every function call prevents scalability. We present an extension to the algorithm that treats internal function calls compositionally. That is, it ensures that each function is explored only once.

External Functions

The function name is matched against a list of hard-coded function names that are known to be terminating, such as `exit` and `stack_chk_fail`. In case of a terminating function, function $\textsc{step}_\Sigma$ will produce the empty set, stopping further exploration from the current state. Otherwise, the function is some unknown external function. We make the assumption that this unknown function adheres to the 64-bit System V ABI's calling convention. Function $\textsc{step}_\Sigma$ therefore modifies the current state by assigning $\bot$ to all registers, flags and heap regions currently in the state that may not be assumed to be preserved by a function call. In other words, only the clauses concerning the stack frame and callee-saved non-volatile registers are kept. Similarly, all relations in the memory model concerning the heap are removed. We call this *cleaning* the current symbolic state. As with the join operation, this usage is sound as the end result is always a weakening of the postcondition.

Internal Functions

If the operand of a `call` can be resolved to an address inside the executable range of the binary, it is recognized as an internal call. Consider the following assembly code:

Function Call	Return	Exit
`100: call 400`	`400: ...`	`400: ...`
`105: ...`	`450: ret`	`450: call exit`

Intuitively, exploration from address `0x100` can proceed both at addresses `0x400` (entering the function) and at `0x105` (after the function). The latter, however, may not safely be assumed, as it is not known whether the called function returns normally. A function may always exit, in which case address `0x105` is never visited. Other issues, such as buffer overflows, can prevent a normal return as well.

We therefore introduce the notion of *reachability*. Each symbolic state has a Boolean field ($\mathbb{B}$) that is set to true only if the state is known to be reachable. States in the bag whose reachability field is false are not selected. Line 3 of the algorithm becomes:

3: **if** $\exists \sigma_c \in HG \cdot \sigma_c \cong \sigma \wedge \mathsf{reachable}(\sigma_c)$ **then**

Moreover, after Line 14, any symbolic state with the same `rip` as the newly explored σ' is marked as reachable:

14: mark all $\sigma \in bag$ with $\texttt{rip}(\sigma) = \texttt{rip}(\sigma')$ as reachable

As stated, we treat internal function calls as *context free*. In the example above, exploration of address `0x400` is done in a fresh empty symbolic state. In that state, instead of pushing the concrete return address `0x105`, a symbol $\mathcal{S}_{\texttt{0x400}}$ is pushed. As a result, wherever the internal function is called, it will always start in the exact same state and therefore exploration happens only once.

A global mapping is maintained that remembers that symbol $\mathcal{S}_{\texttt{0x400}}$ is linked to return address `0x105`. It may be the case that the internal function is called from different call sites, in which case the mapping is updated accordingly: one symbol may be mapped to multiple return addresses. As soon as the instruction pointer is set to symbol $\mathcal{S}_{\texttt{0x400}}$ (for example, by a `ret` instruction), all mapped return addresses are set to reachable.

To summarize, for the internal function call above, the following actions are undertaken:

1. Clean the current symbolic state and add it—with `rip` set to `0x105`—to the bag. At this point, it is marked as *unreachable*.
2. Add an empty state to the bag, with a symbolic return address $\mathcal{S}_{\texttt{0x400}}$ pushed to the top of the stack.
3. Add return address `0x105` to the set of addresses mapped to symbol $\mathcal{S}_{\texttt{0x400}}$.
4. As soon as an instruction sets `rip` to the symbol $\mathcal{S}_{\texttt{0x400}}$, any symbolic state whose `rip` is in the set of return addresses mapped to that symbol is set to *reachable*.

Chapter 8

Experimental Results

This chapter covers our real-world exploration of the HG generation algorithm. It also includes a verification methodology for the generated HGs. Finally, it discusses some of the potential error cases that would prevent successful verification.

The work in Sections 8.2 and 8.3 is credited primarily to Dr. Freek Verbeek. I have included it in this dissertation for completeness and consistency, as well as to have a space where it can be seen without the restrictions of a limited-page paper format.

8.1 Hoare Graph Extraction

We applied HG extraction to:

1. several stripped binaries of CoreUtils as found in a standard Ubuntu distribution;
2. a binary with a manually induced buffer overflow, confirming that no HG is extracted; and
3. all 63 x86-64 binaries and all 2151 functions from the 25 shared objects we identified in the the Xen Project hypervisor.

This section focuses on the Xen case study specifically. The Xen Project is a mature, industrial-strength hypervisor used in many production systems, including Amazon's cloud platforms [34]. Hypervisors provide a method for managing multiple virtual instances of operating systems (guests) on a physical host. Xen is a suitable case study because of two things:

- its complexity and
- the wide range of programs and shared libraries produced by its build process.

The analysis was performed on a machine with a six-core (twelve logical cores), 2.9 GHz Intel Core i9-8950HK CPU. That machine had 31 GiB of random-access memory (RAM) and 32 GiB of swap space on a KXG50PNV1T02 NVMe solid-state drive (SSD). Its OS was Linux Mint 20.1 Cinnamon. The version of Xen under test was 4.12.

Table 8.1: Xen case study statistics summary

Directory		Instrs.	Symbolic States	A	B	C	Time/ h:m:s
	Binaries						
`bin`	15 = 12 + 2 + 1 +0	6751	6829	21	19	0	0:15:54
`xen/bin`	17 = 7 + 1 + 8 +1	2433	2468	8	3	3	0:01:17
`libexec`	1 = 1 + 0 + 0 +0	82	87	1	0	0	0:00:10
`sbin`	30 = 25 + 1 + 4 +0	8858	9178	26	4	8	0:18:39
Total	63 = 45 + 3 +13+1	18 124	18 562	56	26	11	0:35:59
	Library functions						
`lib`	1907=1874+29+ 0 +4	353 433	362 635	1	244	600	15:28:17
`xenfsimage`	109 = 106 + 3 + 0 +0	17 184	17 683	0	0	27	1:58:36
`dist-packages`	16 = 16 + 0 + 0 +0	379	407	0	0	3	0:00:06
`lowlevel`	119 = 119 + 0 + 0 +0	10 651	10 799	0	0	90	0:08:43
Total	2151=2115+32+ 0 +4	381 647	391 524	1	244	720	17:35:42

$w + x + y + z$: w lifted, x unprovable return address, y concurrency, z timeout
A = Resolved indirection B = Unresolved jump(s) C = Unresolved call(s)

Remark 8.1 (Parallelization). Our tool for HG extraction is not in itself parallelized. That means the core count listed above did not directly affect the execution times shown below. However, the only restriction on running multiple instances of the tool simultaneously is the availability of system resources. Thus, in the published artifact [21] we provided examples of using GNU parallel [170] to perform analyses efficiently.

Table 8.1 shows an overview of the results. The upper part of the table shows binaries. Lifting of the binaries was done by starting the extraction algorithm at each binary's ELF entry point and exploring all reachable assembly instructions. This includes all resolvable internal function calls. The lower part shows library functions in shared objects (SOs). For every SO file, all externally exposed functions as reported by the `nm` utility were considered. Lifting individual functions required starting the extraction algorithm at the function's address and exploring all reachable assembly instructions from that point. As with the binaries, that included resolvable calls to other internal functions.

8.1.1 Failure Cases

Three issues may prevent lifting a binary to an HG, shown in the second column of Table 8.1. These issues are explained below.

Unprovable Return Addresses

This case calls back to Section 6.1. When a `ret` instruction is encountered, the current precondition[1] must be strong enough to prove that the return address at the top of the current stack frame has not been modified. Furthermore, that precondition must also be strong enough to show that the value of the stack pointer has been restored to the initial value it held on function entry. If the current precondition is not strong enough to satisfy those conditions, the algorithm does not produce an HG. This is because it cannot prove return address integrity. A breakdown of these cases occurs later in Section 8.1.4.

Concurrency

Binaries that contain multithreading-related function calls are out of scope for this analysis. This was determined primarily by the presence of `pthread` calls, mutexes, and semaphores. However, those binaries were still included in Table 8.1 in order to account for all x86-64 Xen binaries.

Timeout

While our algorithm has a proof of termination, state space explosion or other resource limitations sometimes make full execution infeasible. In order to ensure a full analysis, we set a timeout on analysis to 4 h per binary/function. This resulted in failure for only one binary and four library functions. Further discussion of the function timeouts can be found later in Section 8.1.3.

8.1.2 Successful Cases

As a reminder, the basic sanity properties being checked were return address integrity, bounded control flow, and calling convention adherence. In total, those properties could be proven and an HG could be generated for 45 out of 63 binaries and 2115 out of 2151 library functions.

The third and fourth columns of Table 8.1 show the number of instructions lifted and the number of states of the HG. Taking both the binaries and shared objects into account, 399 771 instructions were lifted. As states belonging to the same address are joined whenever compatible, the number of resultant states is close to the number of instructions. State overhead scales with the amount of predicted control flow-related immediate values. This was described in the addendum to Definition 7.23.

Next, Column A shows the number of *resolved* indirections. By resolved, we mean the indirect jumps and calls where the postcondition's instruction pointer value could be overapproximatively established. Meanwhile, columns B and C show the number of *annotations* for unresolved indirect jumps and calls, respectively. Unresolved indirect jumps were primarily caused by two things. First, a handcrafted jump-table-matching heuristic that did not necessarily match all possible jump tables. Second, a lack of context for those indirect jumps

[1] symbolic state predicate

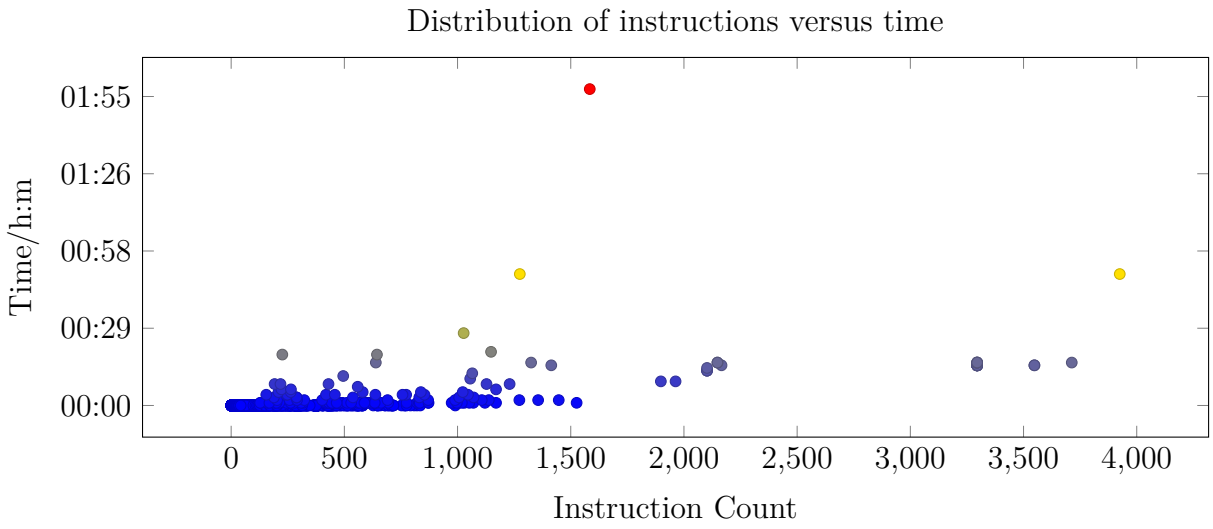


Figure 8.1: Case study library function timing analysis

not involving jump tables. Similarly, unresolved indirect calls are often caused by a lack of context for function callbacks. In such cases, a function pointer is passed as a parameter (or through a global variable) from function to function. Programmer-supplied function arrays are another source of non-resolution. As function calls are handled without context, the specific function pointer to select is unknown at call time. The utilized jump table heuristic cannot deal with such cases either as it cannot process the programmer-provided ELF data sections.

8.1.3 Timing

Figure 8.1 relates the instruction count of library functions to their verification time. The largest function successfully verified was `libxl_domain_suspend` from `libxenlight.so.4.12.0`, with 3925 instructions and 4207 symbolic states. Its analysis took 49 minutes and 10 seconds to complete. The second-largest function verified, `libxl_domain_suspend_only`, had 3713 instructions with 4100 symbolic states and took 16 minutes 34 seconds to complete. Interestingly, the longest verification time did *not* inolve the largest function. It was instead an outlier. Function `libxl_domain_build_info_gen_json` took around 2 hours despite having only 1584 instructions.

As shown in Table 8.1 and as previously mentioned in Section 8.1.1, out of the 2151 library functions, we had 4 timeouts.[2] The functions that timed out have a large number of states that could not be joined, causing explosion in the number of states to be explored. In general, while some correlation exists between instruction count and verification time, there are major outliers. It can thus be inferred that complexity of the control flow itself is a major contributor to analysis time.

[2]Because they timed out, they were not included in the 15:28 total verification time.

8.1.4 Summary

To reiterate, we lifted an HG for 2115 out 2151 functions (98 %). We can account for why this number is relatively high:

- For many functions, any pair of pointers *to the local stack frame* abided by any of the four relations for which we accurately model memory relations (aliasing, separations, enclosed within, encloses). As a result, even if the heap and the global memory space were grossly overapproximated, the local stack frame was modeled accurately and return address integrity could be proven.
- In the case of an unresolved function call, we treated the function overapproximatively as an unknown external function, allowing continued analysis. Typical reasons for unresolved indirections include callbacks: a function pointer is set by some function f and is retrieved and called back in function g. A context-sensitive approach would be able to increase the number of supported indirect calls, but this would need to be done in a sufficiently scalable way.
- Even though not all instructions of the x86-64 ISA are supported, all instructions occurring in the case study are, so this is not a reason why functions were rejected.

The unprovable return address cases can also be broken down further:

- Some of the rejections constitute functions that do not adhere to the calling conventions of the System V ABI. Manual analysis of these cases shows that these are all compiler-generated functions that are not required to follow calling conventions.
- Other rejections were caused by preconditions that were insufficiently strong enough for the derivation of overapproximative bounded sets of concrete values for the next instruction pointers. This may occur when an array or struct is stored on the stack and accessed via variable offset. Such constructs may lead to complicated pointer arithmetic *within* the stack frame. The result is that the algorithm cannot prove that a memory region was separate from the top of the stack frame that stores the return address.

8.2 Formal Proofs in Isabelle/HOL

For several CoreUtils binaries, we extracted an HG and exported it to the Isabelle/HOL theorem prover. The binaries are closed-source, taken from a standard MacOS 11.5.2 distribution. Table 8.2 provides an overview of the binaries, the number of instructions and Hoare triples, and the number of resolved indirections (there are no unresolved indirections). Without exception, all Hoare triples could be proven automatically.

We have developed a formal model of the semantics of roughly 120 different x86-64 assembly instructions. These instructions include various moves, arithmetic/logical operations, jumps, and `call`/`ret`. Floating-point operations are mapped to uninterpreted functions. The model provides semantics for register aliasing and a byte-level little-endian memory model. Moreover, we have developed a symbolic execution engine that applies the formal semantics of an instruction to a symbolic state, and matches that to a given postcondition. This engine

Table 8.2: Overview of binaries exported to Isabelle/HOL

Binary	Number of Instructions	Number of Indirections
`hexdump`	2515	11
`od`	3040	11
`wc`	445	0
`tar`	5730	5
`du`	883	3
`gzip`	3465	7
Total	16 078	37

is based on a library of formally proven correct simplification theorems, as well as theorems that prove separation properties over different memory writes. Finally, we support automatic generation of implicit assumptions necessary for formal proofs. The informal algorithm can implicitly make assumptions that, e.g., regions in the global memory space are not overlapping with regions from the stack frame. A formal proof must explicitly assume that. Effectively, each and any implicit assumption made during HG generation is formalized and exported to Isabelle/HOL.

8.3 Examples of Failures

This section covers some specific failure cases for our verification tool.

8.3.1 Stack Overflow

ROP emporium[3] provides pedagogical examples that contain an exploitable stack overflow. For the `ret2win` example, the exploit simply amounts to ensuring that a call to `memset` writes 48 bytes to its given pointer. For our tool, `memset` is an unknown external function, and thus it is annotated with the assumptions needed to ensure return address integrity. The annotation states that:

$$\texttt{@400701} : \texttt{memset}(\texttt{RDI} := \texttt{RSP}_0 - 40)\ \texttt{MUST PRESERVE}\ [\texttt{RSP}_0 - 8\ \texttt{TO}\ \texttt{RSP}_0 + 8]$$

At address `0x400701`, a call to `memset` occurs with as first parameter a pointer into the stack frame of the caller ($\texttt{RSP}_0 - 40$). The algorithm needed to assert that this call did not overwrite the memory region $[\texttt{RSP}_0 - 8\ \texttt{TO}\ \texttt{RSP}_0 + 8]$, where, among other things, the return address is stored. In other words, the algorithm asserted and noted as proof obligation that the write executed by `memset` did not exceed 32 bytes. In this example, the algorithm did not produce a verification error, but generated proof obligations that can be violated. Such a violation constituted an exploit candidate.

[3] https://ropemporium.com

8.3.2 Stack Probing

In the binary `/usr/bin/zip`—as available in a standard MacOS distribution—a certain function executes the following function call:

```
100009fe6:  mov eax, 0x1400
100009feb:  call 0x10000a6a0
100009ff0:  sub rsp, rax
```

Register `rax` gets the value `0x1400`, then an internal function is called, and then the number of bytes in `rax` is allocated locally on the stack frame. The called function executes a compiler-generated technique called *stack probing*. That function traverses the stack and reads-then-discards individual bytes below the current stack pointer at intervals of `0x1000` bytes. The instruction at address `0x100009ff0` eventually causes a verification error, since the tool cannot establish whether register **`rax`** has been modified during that function call.

8.3.3 Nonstandard Stack Pointer Restoration

Normally, a function restores the stackpointer **`rsp`** to its initial value, plus eight due to the pushing of the return address. That is, after a **`ret`** statement the symbolic state is verified for:

$$\texttt{RSP} == \texttt{RSP}_0 + 8$$

In the binary `/usr/bin/ssh`—as available in a standard MacOS distribution—a function returns with the stackpointer **`rsp`** set to the following symbolic value:

$$\begin{array}{l}\texttt{RSP} == *[\\ \quad ((\texttt{RSP}_0 - (48 - (((-4) - \texttt{R9}_0) * 8))) \;\&\; (-400)) \\ \quad\; + \;((\texttt{udiv64}(\texttt{R9}_0, 4) * 4) * 8) + 8 \\ \quad] + 56\end{array}$$

This complicated symbolic value shows that the stackpointer is not normally restored, but instead read from a region in memory whose address is based on the initial value of register `R9` (as before, notation $*[\texttt{a}]$ denotes reading from address `a`). This function leads to a verification error, as the stack pointer cannot be proven to be normally restored, no accurate memory relations over the local stack frame can be formulated.

Chapter 9

Discussion

Having covered the basic concepts (Chapter 6), formulation (Chapter 7), and experimental results (Chapter 9) of HGs, we draw to an end for this part of the dissertation. This chapter wraps things up with a high-level discussion of how the assumptions we made in Section 1.7.2 affect the usability of overapproximative binary lifting in various application domains. It also covers the importance of the per-instruction disassembler (a part of the TCB) being correct.

9.1 Security Analysis

The central claim in this paper is that *if* all assumptions and proof obligations are met, *then* the lifted representation is a sound overapproximation of the binary. Section 8.3 showed an example where an assumption can be violated: `memset` may not preserve the indicated region. The negation of assumptions required for "normal" behavior may lead to "weird" behavior. In other words, the negation of the generated assumptions may be useful in the generation of exploits. A key challenge here is to filter out the relevant (exploitable) assumptions from the irrelevant ones.

9.2 Binary Verification

We argue that the majority of existing work on binary verification *assumes* the existence of a trustworthy disassembler. This work exposes and makes explicit assumptions that otherwise may remain implicit. Therefore, basing a verification effort on a verified HG instead of directly on the output of any of-the-shelf disassembler reduces the TCB of the verification effort.

9.3 Decompilation

Similarly, we argue that the majority of existing decompilation tools *assume* the existence of a reliable disassembler. A verified HG is a reliable base for decompilation. For example, the

provably correct assembly and control flow inferred by our approach could be used as input to McSema [56] in order to produce provably correct LLVM code. The assumptions then may be translated to higher-level `assert`-statements: the decompiled code is correct as long as no assert is triggered.

9.4 Patching

Binary patching typically either involves some stages of decompilation, or replacing snippets of assembly instructions with different ones [57]. We argue that lifting both an original binary and its patched version to HGs would increase the trustworthiness of the patch effort. Both the HGs—but also the assumptions required for lifting the binaries—could be mutually compared, and this comparison may expose unexpected effects of the patch.

9.5 Do Not Forget to Check Your Disassembler!

The library we were originally using to interpret the binary representation of individual instructions did not correctly identify the size of operands for certain instructions, such as some SIMD ones. That size differential had the potential to cause bad semantics or even errors in Z3 usage due to incorrect typing and value truncations or lack thereof. However, we did not uncover this issue until a compatriot (Ian Roessle) noticed incorrect disassembly output while working on a separate project using a portion of our codebase. To solve this issue, we switched to Capstone [4]. When tested on the binary our compatriot was working with, the individual instructions were extracted as expected.

This is an example of why it is important to reduce the TCB as much as possible. If your assumptions of trust are misplaced, such as an unverified component of your disassembler being buggy, the results as a whole have the potential to be wrong.

Part IV

Exceptional Interprocedural Control Flow Graphs

Chapter 10

Exceptional Interprocedural Control Flow Graphs

HGs work well for C programs or others that do not use structured EH, but they do not provide the full picture for programs that do. For this purpose, we provide *exceptional interprocedural control flow graphs (EICFGs)*. These EICFGs contain control flow edges for the process of structured EH. Those edges are obtained by modeling the C++ EH ABI [32]. The states used in EICFGs are specifically targeted to the process of EH, focusing on information about the exception objects and global exception info. Enough additional information and context are included to support indirect jumps and calls.

We first provide more motivation for EICFGs in Section 10.1. This is followed by an abbreviated explanation of EICFGs in Section 10.2, which is illustrated by a running example in Section 10.3. After that is the technical formulation of our tool in Chapter 11. Chapter 12 contains single-step validation in Section 12.1 and a practical demonstration in Section 12.2. The goal of that demonstration is to show that EICFGs provide improved coverage compared to a non-exceptional baseline analysis.

All code produced for this part, to be published as an artifact, is available at `https://drive.google.com/file/d/1mFjHT0p-w9YbyRjcBCQ8sPHp3eboUdM2/view?usp=sharing`.

10.1 Extended Motivation

A normal CFG provides a user with information on control flow transfers induced by *jumps* and *calls*. Given a specific jump or call instruction, one can look up in the CFG the set of successor instructions and the information on which that successor selection is based. For sets with more than one element, such decisions are typically represented by expressions over flags (for example, `CF` and `ZF` for the carry and zero flag) or a jump table calculation.

The labels of a CFG can be seen as *state predicates*. That is, when an edge is labeled with a flag such as `ZF`, that notation can be seen as a predicate on the value of the zero flag in the state of the originating node. More complex predicates can consist of logical expressions such

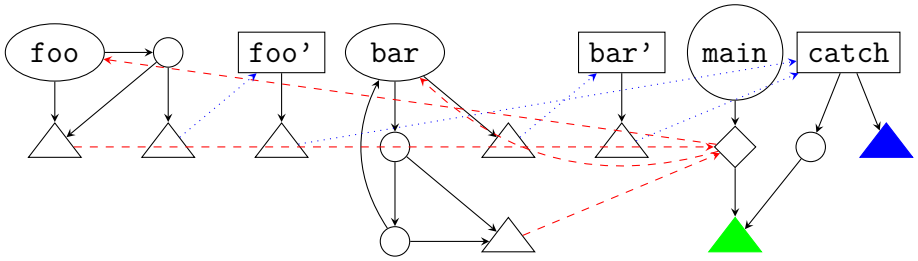


Figure 10.1: Ghidra-generated graph (summarized) with basic EICFG edges added

as `CF` $\wedge$ `ZF`, which checks if both `CF` and `ZF` are set. Higher-level predicates can have clauses such as `rax` $= 0$ or `DWORD PTR [rdi]` < 5.

However, traditional CFGs produced by existing static analysis tools lack information on interprocedural indirect control flow. This includes stack unwinding due to exception throwing, try-catch handling, and C++-style object cleanup. This information is not present because their analyses do not model the semantics of throw operations, even when static try-catch information is available.

A summarized reproduction of the interprocedural control flow graph generated by Ghidra for an example program (Listing 10.1) can be seen in Fig. 10.1. While Ghidra can identify catch and cleanup landing pads (`foo'`, `bar'`, and `catch`), it cannot directly show that the throws in `foo` and `bar` will unwind there. Furthermore, it does not show that `foo` and `bar` can be indirectly called from `main`. The diamond in the graph indicates the indirect call location, but the only edge from it is the edge after its return (to a triangle). IDA Pro and Binary Ninja produce similar results; they can identify landing pad locations intraprocedurally, but they cannot trace exceptional control flow interprocedurally. We have verified that this holds for more complicated programs as well (some of the programs used in Section 12.2).

10.2 EICFGs

Therefore, we produce an EICFG, which augments a normal CFG with edges produced by `try`, `catch`, and `throw` behavior. Specifically, it contains the edges for control flow produced by the execution of the compiled form of the C++ `throw` command. These additional edges are the dotted blue ones in Fig. 10.1. We also illustrate the edges for standard interprocedural control flow as dashed red ones.

Instead of simple labels, EICFGs use *exceptional state predicates* as labels. Informally, the information on which exceptional control flow is based is:

1. the exception object, e.g., type info and rethrown state;
2. the current set of return addresses on the stack;
3. the current uncaught exception count;
4. the current caught exception stack; and

5. a static address (landing pad) table for the unwinding process.

Exception type info is used to determine which catch blocks, if any, are applicable to the exception being thrown. Rethrown status is necessary when determining behavior when dealing with the binary equivalent of an argumentless `throw`. The return address stack is necessary to provide context for unwinding. The uncaught exception count keeps track of how many thrown exceptions are currently uncaught. This is useful for diagnostic information. Next, the caught exception stack provides information to set up implicit rethrowing. Finally, the landing pad table (LPT) maps from potential unwind spots in a binary to locations where unwinding can exit. The full formal definition of the abstract states involved can be found later in Section 11.2.

Definition 10.1 (Exceptional state predicate)**.** An *exceptional state predicate* is based on an exception object E, a return address stack R, uncaught exception count u, caught exception stack C, and instruction pointer I. It is a predicate that, given a state, checks the following things:

- Is the current exception object equal to E?
- Is the current return address stack equal to R?
- Is the current uncaught exception count equal to u?
- Is the current caught exception stack equal to C?
- Is the current `rip` equal to I?

Example 10.2 (Try and catch)**.** Consider the catch statement on Line 32 of Listing 10.1. Control flow will reach the contents of that catch statement's block if and only if the exceptions propagated to the corresponding `try` block are instances of `std::exception` or a subclass.

We here provide a formal definition of an EICFG.

Definition 10.3 (EICFGs)**.** An EICFG is a directed graph with instruction addresses as vertices and edges labelled with exceptional state predicates. There is an edge between instruction addresses a_0 and a_1 with label P if the instruction at address a_0 leads to instruction address a_1 for any state that satisfies predicate P.

10.3 Running Example

Some of the additional information provided by an EICFG is illustrated in Fig. 10.2, which models the process of throwing an exception from the same example program as Fig. 10.1. The representation in the figure indicates the process of unwinding from one throw site in the control flow graph to a try-catch block or cleanup landing pad. This path was triggered by the snippet of assembly shown in Listing 10.2, which allocates (`0x125b`), initializes (`0x126d`), and throws (`0x1286`) an exception. The landing pad table of the binary, LPT, directs the unwinding process: when stack unwinding reaches address i and $j \in \mathsf{LPT}(i)$, control flow

Listing 10.1: Example program

```
#include <stdexcept>

int foo(int x) {
    if (x < 0) {
        return x;
    }
    while (x > 0) {
        x--;
        if (x == 5) {
            throw std::domain_error("5");
        }
    }
    return 0;
}

int bar(int x) {
    if (x < 0) {
        throw std::out_of_range("negative");
    } else if (x == 0) {
        return 1;
    }
    return x * bar(x - 1);
}

int (*const FOOBAR[])(int) = {foo, bar};

int main(int argc, char* argv[]) {
    try {
        if (argc < 2) {
            return FOOBAR[argc](argc);
        }
    } catch (const std::exception&) {
        if (argc < 0) {
            throw;
        }
        return 0;
    }
}
```

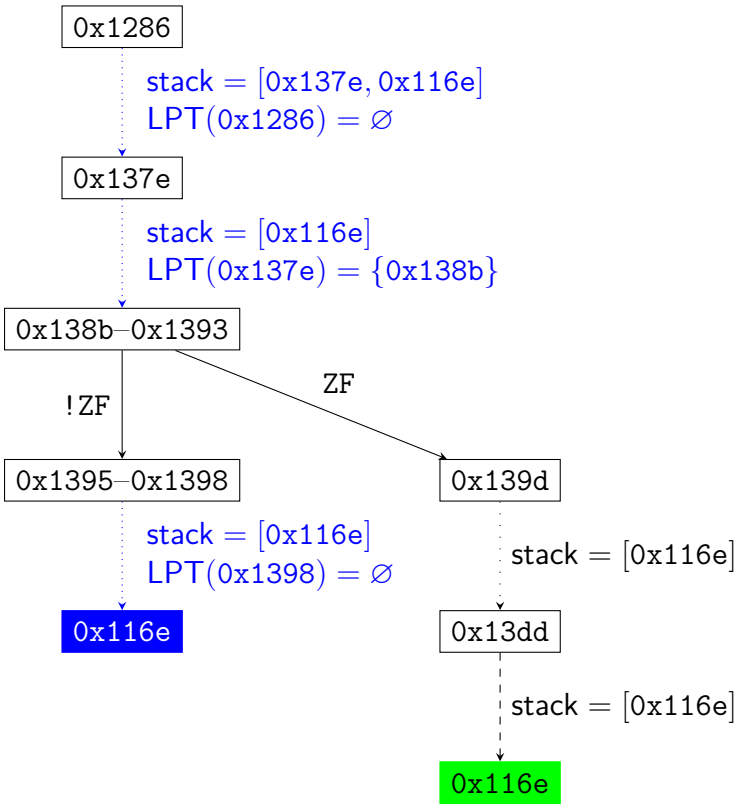


Figure 10.2: Throwing an exception

Listing 10.2: Example throw

```
125b:   call   10d0 <__cxa_allocate_exception>
1260:   mov    rbx,rax
1263:   lea    rsi,[rip+0xd9b] # 2005
126a:   mov    rdi,rbx
126d:   call   10e0 # std::domain_error init
1272:   mov    rax,QWORD PTR [rip+0x2d4f]
1279:   mov    rdx,rax
127c:   lea    rsi,[rip+0x2abd] # 3d40
1283:   mov    rdi,rbx
1286:   call   1120 <__cxa_throw>
...
129c:   call   10f0 <__cxa_free_exception>
```

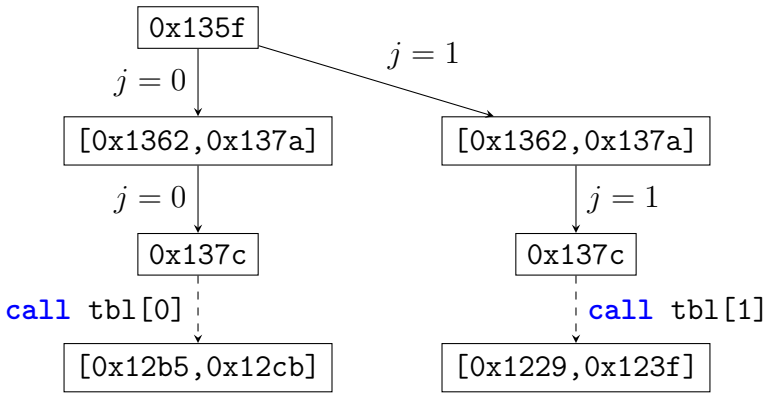


Figure 10.3: Identifying indirection

branches to address j. Otherwise, another frame is popped off the stack. This will be elaborated on in Section 11.1. For this example, we get unwinding from address (`0x1286`) to `0x137e` to `0x138b`.

Due to overapproximation, there are two possible paths from that point. One is the path for an exception object that is not of the caught type, some checks of which occur via the assembly instructions `0x138f` and `0x1393` of Listing 10.3. This path results in unwinding being resumed (`0x1398`) and leads to a bad termination case at instruction `0x116e`. The other path continues from node `0x139d`. It eventually leads to `0x116e` from `0x13dd` (intervening nodes elided). This is a good termination case as reaching that halting instruction occurred outside unwinding.

Meanwhile, indirect function call resolution is shown in Fig. 10.3, which is a representation of Listing 10.4. This process occurs when we detect a value being read from a part of the state that is positively bounded, because such values may be jump table indices. In the given example, that bound is provided by a conditional check previously (instructions `0x1359` and `0x135d` of Listing 10.4). When this happens and the state part being read from does not already have a fixed value, we generate non-deterministic edges where the read value ranges from 0 to the upper bound. This can be seen in the middle graph edges of Fig. 10.3, which connect the same addresses but have different jump table indices. In the compiled assembly of the program, the jump table index selection occurs at `0x135f`. Such values can then be used later on in the calculation of jump table addresses, in this case by `0x136c` and `0x1373`. This non-deterministically results in `0x137c` going to different address ranges, as shown in the graph.

Listing 10.3: Example throw landing pad

```
1387:   mov     eax,ebx
1389:   jmp     13d7 <main+0x92>
138b:   endbr64
138f:   cmp     rdx,0x1
1393:   je      139d <main+0x58>
1395:   mov     rdi,rax
1398:   call    1130 <_Unwind_Resume>
139d:   mov     rdi,rax
13a0:   call    10c0 <__cxa_begin_catch>
13a5:   mov     QWORD PTR [rbp-0x18],rax
13a9:   cmp     DWORD PTR [rbp-0x24],0x0
13ad:   jns     13b4 <main+0x6f>
13af:   call    1100 <__cxa_rethrow>
13b4:   mov     ebx,0x0
13b9:   call    1110 <__cxa_end_catch>
13be:   jmp     1387 <main+0x42>
13c0:   endbr64
...
13d7:   add     rsp,0x28
13db:   pop     rbx
13dc:   pop     rbp
13dd:   ret
```

Listing 10.4: Example indirect call

```
1359:   cmp    DWORD PTR [rbp-0x24],0x1
135d:   jg     1382 <main+0x3d>
135f:   mov    eax,DWORD PTR [rbp-0x24]
1362:   cdqe
1364:   lea    rdx,[rax*8+0x0]
136c:   lea    rax,[rip+0x29ad] # 3d20
1373:   mov    rdx,QWORD PTR [rdx+rax*1]
1377:   mov    eax,DWORD PTR [rbp-0x24]
137a:   mov    edi,eax
137c:   call   rdx
137e:   mov    ebx,eax
```

Listing 10.5: The `_start` function

```
1140:   endbr64
1144:   xor     ebp,ebp
1146:   mov     r9,rdx
1149:   pop     rsi
114a:   mov     rdx,rsp
114d:   and     rsp,0xfffffffffffffff0
1151:   push    rax
1152:   push    rsp
1153:   lea     r8,[rip+0x2f6] # 1450
115a:   lea     rcx,[rip+0x27f] # 13e0
1161:   lea     rdi,[rip+0x1dd] # 1345<main>
1168:   call    QWORD PTR [rip+0x2e6a] # 3fd8
116e:   hlt
```

Chapter 11

EICFG Formulation

The derivation of EICFGs from a binary requires four additional things:

1. a static landing pad table,
2. an abstract state model,
3. an abstract transition relation, and
4. a symbolic execution engine to apply the rules making up our abstract transition relation.

We describe those here.

11.1 Landing Pad Table

This information describes how unwinding should proceed given the unwinding reaching specific locations in a binary. It is extracted from the Common Information Entries (CIEs), Frame Description Entries (FDEs), and language-specific data areas (LSDAs) of the binary under test and assumed to be correct. In our current formulation, an entry in the catch table is merely a pointer to the corresponding landing pad for this entry. While type pointers and exception specifications exist within the LSDAs as well, we do not currently utilize that information.

Thrown exceptions in C++ can be caught by catch blocks. Individual stack frames may require cleanup during the process of unwinding as well. The addresses of those catch blocks and cleanup routines are called landing pads. To accomplish reaching those addresses during unwinding, we require a *landing pad table*.

Definition 11.1 (Landing pad table)**.** A landing pad table LPT is a static map from instruction address to set of possible landing pads. Formally, LPTs have type $\mathbb{P} \to \mathcal{P}(\mathbb{P})$, where $\mathbb{P}$ denotes *immediate 64-bit addresses*.

Currently, we overapproximate and do not include exception type when determining landing pads. The keys of the table are the ranges of addresses to which the corresponding landing pad entry applies, intervals that are open in the lower bound but closed in the upper bound. This interval layout was chosen to support `rip` being incremented at the start of instruction evaluation. It is traditionally represented in the form $(a, b]$, where a is the lower bound of the interval and b is the upper bound.

Example 11.2 (Landing pad table)**.** One of our running example landing pad entries is $\mathsf{LPT}(\texttt{0x137e}) = \{\texttt{0x138b}\}$. Thus, when an unwinding routine reaches instruction `0x137e` of Listing 10.4, that routine will jump to `0x138b`.

11.2 Abstract State

Our exception-containing abstract states, type Σ, are records with named fields. In defining this record type, the following elemental types are used, some of which are new:

- $\mathbb{B}$ and $\mathbb{N}$ denote Booleans and natural numbers respectively;
- $\mathbb{V}$, which may be $\perp$, indicating any or an unknown or undefined value;
- $\mathbb{W}$ denotes *exception IDs*; and
- $\mathbb{T}$ denotes *program termination*, consisting of the set $\{\perp, \texttt{Good}, \texttt{Bad}\}$.

11.2.1 Exception Objects

Definition 11.3 (Exception objects)**.** Exceptions are also records, having type

$$\mathbb{E} := \begin{cases} \mathsf{rethrown} & : \mathbb{B} \\ \mathsf{handlerCount} & : \mathbb{N} \end{cases}$$

This record has two fields. Boolean field rethrown indicates rethrown status of the exception. Natural field handlerCount stores the current count of catch block handlers for the exception.

Σ, meanwhile, has the following fields:

11.2.2 Register Map

The field rmap has type $\mathbb{R} \to \mathbb{V}$. Reading and writing registers smaller than 64 bits (e.g. `ebp` versus `rbp`) requires bit masking and shifting the underlying 64-bit register's value. This behavior is integrated into our symbolic execution engine. Larger registers, such as the `xmm` vector register set, exist as operands in our instruction representation but are not used for state updates or reads.

11.2.3 Call Stack

Maintaining the current list of return addresses in stack is necessary in order to perform stack unwinding and handle thrown exceptions. Doing so is also necessary for detecting

recursion in our framework. With $[X]$ representing a *list* of type X, this field has type $[\mathbb{P}]$. The following functions perform standard stack operations on such lists:

$$\begin{aligned} \text{push} &: X \times [X] \to [X] \\ \text{pop} &: [X] \to [X] \\ \text{peek} &: [X] \to X. \end{aligned}$$

For our purposes, usage of peek, which looks at the top element of stack, assumes a non-empty list.

Example 11.4 (Call stack)**.** Listing 10.5 illustrates the entry point to our example program. For the initial state, we have an empty stack: []. The call at `0x1168` pushes the return address to the stack. Thus, after execution of the call, we have a stack [`0x116e`].

Additionally, instruction `0x1168` of that listing calls `__libc_start_main`. We model that call simply as a call to the function pointer in the `rdi` register, the program's `main` function. For our purposes, that means pushing the instruction following the call onto the stack as a return address. Thus, for some state after the result of transitioning from `0x1168`, the stack is [`0x116e`].

11.2.4 Exception Map

This field, emap, has type $\mathbb{W} \to \mathbb{E}$. When an exception is created, it receives an ID based on its creation location and is stored in emap with that ID as the key.

11.2.5 Termination State

This field, terminated, has type $\mathbb{T}$. It defaults to the bottom value $\bot$, indicating the program or function has not terminated yet. When a path of execution completes, it is set to either `Good` or `Bad`, indicating either normal or abnormal termination, respectively. We treat cases where an exception propagates to the top of the stack without being caught to be such "bad" cases.

11.2.6 Auxiliary Exception Variables

Σ also contains a count of the number of currently-uncaught exceptions (uncaught: $\mathbb{N}$) and a stack of the currently-caught exception IDs (caught: $[\mathbb{W}]$). These fields are manipulated and used during entry to and exit from catch blocks as well as when rethrowing exceptions. This handling comes into play when dealing with nested catch blocks, exceptions (re)thrown within such blocks, etc.

Definition 11.5 (Abstract states)**.** The type of abstract states, notation Σ, is a record

$$\Sigma := \begin{cases} \mathsf{rmap} & : \mathbb{R} \to \mathbb{V} \\ \mathsf{stack} & : [\mathbb{P}] \\ \mathsf{emap} & : \mathbb{W} \to \mathbb{E} \\ \mathsf{terminated} & : \mathbb{T} \\ \mathsf{uncaught} & : \mathbb{N} \\ \mathsf{caught} & : [\mathbb{W}] \end{cases}$$

To ease register references, for some state σ and named register $\mathtt{r}$, the notation $\sigma.\mathtt{r}$ is shorthand for $\sigma.\,\mathsf{rmap}(\mathtt{r})$, e.g. $\sigma.\mathtt{rdi} \equiv \sigma.\,\mathsf{rmap}(\mathtt{rdi})$.

11.3 Abstract Transition Rules

To go along with our definition of EICFGs in Definition 10.3, we have defined our abstract transition relation in terms of logical rules. There are two sets of those rules.

The first set of rules defines behavior for the instructions from the x86-64 ISA. This includes non-deterministic conditional jump handling as well as handling for unknown external functions and a subset of indirect jumps and calls. Those rules and the additional state parts required to model them are elided here. The exception is how we deal with recursion, as that can be explained informally using elements already in Σ. Assume a call to another function inside the binary for some state σ. Then, if the return address to be pushed on the stack is already in $\sigma.\mathsf{stack}$, we instead treat that call as an unmodeled external call and continue execution past it.

The second set of rules provides modeling for exception-related ABI calls. This set of rules is documented in Figs. 11.1 to 11.3 and elaborated on below. The following abbreviations are utilized in those rules.

$$\mathrm{handler}(id, \sigma) \equiv \sigma.\,\mathsf{emap}(id).\mathsf{handlerCount} \tag{11.1}$$
$$\mathrm{rethrown}(id, \sigma) \equiv \sigma.\,\mathsf{emap}(id).\mathsf{rethrown} \tag{11.2}$$
$$\mathrm{pushStack}(fr, \sigma, \sigma') \equiv \sigma'.\mathsf{stack} = \mathrm{push}(fr, \sigma.\mathsf{stack}) \tag{11.3}$$
$$\mathrm{popStack}(\sigma, \sigma') \equiv \sigma'.\mathsf{stack} = \mathrm{pop}(\sigma.\mathsf{stack}) \tag{11.4}$$
$$\mathrm{pushCght}(c, \sigma, \sigma') \equiv \sigma'.\mathsf{caught} = \mathrm{push}(c, \sigma.\mathsf{caught}) \tag{11.5}$$
$$\mathrm{popCaught}(\sigma, \sigma') \equiv \sigma'.\mathsf{caught} = \mathrm{pop}(\sigma.\mathsf{caught}) \tag{11.6}$$

We also have notation for incrementing and decrementing and special handling for certain rules:

$$\mathrm{handler}(id, \sigma'){+}{+} \equiv \mathrm{handler}(id, \sigma') = \mathrm{handler}(id, \sigma) + 1 \tag{11.7}$$
$$\mathrm{handler}(id, \sigma'){-}{-} \equiv \mathrm{handler}(id, \sigma') = \mathrm{handler}(id, \sigma) - 1 \tag{11.8}$$
$$\mathrm{handler}(id, \sigma')\oplus \equiv \mathrm{handler}(id, \sigma') = |\mathrm{handler}(id, \sigma)| + 1 \tag{11.9}$$
$$\mathrm{handler}(id, \sigma')\ominus \equiv \mathrm{handler}(id, \sigma') = -\operatorname{sign}(\mathrm{handler}(id, \sigma)) * (|\mathrm{handler}(id, \sigma)| - 1) \tag{11.10}$$

As a special case, $0\ominus = 0$.

The transition rules are placed into two groups. Group one, in Fig. 11.1, does not involve unwinding. The second, in Fig. 11.3, does.

11.3.1 Non-Unwinding Rules

Figure 11.1a shows the rule for the special starting function `__libc_start_main`. For this rule, we require the post-state's current instruction pointer be restricted to whatever was previously stored in `rdi`, `rcx`, or `r8`. We also require the stored return address to be on the top of a newly-pushed stack frame.

After the call to `__libc_start_main` that is instruction `0x1168` of Listing 10.5, we will have $\sigma'.\texttt{rip} \in \{\texttt{0x1345}, \texttt{0x13e0}, \texttt{0x1450}\}$ and $\sigma'.\mathsf{stack} = [0x116e]$.

Next, Fig. 11.1b illustrates the rule for `__cxa_allocate_exception`. Our modeling assumes a system where virtual memory allocations always succeed (and the runtime terminates programs when they use up too much memory). It results in an exception object added to the exception map with the post-state σ''s instruction pointer as its ID. The object starts in a non-rethrown state and with no handlers. The ID is also set as the return value of the function in $\sigma'.\texttt{rax}$.

After instruction `0x125b` of Listing 10.2, we have:

$$\sigma'.\texttt{rax} = \texttt{0x1260}$$
$$\sigma'.\,\mathsf{emap}(\texttt{0x1260}).\mathsf{handlerCount} = 0$$
$$\neg\sigma'.\,\mathsf{emap}(\texttt{0x1260}).\mathsf{rethrown}.$$

The rule in Fig. 11.1c is for function `__cxa_free_exception`. This rule ensures the absence of an exception in the exception map based on the given ID. At our level of abstraction, `_Unwind_DeleteException` exhibits the same semantics and is thus elided.

Consider instruction `0x129c` of Listing 10.2. The result of this instruction is $\sigma'.\,\mathsf{emap} = \varnothing$.

The rules in Figs. 11.1d and 11.1e define `__cxa_begin_catch` behavior for different cases. For an exception not already caught, the associated rule pushes it onto the caught-exception stack. The rule for already-caught exceptions does not do this. However, both rules increment that exception's handler count and decrement the state's count of uncaught exceptions. Though not listed, there is also a rule for an ID not currently in the exception map. That rule operates the same as our (elided) rule for unmodeled external calls. This allows for safe overapproximation.

Consider instruction `0x13a0` of Listing 10.3. Assuming the existence of a valid exception object with ID id that was just thrown, the post-state σ' will satisfy $\mathrm{handler}(id, \sigma') = 1$, $\sigma'.\mathsf{caught} = [id]$, and $\sigma'.\mathsf{uncaught} = 0$.

$$\frac{\sigma'.\texttt{rip} \in \sigma.\{\texttt{rdi}, \texttt{rcx}, \texttt{r8}\} \qquad \text{pushStack}(\sigma.\texttt{rip} + 5, \sigma, \sigma')}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(a) `__libc_start_main`

$$\frac{\sigma'.\texttt{rax} = id \qquad \sigma'.\,\mathsf{emap}(id) = e \qquad \neg e.\mathsf{rethrown} \qquad e.\mathsf{handlerCount} = 0}{\sigma \xrightarrow{\mathsf{A}} \sigma'}\ id = \sigma'.\texttt{rip}$$

(b) `__cxa_allocate_exception`

$$\frac{\sigma'.\,\mathsf{emap}(\sigma.\texttt{rdi}) = \bot}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(c) `__cxa_free_exception`

$$\frac{id \notin \sigma.\mathsf{caught} \qquad \text{handler}(id, \sigma')\oplus \qquad \text{pushCght}(id, \sigma, \sigma') \qquad \sigma'.\mathsf{uncaught}{-}{-}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}\ id = \sigma.\texttt{rdi}$$

(d) `__cxa_begin_catch` (not already caught)

$$\frac{id \in \sigma.\mathsf{caught} \qquad \text{handler}(id, \sigma')\oplus \qquad \sigma'.\mathsf{uncaught}{-}{-}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}\ id = \sigma.\texttt{rdi}$$

(e) `__cxa_begin_catch` (already caught)

$$\frac{\begin{array}{ll}\text{handler}(id, \sigma) = 1 & \text{rethrown}(id, \sigma) \\ \text{handler}(id, \sigma') = 1 & \neg\,\text{rethrown}(id, \sigma')\end{array} \qquad \text{popCaught}(\sigma, \sigma')}{\sigma \xrightarrow{\mathsf{A}} \sigma'}\ id = \text{peek}(\sigma.\mathsf{caught})$$

(f) `__cxa_end_catch` (have caught exception, last handler, rethrown)

$$\frac{\begin{array}{ll}\text{handler}(id, \sigma) = 1 & \neg\,\text{rethrown}(id, \sigma) \\ \text{handler}(id, \sigma') = 1 & \sigma'.\,\mathsf{emap}(id) = \bot\end{array} \qquad \text{popCaught}(\sigma, \sigma')}{\sigma \xrightarrow{\mathsf{A}} \sigma'}\ id = \text{peek}(\sigma.\mathsf{caught})$$

(g) `__cxa_end_catch` (have caught exception, last handler, not rethrown)

Figure 11.1: Non-unwinding abstract transition rules (unchanged state parts mostly elided)

$$\frac{\sigma'.\texttt{rip} = \text{peek}(\sigma.\mathsf{stack}) \qquad \text{popStack}(\sigma, \sigma') \qquad \sigma' \xrightarrow{\mathsf{U}} \sigma''}{\sigma \xrightarrow{\mathsf{U}} \sigma''}$$

(a) Repeating unwinding

$$\frac{\sigma'.\texttt{rip} \in \mathsf{LPT}(\sigma.\texttt{rip})}{\sigma \xrightarrow{\mathsf{U}} \sigma'}$$

(b) Landing pad found

$$\frac{\sigma.\mathsf{stack} = []}{\sigma \xrightarrow{\mathsf{U}} \sigma}$$

(c) No landing pad

Figure 11.2: Unwinding

To complete the above, the rules in Figs. 11.1f and 11.1g define some `__cxa_end_catch` behavior. The first rule applies when an exception ID is available on top of the caught stack, there are no more handlers for the corresponding exception object, and it is being rethrown. In this case, it is popped off the caught stack and no longer treated as being rethrown. The second rule applies when an exception is available, has no more handlers, and is not being rethrown. In that case, it is popped off the caught stack and removed from the exception map. Not shown is the rule for an exception that still has handlers remaining. In that case, its handler count is decremented but no other changes are made. Additionally, the case for an empty $\sigma.\mathsf{caught}$ again operates as an unmodeled external call for the sake of overapproximation.

Consider instruction `0x13b9` of Listing 10.3. Assume the statements in Example 11.9 hold for the pre-state. Then, the post-state σ' for that instruction will satisfy $\sigma'.\,\mathsf{emap}(id) = \bot$ and $\sigma'.\mathsf{caught} = []$.

11.3.2 Unwinding Rules

Figure 11.2 shows the rules for stack unwinding transitions, which utilize notation $\xrightarrow{\mathsf{U}}$ instead of $\xrightarrow{\mathsf{A}}$. In these rules, the stack is repeatedly popped (Fig. 11.2a) until one of two conditions occurs: a landing pad is found (Fig. 11.2b) or the stack is completely unwound (Fig. 11.2c). For shorthand notation, we respectively use $\xRightarrow{\mathsf{U}^+}$ and $\xRightarrow{\mathsf{U}^-}$ to indicate the compound stack unwinding transition from a state until one of those conditions is met.

Example 11.11 (Single unwinding step)**.** Assume $\sigma.\mathsf{stack} = [\texttt{0x116e}]$. Then, for $\sigma \xrightarrow{\mathsf{U}} \sigma'$ to hold, we must have $\sigma'.\mathsf{stack} = \varnothing$ and $\sigma'.\texttt{rip} = \texttt{0x116e}$.

Figures 11.3a and 11.3b show the simplest unwinding function rules, those for the function `_Unwind_Resume`. The main addition to the general unwinding transition is that, when landing pads are found, the original function argument ($\sigma.\texttt{rdi}$) is preserved in the result state's return register ($\sigma'.\texttt{rax}$). This models the concrete handling for carrying through exceptions during unwinding.

Consider instruction 0x1398 of Listing 10.3. As previously described in Section 10.3, this instruction is intended to continue unwinding for exceptions that do not satisfy the source code's catch type specification. Assuming no more applicable landing pad table entries, the only valid post-states for the transition here satisfy $\sigma'.\mathsf{stack} = []$ and $\sigma'.\mathsf{terminated} = \mathtt{Bad}$.

The rules for the initiating function `__cxa_throw`, shown in Figs. 11.3c and 11.3d, expand on those for `_Unwind_Resume`. They add the condition that the post-state's uncaught exception count is incremented. At our level of abstraction, the function `_Unwind_RaiseException` is semantically equivalent to `__cxa_throw` and thus shares its rules.

Consider instruction 0x1286 of Listing 10.2. We previously stepped through the process of throwing using this instruction in Section 10.3, so we merely state the results here. As this is the first throw at this time, we have $\sigma'.\mathsf{uncaught} = 1$. Additionally, the unwinding process stops for $\sigma'.\mathtt{rip} \in \mathsf{LPT}(\mathtt{0x137e}) = \{\mathtt{0x138b}\}$, giving us $\sigma'.\mathtt{rip} = \mathtt{0x138b}$.

The rules for `__cxa_rethrow` in Figs. 11.3e and 11.3f add a twist by utilizing the current caught-exception stack. When an exception object ID is available on the top of the caught stack, unwinding proceeds as usual. Futhermore, the corresponding exception object is marked as being rethrown and its ID is stored in `rax` for later usage. By contrast, when no caught exception objects are available, `__cxa_rethrow` must lead to an abnormal termination for strict modeling. However, that second rule can be relaxed for additional overapproximation by using the `_Unwind_Resume` rules instead.

Consider instruction 0x13af of Listing 10.3. Assume a caught stack $\sigma.\mathsf{caught} = [id]$, an exception object $\sigma.\mathsf{emap}(id) = e$, and landing pad table $\mathsf{LPT}(\mathtt{0x13af}) = \{\mathtt{0x13c0}\}$. Given those conditions, we end up with $\sigma'.\mathtt{rip} = \mathtt{0x13c0}$ and $\sigma'.\mathsf{emap}(id).\mathsf{rethrown}$.

Additional rules exist for the process of *forced unwinding*, or manual stack unwinding. Those are summarized here. `_Unwind_ForcedUnwind` functions similarly to `__cxa_throw` (Figs. 11.3c and 11.3d). However, instead of stopping based on landing pad table information, it executes the function stored in $\sigma.\mathtt{rsi}$ in each frame and uses the result to determine when to stop. `_Unwind_DeleteException` functions like `cxa_free_exception` (Fig. 11.1c) at the end of that process. The helper function `_Unwind_GetIP` stores the current frame's instruction pointer in $\sigma'.\mathtt{rax}$. Finally, the other helper function `_Unwind_GetRegionStart` stores the current procedure fragment's starting address in $\sigma'.\mathtt{rax}$.

11.4 Symbolic Execution

We perform symbolic execution by application of the rules making up our abstract transition relation. For some initial abstract state σ_0, $\sigma_0.\mathtt{rip}$ is either manually provided or obtained from the binary's ELF info. Then we iteratively fetch the instruction at that address, increment `rip` appropriately, and apply the applicable abstract transition rule to obtain successor states. If the transition rule results in multiple possible continuing states, we apply the symbolic execution step to each successor state. If no non-terminating states result, this path of execution ends.

$$\frac{\sigma \overset{\mathsf{U}^+}{\Longrightarrow} \sigma' \qquad \sigma'.\texttt{rax} = \sigma.\texttt{rdi}}{\sigma \overset{\mathsf{A}}{\to} \sigma'}$$

(a) `_Unwind_Resume` (have landing pad(s))

$$\frac{\sigma \overset{\mathsf{U}^-}{\Longrightarrow} \sigma' \qquad \sigma'.\mathsf{terminated} = \mathsf{Bad}}{\sigma \overset{\mathsf{A}}{\to} \sigma'}$$

(b) `_Unwind_Resume` (no landing pads)

$$\frac{\sigma \overset{\mathsf{U}^+}{\Longrightarrow} \sigma' \qquad \sigma'.\mathsf{uncaught}{+}{+} \qquad \sigma'.\texttt{rax} = \sigma.\texttt{rdi}}{\sigma \overset{\mathsf{A}}{\to} \sigma'}$$

(c) `__cxa_throw` (landing pad(s))

$$\frac{\sigma \overset{\mathsf{U}^-}{\Longrightarrow} \sigma' \qquad \sigma'.\mathsf{terminated} = \mathsf{Bad}}{\sigma \overset{\mathsf{A}}{\to} \sigma'}$$

(d) `__cxa_throw` (no landing pads)

$$\frac{\sigma \overset{\mathsf{U}^+}{\Longrightarrow} \sigma' \qquad \begin{array}{l}\sigma'.\mathsf{uncaught}{+}{+}\\ \mathrm{handler}(id, \sigma')\ominus\end{array} \qquad \mathrm{rethrown}(id, \sigma') \qquad \sigma'.\texttt{rax} = id}{\sigma \overset{\mathsf{A}}{\to} \sigma'} \; id = \mathrm{peek}(\sigma.\mathsf{caught})$$

(e) `__cxa_rethrow` (caught exception+landing pad(s))

$$\frac{\sigma.\mathsf{caught} = [] \qquad \sigma'.\mathsf{terminated} = \mathsf{Bad}}{\sigma \overset{\mathsf{A}}{\to} \sigma'}$$

(f) `__cxa_rethrow` without a caught exception

Figure 11.3: Abstract transition rules involving unwinding (unchanged state parts are elided)

To prevent infinite loops and alleviate some of the state space issues that can occur with such non-deterministic evaluation, we provide a join operation. This join operation is focused on exceptional state. From Σ it preserves emap, uncaught, and caught. To maintain contextual awareness, it also preserves `rip`, stack, and terminated. As an implementation detail, it also includes the temporary indices used by our jump table heuristic to ensure proper separation. All other state parts are combined, with priority given to the first equivalent state produced. For a more aggressive join, emap, uncaught, and caught can be excluded from the preserved state parts. The abstract transition rules are also simplified to support this exclusion.

11.5 Argument for Overapproximation

We consider a formal definition of the concrete transition rules our abstract ones overapproximate outside the scope of this dissertation. This is because our abstraction focuses on the domain of exceptional control flow in terms of its ABI-level definition. By contrast, concrete rules require a concrete implementation. Instead, we provide an informal argument for why our abstract transition rules are overapproximative.

First, for normal (non-exception-related) assembly instructions, our abstract transition rules default to assigning $\bot$ to destination operands, overapproximating their effect. Only those instructions whose arguments affect exception- and stack-related behavior as well as global memory operations receive full modeling. They include `mov` and its relatives, `push`+`pop` and related instructions, and basic arithmetic/bitwise instructions. If we did not model those

instructions, we could lose too much information concerning exceptional or even regular control flow.

Second, for exception-related function calls, the semantics in Figs. 11.1 and 11.3 purposefully omit information from the abstract state. An example is the type of the exception being allocated. The abstract step function, then, considers *all* possible next states for *any* exception type.

Furthermore, not all indirections are resolvable. In these cases, we do not apply additional heuristics or guesses. Instead, we stop further exploration at the indirection, if a jump, and clearly annotate the output accordingly. Unresolved indirect calls are treated as unmodeled external calls, but the same principle applies. We thus informally argue that the produced EICFG is overapproximative *modulo* unresolved indirections. If the EICFG is not annotated with any unresolved indirections, it is an overapproximation.

11.6 Graphs

We have a specific node state for the EICFGs produced by our toolchain, previously described in Section 10.2. The graph node type $\mathbb{N}$ contains the following information: The current program counter, a list of the return addresses for all current stack frames, the exception objects currently allocated, the number of uncaught exceptions, and the IDs of those exception objects currently caught. The current jump table index and termination state will also included in this if they exist. In notation, this is $\mathbb{N}$: $\mathbb{P} \times [\mathbb{P}] \times [\mathbb{E}] \times \mathbb{N} \times [\mathbb{P}] \times (\mathbb{W}|\bot) \times \mathbb{T}$. We further have a function α': $\mathbb{\Sigma} \to \mathbb{N}$ that maps from the more specific and detailed abstract states for execution to the exceptional state for control flow representation.

Definition 11.15 (Abstract state to exceptional state)**.**

$$\alpha'(r, s, \_, e, u, c, j, t, \_) = (\text{rip}(r), \text{retAddrs}(s), e, u, c, j, t)$$

Where rip gets the current `rip` from a register map and retAddrs extracts the list of return addresses from a stack.

Example 11.16 (Graph node details)**.** Again consider `0x135d` of Listing 10.4. The precondition for this instruction converts to

$$(\texttt{0x133d}, [\texttt{0x114e}], [], 0, [], \bot, \bot),$$

which is what is shown in the EICFG in full-detail mode. In basic block mode, performed via postprocessing of the EICFG, we can have a range of addresses instead of a single address for the node state. For this specific case, the basic block node representation is

$$([\texttt{0x1325}, \texttt{0x133d}], [\texttt{0x114e}], [], 0, [], \bot, \bot).$$

The EICFGs themselves are represented in a slightly different form from that described in Section 10.2. They consist of a set of nodes and a set of annotated edges that connect those nodes: $\mathbb{G}$: $\{\mathbb{N}\} \times \{\mathbb{N} \times \mathbb{N} \times (\texttt{Inst}|\bot)\}$. The edge annotations are instructions for edges that follow standard control flow.

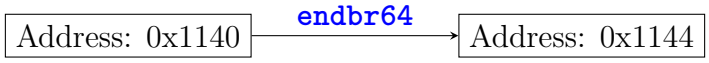


Figure 11.4: Simple graph

Example 11.17 (Full graph)**.** Consider the simple graph shown in Fig. 11.4. This graph is the graph you would get after analyzing Line 1 of Listing 10.5. Mathematically, this graph has nodes

$$n_0 = (\texttt{0x1140}, [], [], 0, [], \bot, \bot)$$
$$n_1 = (\texttt{0x1144}, [], [], 0, [], \bot, \bot)$$

and edge $(n_0, n_1, \texttt{endbr64})$.

Chapter 12

Validation and Results

This chapter wraps up Part IV with coverage of the parts of our tool that we validated, the results from our case study, and a discussion of some challenges.

12.1 Validation

To increase trustworthiness, we validated some of our abstract transition rules against the corresponding real-world implementations. Specifically, we generated abstract states σ, and validated that:

$$\sigma \xrightarrow{\mathsf{A}} \sigma' \wedge \gamma(\sigma) \xrightarrow{\mathsf{C}} s' \implies \alpha(s') = \sigma' \tag{12.1}$$

Here α and γ denote abstraction and concretization functions, and $\xrightarrow{\mathsf{C}}$ denotes concrete execution.

Remark 12.1 (Why rule validation instead of full verification?). This process was used instead of a more formal verification approach as there are no formal specifications for the functions under examination. Thus, the closest we can get to establishing soundness of our abstract transition rules is to compare the results of those rules against concrete executions. Because of this, we did not do a full top-to-bottom verification effort as was done for HGs.

Abstract states σ are obtained through via *fuzzing* [40]. For each rule under validation, we generated 10 000 arbitrary initial abstract states (σ) and then applied the rule to obtain the corresponding abstract post states (σ'). Then, using a test harness implemented as a combination of Python and GNU Project debugger (GDB) scripts, we ran constructed real-world binaries featuring the desired concrete functions. The usage of GDB allowed easy interceding at specific points in the binaries in order to set up the initial state and extract the state after the step. Function γ operates before the concrete library function is executed, setting the state parts in Table 12.1 to their test case values. Function α operates after the concrete library function is executed, extracting the listed state parts from the concrete program state. The test harness then verifies that the abstracted state parts match the expected ones generated previously, satisfying Eq. (12.1). Table 12.1 shows our validation status.

Table 12.1: Validated state parts

Rule	rip	in/out regs	handlerCount	uncaught	hndlrSwitchVal	caught
__cxa_throw	✓	✓	✓	✓		
__cxa_begin_catch	✓	✓	✓	✓	✓	
__cxa_end_catch	✓	n/a	✓	✓	✓	Part.
__cxa_rethrow	✓	✓	✓	✓		Part.
_Unwind_Resume		✓	✓	✓	✓	

12.1.1 Test Programs

Our constructed test programs are designed to be minimal but still call the specific library functions we provided abstract transition rules for. To easily read and write memory using GDB, we provided dummy versions of certain structs. Specifically, the hidden library structs for individual exception objects as well as global exception information. To utilize those structs within GDB, the programs must be built in debug mode.

12.1.2 Abstract State Generation

Next, the abstract start and end states are generated by a small wrapper around our abstract transition rules. We used components of the property testing library QuickCheck [40] to instrument the start state generation. The initial starting addresses are defined by the binary versions of the above-mentioned test programs. The end state generation is performed by applying a single step of our methodology to those test programs using those start states. The start states as well as the end states for each step are then exported for use by the test harness.

12.1.3 Concretization and Abstraction

The concretization and abstraction functions γ and α are part of that test harness. As in abstract interpretation, they interface between the generated abstract states and the concrete memory layouts mentioned above. Both functions operate via GDB breakpoints that are set depending on the rule under test. γ operates before the concrete library function is executed, setting the state parts in Table 12.1 to their test case values. α operates after the concrete library function is executed, extracting the listed state parts from the concrete program state after the library function is executed. The test harness then verifies that the abstracted end state parts match the expected ones generated previously, satisfying Eq. (12.1).

12.2 Results

Here we present the results of generating EICFGs for 341 real-world programs and libraries. These programs and libraries have a variety of sizes and use cases and were sourced from places like GitHub and the Advanced Package Tool (APT) repositories. 49 of these programs utilize C++ EH (several despite the lack of EH tables) while all have been compiled for the

Table 12.2: Case study results.

	Absolute Numbers						Baseline Comparison
Groups	Binary Count	Covered Insts	Unwind Edges	Unique Throws	Caught Throws	Time/s	Inst Diff
NASA	13/14	1 741 089	1410	167	136	8276	1027
Xen	85/90	231 880	0	0	0	32 200	0
Magick	15/17	172 811	14	14	2	129	15
Cups	163/164	317 137	3938	33	0	6811	−9
Other	18/23	763 063	63 260	830	526	9324	11 766
caf	5/6	632 127	7952	466	254	1767	4251
art	1/4	57 866	216	31	30	264	1885
audio	5/7	100 207	49 199	523	380	1153	8509
drives	3/3	5470	21 538	31	30	411	353
games	4/7	117 375	735 224	519	473	19 560	5551
science	1/1	23 804	3567	94	92	80	3009
tasking	1/1	23 977	6	4	3	646	119
torrent	2/5	512 095	383 180	600	399	25 122	8235
Totals	316/341	4 715 806	1 269 649	3350	2356	105 809	45 032

x86-64 ISA and the System V ABI. The EICFG generation for each binary was executed on a server with four Intel® Xeon® E7-8890v4 CPUs (for a total of 96 cores) running at 2.20 GHz with 252 GiB of RAM. The server's OS was Ubuntu 18.04.5 long-term support (LTS). Execution timeout was set to eight hours. Some of the numbers were collected with the assistance of GNU parallel [170].

A summary of the results comparing EICFG generation to our tool with EH disabled can be found in Table 12.2. That baseline version functions as our regular tool but without the unwinding-related abstract transition rules from Figs. 11.1 to 11.3. Specifically, the throw-related functions (such as `__cxa_throw`) were treated as terminating functions while the catch-related functions (such as `__cxa_begin_catch`) were treated as no-ops. 25 of the binaries we analyzed are not included in the table as the tool ran out of of memory or timed out due to state space explosion, in one case just in the baseline version.

We identified 3350 unique throws and traced the exceptional control flow of each one. Based on our analysis, we were able to identify 994 of them as uncaught; the remaining 2356 all had a potential catch block in their unwinding path. On average, dealing with exceptional control flow can increase coverage by 13 instructions per unique throw, with each throw averaging 379 unwind edges. Those edges are ones tools such as Ghidra do not produce. Note the Xen binaries exhibited no change, as none contained any exceptional control flow.

12.2.1 Unsound Heuristics

For fully sound, overapproximative analysis, EICFG generation must start from the address recorded in the ELF entry field and not consider any other entry points. However, this can have some practical limitations if unresolved indirect branches or external functions that take callbacks are present. Thus, we provide some additional features that can be activated if so desired.

First, we check the binary's list of function symbols, if available, and use each one as a further entry point if not already in the graph. Note that *we still maintain soundness within those function call subgraphs*. While not all contextual information is present, any unwinding originating within such a subgraph will propagate properly up to the point of the function being treated as an entry point.

Additional coverage was provided by modeling those external functions we could identify as taking direct function pointers as callbacks. This was done by treating the external functions as calls to those callbacks instead. When external functions took multiple callbacks, we used nondeterminism to model each call. Thread-spawning functions such as `pthread_create` are included here, though we do not model their concurrent behavior.

12.2.2 Indirection

Though not shown in Table 12.2, unresolved calls and jumps are primarily due to symbolic callbacks and other passed-around jump targets that cannot be concretized after our abstraction. However, many instances of indirection *were* resolvable.Those cases rely on a nondeterministic heuristic for basic jump table calculations, which is documented in Section 10.3 for the example EICFG and corresponding code snippet in Fig. 10.3 and Listing 10.4.

Because not all calculations involving a static upper bound are jump table calculations, this heuristic can be tuned in order to control state space expansion. When the EICFG generation algorithm is supplied with a specific jump table upper bound (JTUB), predicted jump table index locations with bounds greater than that JTUB will fall back to normal state read handling. This does not affect our overapproximation, as it results in annotated EICFGs, but may reduce coverage. For the specific results displayed in Table 12.2, we utilized a JTUB of 25. Our results showed that using that limit did not significantly reduce reported instruction coverage and did increase binary coverage overall.

12.2.3 State Space Reduction

As our approach is context-sensitive where possible, this resulted in significantly more EICFG nodes than instructions. To reduce state space explosion and performance issues we encountered, we implemented some state space reduction measures.

For the first such measure, our analysis tool has an option to reduce the exceptional state space by simplifying the rules in Figs. 11.1 and 11.3. Specifically, it removes usage of emap, caught, and uncaught, instead assuming the existence of correct behavior regarding those fields. This did not reduce instruction coverage in any of our successful tests and allowed more

analyses to complete without timing out or running out of memory. In fact, for some programs, such as `transmission-edit` and `AntSimulator`, it slightly increased coverage.

A further reduction technique is to restrict assumed JTUBs. Cases with bounds greater than a user-supplied value can be instructed to fall back to normal rather than JTUB handling. This is available as using an unbounded upper bound can result in a significant number of unnecessarily-generated graph nodes.

12.3 Challenges

The process of developing EICFGs was not without difficulty. Here are some of the challenges we encountered.

12.3.1 For Validation

During the process of validation, we uncovered several implementation quirks that were not obvious. For example, the field handlerCount is actually a signed integer. This means that, when generating initial states, a negative value may be produced. As it turns out, the concrete implementation of `__cxa_begin_catch` takes the absolute value of negative handler counts supplied to it before incrementing that value. `__cxa_rethrow` performs a similar, but stranger, transformation. It decreases the magnitude by one, then inverts the sign; if the magnitude is 0, handlerCount is unchanged. The implementation of our abstract transition rules was updated to reflect those unearthed quirks.

Additionally, some issues arose when constructing arbitrary concrete states. For example, creating arbitrary exception objects requires explicitly allocating memory, as modern real-world programs have memory protection and do not allow accessing unallocated memory locations. Thus, we did not perform fuzzing with the `exceptionType` field, nor did we do a full analysis of the `caught` linked list and the necessary `nextException` field. Rethrown status is not dealt with here as well as in concrete implementations it is not explicitly part of the exception object header struct or the global exceptional state.

We also did not cover those cases where an exception does not get caught and results in program termination due to stack unwinding. This is because we were unable to easily check the desired state parts in such cases. Similarly, we could not validate the `rip` modification of `_Unwind_Resume`. When running in GDB with our test program constructed to utilize `_Unwind_Resume`, handler switch value manipulation is required to trigger that path. That in itself is not necessarily an issue, but when that path is taken, control flow is ultimately redirected to the landing pad for the catch block that leads to the `_Unwind_Resume` rather than an appropriate parent landing pad. This prevents us from validating the target landing pad (or lack thereof) for that function (i.e. σ'.`rip`). However, we were still able to validate the other exceptional state components manipulated by that function.

12.3.2 For Integration Concerns

During comparative analysis, we discovered that termination states provided excess overapproximation. Specifically, two of the cups binaries, which both throw exceptions, have slightly more covered instructions in the baseline (`pdftohtml` and `pdftocairo`). This appears to be related to the usage of specific C++ functions that wrap exception throwing such as `std::__throw_logic_error(char const*)`, which are treated as terminating in the baseline. The "missing" instructions are the ones immediately following those calls. This indicates that the instruction after a terminating function call is included in the set of covered instructions, a minor flaw of our overapproximation. The specific cause is within the implementation, an interaction of when `rip` is incremented during step function execution with how termination conditions are handled.

A simple fix for this issue is to post-process the final EICFG and remove from the instruction coverage count the states that have unique `rip` combined with termination conditions. **Further such issues could be avoided or identified by doing verification of various example EICFGs. However, that would require assuming the correctness of our abstract transition rules, as our validation provides high assurance of their correctness but does not provide a formal proof of such. Furthermore, it would also require manual analysis of the programs under test in order to determine the expected EICFGs, as we again do not have a full formal specification to model and produce expected results with.**

12.4 Summary

This part covered the generation of EICFGs. These EICFGs contain nodes and edges for exceptional control flow. Such nodes and edges are not produced by COTS tools such as Ghidra [134].

The EICFG generation algorithm was formulated using abstract transition rules that model the functions from the C++ EH ABI [32]. Those rules are utilized by a step function that performs the process of disassembly, modeling any other relevant instructions as well. The algorithm itself was informally proven sound using pen-and-paper proofs.

For increased assurance, a subset of the abstract transition rules were subsequently validated using fuzzing and comparative analysis with 10 000 concrete executions of the corresponding functions. We also performed a real-world case study on 341 off-the-shelf binaries compiled from C++, C, and Fortran source code. Our tool was able to successfully execute on 316 of those binaries. Furthermore, it was able to identify 3350 unique throws and successfully trace the exceptional control flow for every one of them. On average, dealing with exceptional control flow can increase coverage by 13 instructions per unique throw, with each throw averaging 379 unwind edges.

Part V

Epilogue

Chapter 13

Conclusions

Certain properties, such as memory usage and control flow-related ones, can only be proven on the assembly level. This is due to

1. memory usage requiring a concrete representation of memory and
2. compiled programs having more options for control flow than are present in the source language.

Unfortunately, assembly-level verification, or analysis in general, is a fundamentally harder problem than source-level analysis due to the low level of abstraction. However, it can also produce highly reliable claims over software. Furthermore, 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 set of properties mentioned above cannot be determined automatically under all circumstances. They are *undecidable* properties. Because of this and the overheads that ITP can have, we designed

1. *semi-automated* methodologies for memory usage and later
2. non-ITP *fully automated* methodologies for targeted control flow analyses.

Those methodologies are briefly revisited below.

13.1 Contributions Revisited

This dissertation presented two methods for proving memory usage, Floyd-style verification and Hoare-style verification. Both approaches rely on the same symbolic execution model and memory-related rewrite rules documented in Chapter 3, but differ in several major aspects.

Parts III and IV then each provided a method for the lifting of control flow from binaries. The former aims for general control flow in the absence of structured exceptions, while the

latter is focused specifically on exception-related control flow. Neither can resolve all indirect branches, however.

13.1.1 Floyd-Style Verification

This methodology uses a Floyd-style approach [67] with automatically-selected cutpoints. It is very similar to the work of Matthews et al. [121], but with a focus on memory usage specifically. The rewrite rules over memory accesses from Section 3.2.2 result in additional VCs that would not be present in Matthews' framework. Those VCs would require time-consuming word arithmetic if the appropriate preconditions/assumptions were not present. The preconditions/assumptions simply establish separation and enclosure relations for the necessary memory regions.

As a case study, we applied the methodology to 63 functions extracted from the HermitCore unikernel library. Each function was compiled without optimizations, but for 12 we also targeted the optimized versions. In total, more than 2379 assembly instructions were verified in this way.

13.1.2 Hoare-Style Verification

Rather than using a more general CFG to guide the verification, this methodology extracts more structured SCFs from the function(s) under test. Such an SCF is used as one of the proof ingredients for a generated FMUC. The other proof ingredients are the generated memory regions, MRRs, and block conditions. With the invariant generation as it currently is, user interaction is minimal. Under normal circumstances, users only need to weaken the condition for a loop entry block by merging it with the conditions of all of the loop's exit blocks.

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

13.1.3 Hoare Graphs

This contribution provides the first *provably overapproximative* lifting mechanism for x86-64 binaries, with the closest similar work being Jakstab [101–103]. Any overapproximative representation of a binary must *all* of its behaviors. Those are not just the "normal" behaviors intended by the binary's programmers. They also include all of the "weird" behaviors that may arise due to the conversion from source code to machine code. As an example, any potential stack overflow that could overwrite a return address must be represented in the end product, even if not intended by the programmers or even not visible in the source code. This also applies to control flow that could change depending on whether or not two pointers alias. To achieve that goal, the method here takes a potentially-stripped binary as input, with no debugging information or address labeling required. It produces an HG that contains:

1. the assembly instructions found in the binary;

2. the binary's control flow; and
3. evidence: specifically, inductive invariants that have sufficient information to prove soundness.

Our approach can deal with overlapping instructions and aims at providing overapproximative bounds to indirect branches (such as when a `jmp` is based on a computation instead of a constant). Where necessary, unsoundness annotations are used to indicate possible issues. Additionally, calls to external functions require explicit assumptions represented as proof obligations. Full success occurs when our technique completes and the proof obligations are discharged (proven true). Such cases indicate that, under those assumptions, the lifted representation is a provable overapproximation of the binary.

We have applied our approach to most of the binaries and shared objects of the Xen hypervisor, covering 399 771 instructions in total. This case study shows that our methodology is scalable and applicable to COTS software written without verification in mind. Though not my work, HGs can then be exported to the Isabelle/HOL theorem prover, where they can be formally verified. This second step essentially validates any inference made by the algorithms during step one.

13.1.4 Exceptional Interprocedural Control Flow Graphs

Many C++ programs exhibit exceptional control flow that standard CFG extraction tools in disassemblers and decompilers do not identify. To deal with that issue, we have provided EICFGs and a tool for generating them. EICFGs extend and narrow the focus of standard CFGs extracted from binaries. They do this by including nodes and edges for exceptional control flow while limiting the abstract state to only that domain required for exceptional info.

Our abstract transition relation for exceptional control flow has been informally shown to overapproximate the concrete versions of those edges. This was achieved by fuzzing many of the individual functions of the C++ EH ABI. The functions tested were each executed with 10 000 test cases generated with arbitrary exceptional state. This approach also helped identify edge cases that were not specified in the C++ EH ABI [32].

Furthermore, we have applied our EICFG generator to 341 real-world programs and libraries. We identified 3350 unique throws and were able to trace each one's exceptional control flow: 2356 were potentially caught while 994 had no identified potential for being caught. On average, dealing with exceptional control flow can increase coverage by 13 instructions per unique throw, with each throw averaging 379 unwind edges. Those edges are ones tools such as Ghidra do not produce.

13.2 Future Work

There are a multitude of ways in which the contributions of this dissertation could be built upon. Below are some of them.

13.2.1 Invariant Strength

As a formal property, memory usage has been proven to never miss any memory regions written to, assuming the correctness of the semantics and model it is applied to [22, 179]. 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.

Futhermore, in order to enhance automation, we currently generate very weak invariants. This means that our overapproximations may become *very* overapproximative. While such an approach still works reasonably well, being able to generate stronger invariants would allow decreasing overapproximation. In other words, stronger invariants mean stricter memory usage proofs.

One way to do this would be to implement some form of abstract interpretation. As discussed, it has shown great success in the field of sound binary analysis Section 2.3.1. It was even used as inspiration for some of the contributions here (Parts III and IV). However, abstract interpretation is formulated around domains. Determining the best domains to use for such an approach would be a challenge. In other words, what is the best way to maintain precision for the memory properties we want to state while approximating the parts we do not care about? That determination has yet to be made.

13.2.2 Memory Model Realism

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. This is because 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 itself would likely not be a good idea, however, as we are currently focusing on userspace analysis. This means that all of the complexity of page mapping, swap spaces, and the like are hidden from the programs we analyze anyway.

Another restriction that would be interesting to model would be the practical restriction of addresses to their lower 48 bits for actual addressing (or 57 bit with extensions [1]). This could potentially reduce the state spaces of all of the contributions in this dissertation, as it would be a global restriction on memory operations. However, it could also complicate things instead. That is because dereferenced addresses must be *canonical*: they must have their

remaining 16 bits be equal in value to their 48th bit. Usage of non-canonical addresses results in general protection faults (GPFs) [93]. We would thus have to add another termination condition to our symbolic execution engines. If achieved efficiently, however, it would more closely model concrete execution and potentially uncover further weird behavior.

13.2.3 Dealing with Contextual Information

Moreover, we find that the context-free nature of our HG approach in Part III limits the number of function callbacks that are properly dealt with. Passing around immediates between functions as done in Part IV does not significantly increase support for function callbacks either. This is because the EICFG work currently requires the significant manual effort of building up a list of functions that take callbacks and their argument lists. That choice was made as treating all possible immediate-value function arguments as potential callbacks, even just ones within text section range, would cause too much overapproximation. Even then, with an abstract model that involves overapproximation, non-immediate values are sometimes passed to external or internal calls. One can check all not-known-to-be-executed function symbols in a binary, as also done for the EICFG work, but that is unsound due to analyzing the functions out of context.

The situation becomes even worse when stripped programs are involved, even ones that do not perform indirect calls or jumps. Such scenarios mean there are no function symbols available to check! Additional heuristics for identifying possible function entry points [10, 141] may help with this drawback, however.

Of course, unresolved jumps are still an issue as well, even in non-stripped binaries. One solution is better jump table heuristics [3, 39, 66, 75]. This may result in less false positives/negatives for jump table calculations without the need for manual tuning. Doing so would reduce excess nondeterminism and overapproximation without the need for explicit state space reduction techniques.

13.2.4 Concurrency, Interrupts, and Signals

None of the contributions of this document fully support analysis of concurrent code. While Part IV analyzes the callbacks passed to functions such as `pthread_create`, it does so by treating them as being called at that point. This means that even that contribution has no concept of multiple threads.

Therefore, a significant improvement or future work would be support for concurrency. At the same time, doing so would contribute immensely to the state spaces being explored. It would also significantly increase any proof efforts, as many formal approaches to concurrent code verification require either detailed assumptions/proofs of *non-interference* [139] or explicit resource allocation [191].

None of the contributions here provide support for dealing with interrupts or signal handling either. Semantics are not provided for interrupt-triggering instructions, which are instead treated as no-ops if they are included at all. While in principle simpler than modeling concurrency, this would still add additional complexity. Interrupts can be in both enabled

and disabled states, and handlers can be changed. Both active interrupt handlers and signal handlers can be triggered at any point in a program when set. Neither of our CFG-lifting approaches would be able to support state spaces of such sizes. They would require further enhancement of their state space reduction techniques, which leads into the next section.

13.2.5 State Space Reduction

Additionally, it is, in theory, possible to craft a program that would result in state explosion via sufficiently complex indirect control flow. This is due to the overapproximation and prevention of early joining we do in order to deal with jump tables. The EICFG work in particular suffers from such issues. While we were able to target programs with over 400 000 instructions, our EICFGs generally do not scale far beyond that. Even for smaller programs, we experienced timeouts and out-of-memory cases when a significant number of control flow nodes and edges were generated.

Methods of reducing the state space while maintaining interprocedural exceptional analysis would provide for increased scalability and the ability to target even larger programs. For example, modeling of exception type info and integrating it into the LPT determinations would allow pruning of dead branches, reducing the EICFG tool's overapproximation.

Better pointer inference support would be beneficial as well. By that we mean identifying which registers/memory locations hold pointers and what types of memory they point to, similar to Jakstab's handling. This could allow for reduction of infeasible memory models.

13.2.6 Complexity and General Scalability

We have already discussed ways of increasing scalability by reducing nondeterminism and the like. However, predicting the asymptotic performance of the contributions in this dissertation is not a trivial task in itself. While the analysis of HG generation timing with respect to instruction in Fig. 8.1 shows a generally linear scaling, the presence of outliers indicates it is not so simple. We did not provide a similar distribution for EICFGs generation, but its performance was similar.

Furthermore, the Hoare-style verification and HG works rely on SMT solvers. They are used to solve problems expressed in first-order logic to determine if memory regions necessarily alias, are separate, or are enclosed. Thus, the computational complexity of our algorithms are tied to the complexity of those logical expressions. The impact of those region calculations and specific analyses of nondeterministic growth were judged out of scope of this dissertation, but would be useful for future work.

13.2.7 Integrating CFR into Formal Frameworks

Provably sound binary lifting can be the base for *any* trustworthy binary-level technique, including decompilation, binary verification and binary patching. However, while the HGs

we generate can be validated in Isabelle, this validation is a manual approach. Furthermore, the validation done for EICFGs applies to individual steps and, while done as a concrete analysis, does not integrate with formal tools.

Therefore, in order to increase reliability and trustworthiness of Parts III and IV, full integration with an ITP approach is desirable. For the HG work, this would start off with increasing automation for the loading of HGs into a theorem prover and discharging the proofs. For the EICFG work, this would involve providing an ITP infrastructure for validating the work as a whole given the validation of individual steps.

Both contributions would also benefit from a full formal implementation. That, however, would require the most work of all. Development of a rigorous, properly formal model is not a trivial task. This is the case even with access to mechanized proofs. As such, it would likely best be accomplished by breaking the task down into individual, reusable components. This may involve the development of a formal library for semantics that is specifically intended to lift machine code or assembly to an IR designed for the various tasks presented in this dissertation.

Bibliography

[1] *5-Level Paging and 5-Level EPT*. white paper 335252-001. Version 1.0. Intel Corporation, Dec. 2016.

[2] 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, 4 (Nov. 6, 2009), pp. 1–40. ISSN: 1094-9224. DOI: 10.1145/1609956.1609960.

[3] Xiaoxin An, Freek Verbeek, and Binoy Ravindran. "DSV: Disassembly Soundness Validation without Assuming a Ground Truth". In: *NASA Formal Methods*. Proceedings of the 14th International Symposium (Pasadena, California, USA, May 24–27, 2022). Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Lecture Notes in Computer Science 13260. Berlin Heidelberg: Springer-Verlag, May 20, 2022, pp. 636–655. DOI: 10.1007/978-3-031-06773-0_34.

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

[5] Avast Software. *RetDec :: Home*. URL: https://retdec.com/ (visited on 05/18/2022).

[6] Inokentiy Babushkin. *hapstone: Capstone bindings for Haskell*. June 13, 2016. URL: https://hackage.haskell.org/package/hapstone (visited on 12/21/2022).

[7] Henry G. Baker. "CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A." In: *SIGPLAN Notices* 30.9 (Sept. 1995), pp. 17–20. ISSN: 0362-1340. DOI: 10.1145/214448.214454.

[8] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. "CodeSurfer/x86—A Platform for Analyzing x86 Executables". In: *Compiler Construction*. 14th International Conference (Edinburgh, UK, Apr. 4–8, 2005). Ed. by Rastislav Bodik. Lecture Notes in Computer Science 3443. Berlin Heidelberg: Springer-Verlag, 2005, pp. 250–254. ISBN: 978-3-540-31985-6. DOI: 10.1007/978-3-540-31985-6_19.

[9] Gogul Balakrishnan and Thomas Reps. "Analyzing Memory Accesses in x86 Executables". In: *Compiler Construction* (Barcelona, Spain, Mar. 29–Apr. 2, 2004). Ed. by Evelyn Duesterwald. Lecture Notes in Computer Science 2985. Berlin Heidelberg: Springer-Verlag, 2004, pp. 5–23. ISBN: 978-3-540-24723-4. DOI: 10.1007/978-3-540-24723-4_2.

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. "BYTEWEIGHT: Learning to Recognize Functions in Binary Code". In: *23rd USENIX Security Symposium* (San Diego, CA). 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Aug. 2014, pp. 845–860. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao.

[11] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. "Reliable and Fast DWARF-Based Stack Unwinding". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA, 146 (Oct. 2019). DOI: 10.1145/3360572.

[12] 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.

[13] Mohamed Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. "Speculative Disassembly of Binary Code". In: *Compilers, Architectures, and Sythesis of Embedded Systems*. Proceedings of the 2016 International Conference (Pittsburgh, Pennsylvania, Oct. 1–7, 2016). New York, NY, USA: Association for Computing Machinery, Oct. 1, 2016, 16, pp. 1–10. ISBN: 9781450344821. DOI: 10.1145/2968455.2968505.

[14] 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. DOI: 10.1007/11502760_3.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] Joshua 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.

[21] Joshua A. Bockenek and Freek Verbeek. *jabocken/SSM-Construction: Final Version of PLDI 2022 Artifact*. Version v2.0.0.2. Mar. 5, 2022. DOI: 10.5281/zenodo.6330573.

[22] 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. Lecture Notes in Computer Science 11698. Berlin Heidelberg: Springer-Verlag. DOI: 10.1007/978-3-030-26601-1_3.

[23] Maria Paola Bonacina. "On Theorem Proving for Program Checking: Historical Perspective and Recent Developments". In: *Principles and Practice of Declarative Programming*. Proceedings of the 12th International ACM SIGPLAN Symposium (Hagenberg, Austria, July 26–28, 2010). Ed. by Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández. PPDP '10. New York, NY, USA: Association for Computing Machinery, July 26, 2010, pp. 1–12. ISBN: 9781450301329. DOI: 10.1145/1836089.1836090.

[24] 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.

[25] 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.

[26] Robert S. Boyer and Yuan Yu. "Automated Proofs of Object Code for a Widely Used Microprocessor". In: *Journal of the ACM* 43.1 (Jan. 1, 1996). Ed. by Frank Thomson Leighton, pp. 166–192. DOI: 10.1145/227595.227603.

[27] Alex Bradbury, Gavin Ferris, and Robert Mullins. *Tagged Memory and Minion Cores in the lowRISC SoC*. Memo. Original link no longer works. University of Cambridge, Dec. 2014. URL: https://web.archive.org/web/20220814190715/https://lowrisc.org/downloads/lowRISC-memo-2014-001.pdf (visited on 12/21/2022).

[28] 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: Association for Computing Machinery, Oct. 15, 2009, 5, 5:1–5:9. ISBN: 978-1-60558-700-4. DOI: 10.1145/1631716.1631721.

[29] 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.

[30] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. "Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring". In: *22nd USENIX Security Symposium* (Washington, D.C.). 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Aug. 2013, pp. 353–368. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz.

[31] 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).

[32] *C++ ABI for Itanium: Exception Handling*. Version 1.22. Mar. 14, 2017. URL: https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html (visited on 12/21/2022).

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

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

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

[36] Cristina Cifuentes. "Reverse Compilation Techniques". PhD thesis. Brisbane, Queensland, Australia: Queensland University of Technology, 1994.

[37] Cristina Cifuentes and K. John Gough. "Decompilation of Binary Programs". In: *Software: Practice and Experience* 25.7 (1995), pp. 811–829. DOI: 10.1002/spe.4380250706.

[38] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. "Assembly to High-Level Language Translation". In: *Software Maintenance*. Proceedings of the International Conference (Bethesda, MD, USA, Nov. 20, 1998). Piscataway, NJ, US: IEEE, Aug. 6, 2002, pp. 228–237. DOI: 10.1109/ICSM.1998.738514.

[39] Cristina Cifuentes and Mike Van Emmerik. "Recovery of Jump Table Case Statements from Binary Code". In: *Science of Computer Programming* 40.2 (2001). Special Issue on Program Comprehension, pp. 171–188. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(01)00014-4.

[40] Koen Claessen, Björn Bringert, and Nick Smallbone. *GitHub - nick8325/quickcheck: Automatic testing of Haskell programs*. May 6, 2022. URL: https://github.com/nick8325/quickcheck (visited on 10/26/2022).

[41] Denton Leslie Clutterbuck. "The Validation and Verification of Low-Level Code". PhD thesis. University of Southampton, 1986. URL: https://eprints.soton.ac.uk/460779/.

[42] Denton Leslie 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.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[44] corob-msft, v-kents, mikeblome, Mikejo5000, ghogen, and Saisang. *Using setjmp and longjmp | Microsoft Docs*. Aug. 3, 2021. URL: https://docs.microsoft.com/en-us/cpp/cpp/using-setjmp-longjmp?view=msvc-170 (visited on 05/22/2022).

[45] Patrick Cousot. "Abstract Interpretation". In: *ACM Computing Surveys* 28.2 (June 1, 1996), pp. 324–328. DOI: 10.1145/234528.234740.

[46] 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: Association for Computing Machinery, Jan. 19–19, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[47] 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.

[48] 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. 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Jan. 26, 1998, pp. 63–78. DOI: 10.5555/1267549.1267554.

[49] 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). Ed. by Ahmad-Reza Sadeghi, Virgil Gligor, and Moti Yung. New York, NY, USA: Association for Computing Machinery, Nov. 4, 2013, pp. 223–234. DOI: 10.1145/2508859.2516702.

[50] 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). Ed. by Ahmad-Reza Sadeghi, Frederik Armknecht, and Jean-Pierre Seifert. New York, NY, USA: Association for Computing Machinery, Nov. 4, 2013, pp. 3–12. ISBN: 978-1-4503-2486-1. DOI: 10.1145/2517300.2517302.

[51] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. "Scalable Validation of Binary Lifters". In: *Programming Language Design and Implementation*. Proceedings of the 41st ACM SIGPLAN Conference (London, UK). New York, NY, USA: Association for Computing Machinery, 2020, pp. 655–671. ISBN: 9781450376136. DOI: 10.1145/3385412.3385964.

[52] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks". In: *Scalable Trusted Computing*. Proceedings of the 2009 ACM Workshop (Chicago, Illinois, USA). New York, NY, USA: Association for Computing Machinery, Nov. 13, 2009, pp. 49–54. ISBN: 9781605587882. DOI: 10.1145/1655108.1655117.

[53] Jeremy Dawson. "Isabelle Theories for Machine Words". In: *Automated Verification of Critical Systems*. Proceedings of the Seventh International Workshop (Oxford, UK, Sept. 10–12, 2007). Ed. by Michael Goldsmith and Bill Roscoe. Vol. 250. Electronic Notes in Theoretical Computer Science 1. Amsterdam, The Netherlands: Elsevier Ltd., Sept. 1, 2009, pp. 55–70. DOI: 10.1016/j.entcs.2009.08.005.

[54] 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).

[55] 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.

[56] Artem Dinaburg and Andrew Ruef. "McSema: Static Translation of x86 Instructions to LLVM". In: *ReCon* (Montreal, Canada). 2014. URL: https://recon.cx/2014/slides/McSema.pdf.

[57] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. "Binary Rewriting Without Control Flow Recovery". In: *Programming Language Design and Implementation*. Proceedings of the 41st ACM SIGPLAN Conference. 2020, pp. 151–163. DOI: 10.1145/3385412.3385972.

[58] Thomas F. Dullien. "Weird Machines, Exploitability, and Provable Unexploitability". In: *IEEE Transactions on Emerging Topics in Computing* (2017). DOI: 10.1109/TETC.2017.2785299.

[59] 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).

[60] 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 12/16/2022).

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

[62] 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.

[63] 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 '06 6. New York, NY, USA: Association for Computing Machinery, June 11, 2006, pp. 401–414. DOI: 10.1145/1133981.1134028.

[64] 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.

[65] 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: Association for Computing Machinery, Nov. 16, 1972, pp. 369–377. DOI: 10.1145/1478873.1478920.

[66] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. "Interprocedural Control Flow Reconstruction". In: *Programming Languages and Systems*. Proceedings of the 8th Asian Symposium (Shanghai, China, Nov. 28–Dec. 1, 2010). Ed. by Kazunori Ueda. Lecture Notes in Computer Science 6461. Berlin Heidelberg: Springer-Verlag, 2010, pp. 188–203. ISBN: 978-3-642-17164-2. DOI: 10.1007/978-3-642-17164-2_14.

[67] Robert W. Floyd. "Assigning Meanings to Programs". In: *Mathematical Aspects of Computer Science* 19.1 (1967), pp. 19–32. Reprinted as "Assigning Meanings to Programs". In: *Program Verification*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Studies in Cognitive Systems 14. Dordrecht, The Netherlands: Springer Science+Business Media Dordrecht, 1993, pp. 65–81. DOI: 10.1007/978-94-011-1793-7_4.

[68] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. "SmartDec: Approaching C++ Decompilation". In: *2011 18th Working Conference on Reverse Engineering*. Oct. 2011, pp. 347–356. DOI: 10.1109/WCRE.2011.49.

[69] Grant R. Foudree. ""Regsym": Dataflow Analysis of Linear x86 Code". English. PhD thesis. Iowa State University, 2019, p. 46. ISBN: 9781392741757.

[70] 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. Berlin Heidelberg: Springer-Verlag, Aug. 19, 2015, pp. 187–202. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_12.

[71] 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.

[72] 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, 63 (Jan. 2, 2019), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3290376.

[73] Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. "IDAPro for IoT Malware Analysis?" In: *Cyber Security Experimentation and Test*. 12th USENIX Workshop (Santa Clara, CA). 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Aug. 2019. URL: https://www.usenix.org/conference/cset19/presentation/g.

[74] 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.

[75] Andrei Gedich and Artur Lazdin. "Improved Algorithm for Identification of Switch Tables in Executable Code". In: *Open Innovations Association*. 17th Conference. 2015, pp. 44–49. DOI: 10.1109/FRUCT.2015.7117969.

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

[77] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls". In: *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.

[78] 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.

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

[80] 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.

[81] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. North-Holland, 2014, pp. 135–214. DOI: 10.1016/B978-0-444-51624-4.50004-6.

[82] *Heartbleed Bug*. Synopsys, Inc. June 3, 2020. URL: http://heartbleed.com/ (visited on 12/16/2022).

[83] Matthew S. Hecht and Jeffrey D. Ullman. "Characterizations of Reducible Flow Graphs". In: *Journal of the ACM* 21.3 (July 1974), pp. 367–375. ISSN: 0004-5411. DOI: 10.1145/321832.321835.

[84] 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). Ed. by Chandra Krintz and Emery Berger. New York, NY, USA: Association for Computing Machinery, June 2, 2016, pp. 237–250. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908121.

[85] Hex-Rays SA. *Hex Rays - State-of-the-art binary code analysis solutions*. URL: https://www.hex-rays.com/ida-pro/ (visited on 01/11/2023).

[86] Hex-Rays SA. *Hex-Rays Decompiler*. URL: https://www.hex-rays.com/decompiler/ (visited on 11/18/2022).

[87] Charles Antony Richard Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (Oct. 1, 1969). Ed. by M. Stuart Lynn, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[88] 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.

[89] R. Nigel Horspool and Nenad Marovac. "An approach to the Problem of Detranslation of Computer Programs". In: *The Computer Journal* 23.3 (1980), pp. 223–229. DOI: 10.1093/comjnl/23.3.223.

[90] 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.

[91] Graham Hutton and Joel Wright. "Compiling Exceptions Correctly". In: *Mathematics of Program Construction*. Proceedings of the 7th International Conference (Stirling, Scotland, UK, July 12–14, 2004). Ed. by Dexter Kozen. Lecture Notes in Computer Science 3125. Berlin Heidelberg: Springer-Verlag, 2004, pp. 211–227. ISBN: 978-3-540-27764-4. DOI: 10.1007/978-3-540-27764-4_12.

[92] *ICECCS 2012*. Proceedings of the IEEE 17th International Conference (Paris, France, July 18–20, 2012). Piscataway, NJ, US: IEEE, Sept. 13, 2012. ISBN: 978-2-9541-8100-4.

[93] *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).

[94] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. "EXAMINER: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM". In: *Architectural Support for Programming Languages and Operating Systems*. Proceedings of the 27th ACM International Conference (Lausanne, Switzerland, Feb. 28–Mar. 4, 2022). Ed. by Babak Falsafi, Michael Ferdman, Shan Lu, and Tom Wenisch. New York, NY, USA: Association

for Computing Machinery, Feb. 22, 2022, pp. 846–858. ISBN: 9781450392051. DOI: 10.1145/3503222.3507736.

[95] *Journal of Automated Reasoning* 5.4 (Dec. 1989). ISSN: 1573-0670.

[96] Alexander Kamkin, Alexey Khoroshilov, Artem Kotsynyak, and Pavel Putro. "Deductive Binary Code Verification Against Source-Code-Level Specifications". In: *Tests and Proofs*. Proceedings of the 14th International Conference (Bergen, Norway, June 22–23, 2020). Ed. by Wolfgang Ahrendt and Heike Wehrheim. Lecture Notes in Programming and Software Engineering 12165. Cham: Springer International Publishing, 2020, pp. 43–58. DOI: 10.1007/978-3-030-50995-8_3.

[97] Deborah S. Katz, Jason Ruchti, and Eric Schulte. "Using Recurrent Neural Networks for Decompilation". In: *2018 Software Analysis, Evolution and Reengineering*. Proceedings of the 25th IEEE International Conference (Campobasso, Italy, Mar. 20–23, 2018). Piscataway, NJ, US: IEEE, Apr. 5, 2018, pp. 346–356. DOI: 10.1109/SANER.2018.8330222.

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

[99] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. "Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing". In: *Software Testing and Analysis*. Proceedings of the 28th ACM SIGSOFT International Symposium (Beijing, China, July 15–19, 2019). New York, NY, USA: Association for Computing Machinery, July 10, 2019, pp. 192–203. ISBN: 9781450362245. DOI: 10.1145/3293882.3330552.

[100] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. "RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications". In: *Annual Computer Security Applications Conference*. Proceedings of the 33rd. Dec. 4, 2017, pp. 412–424. DOI: 10.1145/3134600.3134627.

[101] Johannes Kinder. "Static Analysis of x86 Executables". PhD thesis. Darmstadt: Technische Universität, Nov. 2010. URL: http://tuprints.ulb.tu-darmstadt.de/2338/.

[102] Johannes Kinder. "Towards Static Analysis of Virtualization-Obfuscated Binaries". In: *Reverse Engineering*. 2012 19th Working Conference (Kingston, ON, Canada, Oct. 15–18, 2012). Piscataway, NJ, US: IEEE, Dec. 20, 2012, pp. 61–70. DOI: 10.1109/WCRE.2012.16.

[103] Johannes Kinder and Dmitry Kravchenko. "Alternating Control Flow Reconstruction". In: *Verification, Model Checking, and Abstract Interpretation*. Proceedings of the 13th International Conference (Philadelphia, PA, USA, Jan. 22–24, 2012). Ed. by Viktor Kuncak and Andrey Rybalchenko. Lecture Notes in Computer Science 7148. Berlin Heidelberg: Springer-Verlag, 2012, pp. 267–282. ISBN: 978-3-642-27940-9. DOI: 10.1007/978-3-642-27940-9_18.

[104] James C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (July 1, 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

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

[106] 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: Association for Computing Machinery, Oct. 11, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[107] Gerwin Klein, Thomas Arthur Leck Sewell, and Simon Winwood. "Refinement in the Formal Verification of the seL4 Microkernel". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Springer Science+Business Media, LLC, 2010, pp. 323–339. ISBN: 978-1-4419-1539-9. DOI: 10.1007/978-1-4419-1539-9_11.

[108] 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. DOI: 10.1007/978-3-662-54434-1_26.

[109] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. "Automating Mimicry Attacks Using Static Binary Analysis". In: *USENIX Security Symposium*. Vol. 14. 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, 2005, pp. 161–176.

[110] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. "Static Disassembly of Obfuscated Binaries". In: *13th USENIX Security Symposium*. Proceedings (San Diego, CA, USA, Aug. 9–13, 2004). USENIX Association, 2004. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/static-disassembly-obfuscated-binaries.

[111] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. "Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB". In: *Interactive Theorem Proving*. Proceedings of the 9th International Conference (Oxford, UK, July 9–12, 2018). Lecture Notes in Computer Science 10895. Berlin Heidelberg: Springer-Verlag, 2018, pp. 362–369. DOI: 10.1007/978-3-319-94821-8_21.

[112] kumarak. *How McSema Handles C++ Exceptions | Trail of Bits Blog*. Jan. 21, 2019. URL: https://blog.trailofbits.com/2019/01/21/how-mcsema-handles-c-exceptions/ (visited on 05/18/2022).

[113] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. "Authentication in Distributed Systems: Theory and Practice". In: *ACM Transactions on Computer Systems* 10.4 (Nov. 1992), pp. 265–310. ISSN: 0734-2071. DOI: 10.1145/138873.138874.

[114] 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: Association for Computing Machinery, 2016, 4, pp. 1–8. ISBN: 978-1-4503-4387-9. DOI: 10.1145/2931088.2931093.

[115] Donald C. Latham. *Department of Defense Trusted Computer System Evaluation Criteria. Department of Defense Standard*. DoDD 5200.28-STD. Dec. 26, 1985. URL: https://csrc.nist.gov/CSRC/media/Publications/white-paper/1985/12/26/dod-rainbow-series/final/documents/std001.txt.

[116] 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. SEE. Toulouse, France: HAL, Jan. 2016. HAL: hal-01238879.

[117] Zhibo Liu and Shuai Wang. "How Far We Have Come: Testing Decompilation Correctness of C Decompilers". In: *Software Testing and Analysis*. Proceedings of the 29th ACM SIGSOFT International Symposium (Virtual Event, USA). New York, NY, USA: Association for Computing Machinery, 2020, pp. 475–487. ISBN: 9781450380089. DOI: 10.1145/3395363.3397370.

[118] Anil Madhavapeddy and David J. Scott. "Unikernels: The Rise of the Virtual Library Operating System". In: *Communications of the ACM* 57.1 (Jan. 1, 2014). Ed. by Moshe Y. Vardi, pp. 61–69. ISSN: 0001-0782. DOI: 10.1145/2541883.2541895.

[119] Filip Marić. "A Survey of Interactive Theorem Proving". In: *Zbornik Radova* 18.26 (2015). Ed. by Silvia Ghilezan, pp. 173–223. ISSN: 0351-9406. URL: http://elib.mi.sanu.ac.rs/files/journals/zr/26/zrn26p173-223.pdf.

[120] Daniel Matichuk, Toby Murray, and Makarius Wenzel. "Eisbach: A Proof Method Language for Isabelle". In: *Journal of Automated Reasoning* 56.3 (Jan. 18, 2016), pp. 261–282. DOI: 10.1007/s10817-015-9360-2.

[121] 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.

[122] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. "Probabilistic Disassembly". In: *Software Engineering*. 2019 IEEE/ACM 41st International Conference. Piscataway, NJ, US: IEEE, 2019, pp. 1187–1198. DOI: 10.1109/ICSE.2019.00121.

[123] 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.

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

[125] David Mosberger, Dave Watson, Arun Sharma, Jose Flavio Aguilar Paulino, Brian Sumner, Hans Boehm, Matthieu Delahaye, Max Asbock, and Lassi Tuura. *The libunwind project*. URL: https://www.nongnu.org/libunwind/ (visited on 11/18/2022).

[126] 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.

[127] 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.

[128] 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.

[129] Stefan Nagy. *Toward a Best-of-Both-Worlds Binary Disassembler*. Trail of Bits. Jan. 5, 2022. URL: https://blog.trailofbits.com/2022/01/05/toward-a-best-of-both-worlds-binary-disassembler/ (visited on 12/21/2022).

[130] 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: Association for Computing Machinery, Jan. 1, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[131] 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/.

[132] Ben Niu and Gang Tan. "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity". In: *Computer and Communications Security*. Proceedings of the 2014 ACM SIGSAC Conference (Scottsdale, Arizona, USA, Nov. 3–7, 2014). Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. New York, NY, USA: Association for Computing Machinery, Nov. 3, 2014, pp. 1317–1328. ISBN: 9781450329576. DOI: 10.1145/2660267.2660281.

[133] Bernard Nongpoh. *Error While running Jakstab on simple hello world program compiled in Ubuntu 16.04 machine · Issue #9 · jkinder/jakstab · GitHub*. Dec. 11, 2017. URL: https://github.com/jkinder/jakstab/issues/9 (visited on 12/21/2022).

[134] NSA. *Ghidra*. June 30, 2021. URL: https://ghidra-sre.org/ (visited on 05/18/2022).

[135] 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: 10.1007/3-540-44802-0.

[136] I. M. O'Neill, Denton Leslie 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. 18. Amsterdam, The Netherlands: Elsevier Ltd., Nov. 1988, pp. 115–120. DOI: 10.1016/S1474-6670(17)54540-1.

[137] 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: 10.1007/978-3-642-24372-1_34.

[138] 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.

[139] Susan Owicki and David Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach". In: *Communications of the ACM* 19.5 (May 1976), pp. 279–285. ISSN: 0001-0782. DOI: 10.1145/360051.360224.

[140] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. "SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask". In: *Security and Privacy*. 2021 IEEE Symposium. 2021, pp. 833–851. DOI: 10.1109/SP40001.2021.00012.

[141] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. "Probabilistic Naming of Functions in Stripped Binaries". In: *Annual Computer Security Applications Conference* (Austin, USA). New York, NY, USA: Association for Computing Machinery, 2020, pp. 373–385. ISBN: 9781450388580. DOI: 10.1145/3427228.3427265.

[142] Thomas Peterson. "Alternating Control Flow Graph Reconstruction by Combining Constant Propagation and Strided Intervals with Directed Symbolic Execution". PhD thesis. 2019. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-271208.

[143] Thomas Peterson. *GitHub - tpetersonkth/AlternatingControlFlowReconstruction: An extension to the jakstab tool to leverage underapproximation for control flow reconstruction.* Mar. 23, 2020. (Visited on 12/21/2022).

[144] Prakas Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta. "Interprocedural Exception Analysis for C++". In: *Object-Oriented Programming*. Proceedings of the 25th European Conference (Lancaster, UK, July 25–29, 2011). Ed. by Mira Mezini. Lecture Notes in Computer Science 6813. Berlin Heidelberg: Springer-Verlag, 2011, pp. 583–608. ISBN: 978-3-642-22655-7. DOI: 10.1007/978-3-642-22655-7_27.

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

[146] Todd A. Proebsting and Scott A. Watterson. "Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)" In: *Object-Oriented Technologies and Systems*. Third USENIX Conference. 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, 1997. URL: https://www.usenix.org/conference/coots-97/krakatoa-decompilation-java-does-bytecode-reveal-source.

[147] 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).

[148] 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.

[149] 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.

[150] 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). Ed. by Assia Mahboubi and Magnus O. Myreen. New York, NY, USA: Association for Computing Machinery, Jan. 14, 2019, pp. 181–195. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294102.

[151] Roman Rohleder. "Hands-On Ghidra – A Tutorial about the Software Reverse Engineering Framework". In: *Software Protection*. Proceedings of the 3rd ACM Workshop (London, United Kingdom). New York, NY, USA: Association for Computing Machinery, 2019, pp. 77–78. ISBN: 9781450368353. DOI: 10.1145/3338503.3357725.

[152] Radu Rugina and Martin Rinard. "Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions". In: *ACM Sigplan Notices* 35.5 (2000), pp. 182–195. DOI: 10.1145/358438.349325.

[153] 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: Association for Computing Machinery, Dec. 1, 1981, pp. 12–21. ISBN: 0-89791-062-1. DOI: 10.1145/800216.806586.

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

[155] 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). Ed. by Gregg Rothermel and Laura K. Dillon. New York, NY, USA: Association for Computing Machinery, July 19, 2009, pp. 225–236. ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572299.

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

[157] 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.

[158] Jonas Schöpf and Stephanie Widauer. *History of Interactive Theorem Proving*. Research rep. Universität Inssbruck, Feb. 26, 2018. URL: http://cl-informatik.uibk.ac.at/teaching/ws17/vs/reports/SWreport.pdf.

[159] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. "Disassembly of Executable Code Revisited". In: *Reverse Engineering*. Proceedings of the Ninth Working Conference (Richmond, VA, USA, Oct. 29–Nov. 1, 2002). Piscataway, NJ, US: IEEE, 2002, pp. 45–54. ISBN: 0-7695-1799-4. DOI: 10.1109/WCRE.2002.1173063.

[160] 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: Association for Computing Machinery, June 16, 2013, pp. 471–482. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462183.

[161] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. ""Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata". In: *7th USENIX Workshop on Offensive Technologies* (Washington, D.C. Aug. 13, 2013). 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Aug. 2013. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/shapiro.

[162] Jianqi Shi, Jifeng He, Huibiao Zhu, Huixing Fang, Yanhong Huang, and Xiaoxian Zhang. "ORIENTAIS: Formal Verified OSEK/VDX Real-Time Operating System". In: *ICECCS 2012*. 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. DOI: 10.1109/ICECCS20050.2012.6299224.

[163] Jianqi Shi, Longfei Zhu, Huixing Fang, Jian Guo, Huibiao Zhu, and Xin Ye. "xBIL – A Hardware Resource Oriented Binary Intermediate Language". In: *ICECCS 2012*. Proceedings of the IEEE 17th International Conference (Paris, France, July 18–20, 2012). Piscataway, NJ, US: IEEE, Sept. 13, 2012, pp. 211–219. ISBN: 978-2-9541-8100-4. DOI: 10.1109/ICECCS20050.2012.6299216.

[164] 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.

[165] 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: *Security and Privacy*. 2016 IEEE Symposium (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/.

[166] 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.

[167] 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. DOI: 10.1007/978-3-540-89862-7_1.

[168] Wei Song, Alex Bradbury, and Robert Mullins. "Towards General Purpose Tagged Memory". In: *RISC-V Workshop*. Proceedings. Vol. 122. Citeseer, 2015. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d111eea891bcddf8d7d2d54fbd4

[169] 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. 9593. Cham: Springer International Publishing, July 2015, pp. 202–222. DOI: 10.1007/978-3-319-29613-5_12.

[170] Ole Tange. "GNU Parallel – The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. DOI: 10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[171] The LLDB Team. *LLDB Homepage — The LLDB Debugger*. Sept. 9, 2022. URL: https://lldb.llvm.org/ (visited on 12/31/2022).

[172] *The seL4® Microkernel*. seL4 Project a Series of LF Projects, LLC. 2022. URL: https://sel4.systems/ (visited on 12/21/2022).

[173] 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).

[174] 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).

[175] Trail of Bits. *lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode*. URL: https://github.com/lifting-bits/mcsema (visited on 12/02/2022).

[176] Hans van Kranenburg. *Xen CPUID masking*. Original website does not appear to be available anymore. Aug. 18, 2016. URL: https://web.archive.org/web/20161124002801/https://tech.mendix.com/linux/2016/08/18/xen-cpuid-masking/ (visited on 12/16/2022).

[177] Hanpeter van Vliet and Eric Smith. *Mocha, the Java Decompiler*. May 20, 2007. URL: http://www.brouhaha.com/~eric/software/mocha/ (visited on 01/15/2023).

[178] VECTOR 35. *Binary Ninja*. May 10, 2022. URL: https://binary.ninja (visited on 05/18/2022).

[179] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. "Formally Verified Lifting of C-Compiled x86-64 Binaries". In: *Programming Language Design and Implementation*. Proceedings of the 43rd ACM SIGPLAN International Conference (San Diego, CA, USA, June 13–17, 2022). New York, NY, USA: Association for Computing Machinery, June 9, 2022, pp. 934–949. ISBN: 9781450392655. DOI: 10.1145/3519939.3523702.

[180] Freek Verbeek, Joshua Bockenek, and Binoy Ravindran. "Highly Automated Formal Proofs over Memory Usage of Assembly Code". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Proceedings of the 26th International Conference (Dublin, Ireland, Apr. 25–30, 2020). Ed. by Armin Biere and David Parker. Lecture Notes in Computer Science 12079. Berlin Heidelberg: Springer-Verlag, Apr. 17, 2020, pp. 98–117. DOI: 10.1007/978-3-030-45237-7_6.

[181] Freek Verbeek, Joshua A. Bockenek, Abhijith Bharadwaj, Binoy Ravindran, and Ian Roessle. "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 (La Jolla, California, Oct. 9–11, 2019). New York, NY, USA: Association for Computing Machinery, 2019, 17, pp. 1–5. ISBN: 9781450369978. DOI: 10.1145/3359986.3361215.

[182] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. "Sound C Code Decompilation for a Subset of x86-64 Binaries". In: *Software Engineering and Formal Methods*. Proceedings of the 18th International Conference (Amsterdam, The Netherlands, Sept. 14–18, 2020). Lecture Notes in Computer Science 12310. Berlin Heidelberg: Springer-Verlag, Sept. 8, 2020, pp. 247–264. DOI: 10.1007/978-3-030-58768-0_14. URL: https://ssrg-vt.github.io/FoxDec/.

[183] 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.

[184] 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 (San Diego, CA, USA, Feb. 26–Mar. 3, 2017). Internet Society, Feb. 27, 2017. DOI: 10.14722/ndss.2017.23225.

[185] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. "Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code". In: *Computer and Communications Security*. Proceedings of the 2012 ACM Conference (Raleigh, North Carolina, USA). New York, NY, USA: Association for Computing Machinery, Oct. 16, 2012, pp. 157–168. ISBN: 9781450316514. DOI: 10.1145/2382196.2382216.

[186] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. "Shingled Graph Disassembly: Finding the Undecideable Path". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2014, pp. 273–285. DOI: 10.1007/978-3-319-06608-0_23.

[187] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. "Differentiating Code from Data in x86 Binaries". In: *Machine Learning and Knowledge Discovery in Databases*. Joint European Conference (Athens, Greece, Sept. 5–9, 2011). Lecture Notes in Computer Science 6913. Berlin Heidelberg: Springer-Verlag, 2011, pp. 522–536. DOI: 10.1007/978-3-642-23808-6_34.

[188] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. "A New Algorithm for Identifying Loops in Decompilation". In: *Static Analysis*. 14th International Symposium (Kongens Lyngby, Denmark, Aug. 22–24, 2007). Ed. by Hanne Riis Nielson and Gilberto Filé. Lecture Notes in Computer Science 4634. Berlin Heidelberg: Springer-Verlag, 2007, pp. 170–183. ISBN: 978-3-540-74061-2. DOI: 10.1007/978-3-540-74061-2_11.

[189] Jordan Wiens. *Binary Ninja*. May 11, 2020. URL: https://binary.ninja/2020/05/11/decompiler-stable-release.html (visited on 07/06/2020).

[190] Jeffrey Wilhelm and Tzi-cker Chiueh. "A Forced Sampled Execution Approach to Kernel Rootkit Identification". In: *Recent Advances in Intrusion Detection*. Proceedings of the 10th International Symposium (Gold Coast, Australia, Sept. 5–7, 2007). Ed. by Christopher Kruegel, Richard Lippmann, and Andrew Clark. Lecture Notes in Computer Science 4637. Berlin Heidelberg: Springer-Verlag, 2007, pp. 219–235. ISBN: 978-3-540-74320-0. DOI: 10.1007/978-3-540-74320-0_12.

[191] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. "The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs". In: *Formal Aspects of Computing* 9.2 (1997), pp. 149–174. DOI: 10.1007/BF01211617.

[192] 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.

[193] Dachuan Yu and Zhong Shao. "Verification of Safety Properties for Concurrent Assembly Code". In: *International Conference on Functional Programming*. Proceedings (Snow Bird, UT, USA). Vol. 39. 9. New York, NY, USA: Association for Computing Machinery, Sept. 19, 2004, pp. 175–188. ISBN: 1581139055. DOI: 10.1145/1016850.1016875.

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

[195] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Hardware Enforcement of Application Security Policies Using Tagged Memory". In: *Operating Systems Design and Implementation*. Proceedings of the 8th USENIX Conference (San Diego, CA, USA, Dec. 8–10, 2008). 2560 Ninth Street, Suite 215, Berkeley, CA, USA: USENIX Association, Dec. 8, 2008, pp. 225–240. DOI: 10.5555/1855741.1855757.

Glossary

A

assembly

80286 An older iteration of the x86 ISA. 27, *see also* x86-64

ARM The RISC ISA developed by Arm Ltd. Comes in both 32-bit and 64-bit forms. 19, 20, 22, 23, 35

Intel 8080 An even older chipset than the 80286. 9, 18, 163, *see also* x86-64

MC68020 A 32-bit microprocessor produced by Motorola. 19, 23

RISC-V An open standard ISA based on established RISC principles. 28

SPACE-8080 A verifiable subset of the Intel 8080 ISA. 9, 18, 23, *see also* Intel 8080

x86 The family of CISC ISAs developed by Intel; in common parlance, refers to the 32-bit version. 19, 20, 23, 24, 26, 27, 163

x86-64 The 64-bit version of the x86 ISA. xiv, xxvii, 10, 11, 13–15, 21–23, 27, 33–35, 38, 41, 42, 44, 48, 50, 53, 56, 80, 94, 100, 102, 104, 121, 131, 137, *see also* x86

F

Floyd-style A binary verification approach using control flow graphs extracted from a restricted set of programs involving the usage of an external control flow analysis tool that are deconstructed using identified cutpoints. i, xiii, xxvi, 3, 9, 11–13, 17–19, 41, 42, 44, 79, 83, 136, 137, *see also* Hoare-style & CFG

Formal Methods

Coq A generic interactive theorem prover. 19, 20

HOL4 A theorem prover that uses HOL. 20, 22

Isabelle A widely used interactive theorem prover implemented in the functional programming language ML with a wide variety of useful libraries and strong support for syntactic and the construction of DSLs. Supports many different logics, but the most commonly used one is HOL. ix, x, xv, xxvi–xxviii, 10–12, 19, 22, 33–36, 39, 41–43, 56, 57, 64–66, 69, 70, 72, 73, 83, 104, 105, 138, 142

Z3 An SMT solver made by Microsoft. Its support for bitvector operations is very useful. xxvii, xxviii, 58, 61, 76, 88, 89, 108

H

Hoare logic An axiomatic framework for reasoning over the semantics of computer programs [87]. 14, 20, 22, 43, 66, 68, 69

Hoare-style A binary verification approach using an abstract lifting of control flow that can be deconstructed using formal rules based on Hoare logic. i, 3, 11, 12, 16, 17, 20, 27, 28, 79, 83, 136, 141, *see also* Floyd-style, Hoare logic & SCF

J

Java 20, 27, 30

L

LLDB The debugger used by the LLVM project. 29, *see also* LLVM

LLVM A project that provides an alternative to GCC for optimizing C/C++ compilation. 5, 27, 29, 108, *see also* C, C++ & GCC

M

[MC]SQUARE A model checker for microprocessors implemented in Java. 20, 21

memory usage A description or representation of the memory read and written by a program. The program is guaranteed to not *use* any memory outside those regions. To ensure correctness, it must be *overapproximative*. xiii, xvi, xxvii, 3–5, 9, 11–13, 17, 19, 21, 28, 33, 40–45, 47, 48, 52, 54–57, 64, 66–68, 71, 72, 74, 76, 136, 137, 139

P

Programming Language

C A classic low-level programming language. xiv, 3–7, 11, 15, 19, 21–23, 27, 29, 47, 49, 56, 60, 110, 134, *see also* C++

C++ A successor language of sorts to C, providing a lot more higher-level features while still being a relatively low-level language. i, iv, 4, 5, 9, 14, 15, 24, 27–30, 58, 110, 111, 118, 130, 134, 138, *see also* C

Fortran A classic language for mathematical computing and optimization. 15, 134

S

seL4 A secure, heavily-tested and verified microkernel. 10, 20, 22, 23, 28

STRATA A machine learning tool to extract instruction semantics developed by Heule et al. [84]. 10, *see also* x86-64

X

the Xen Project A fully-featured, production-capable hypervisor. 13, 21, 23, 24, 57, 74, 76, 77, 80, 94, 100, 102, 131, 137, 138