SCALABLE VALIDATION OF BINARY LIFTERS

BY

SANDEEP DASGUPTA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Abstract

The ability to directly reason about binary machine code is desirable, not only because it allows analyzing binaries even when the source code is not available (e.g., legacy code, closed-source software, or malware), but also avoids the need to trust the correctness of compilers.

Binary analysis is generally performed by existing decompiler projects by (1) converting raw bytes from the binary into a stream of assembly instructions through *disassembly*, (2) translating machine code to an intermediate representation (IR) using a binary *lifter*, and (3) performing various analysis and transformations on the IR pertaining to the specific goals of the decompiler. Many binary analysis frameworks published in academia or as open-source code, use such a lifter as the first step in their pipeline.

Validating the correctness of binary lifters is pivotal to gain trust in binary analysis, especially when used in scenarios where correctness is essential. Unfortunately, existing approaches focus on validating the correctness of lifting a single instruction and do not scale to full programs. I believe an effort in the direction would enable both the developers of binary translators, to validate their implementation, and the clients of those translators, to gain trust in their analysis results.

The overall goal of my work is to develop formal and informal techniques to achieve high confidence in the correctness of binary lifting by leveraging the semantics of the languages involved (e.g., Intel's x86-64 and LLVM IR).

Towards that goal, I made two broad contributions. First, I defined the most complete and thoroughly tested formal semantics of x86-64 to date. The semantics faithfully formalizes all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture. The formal specification covers 3155 instruction variants, corresponding to 774 mnemonics. The semantics is fully executable and has been tested against the GCC C-torture test suite. Moreover, each instruction is individually tested against more than 7,000 instruction-level test cases. This extensive testing paid off, revealing bugs in both the x86-64 reference manual and other existing semantics, which are all acknowledged, and some are fixed. Also, I illustrated potential applications of the semantics in different formal analyses, and discuss how it can be useful for processor verification.

Second, I show that formal translation validation of single instructions for a complex ISA like x86-64 is not only practical but can be used as a building block for scalable full-program validation. My work is the first to do translation validation of single instructions on an architecture as extensive as x86-64, uses the most precise formal semantics available, and has the widest coverage in terms of the number of instructions tested for correctness. Next, we develop a novel technique that uses validated instructions to enable program-level validation, without resorting to performance-heavy semantic equivalence checking. Specifically, I compose the validated IR sequences using a tool we develop called *Compositional Lifter* to create a reference standard. The semantic equivalence check between the reference and the lifter output is then reduced to a graph-isomorphism check through the use of semantic preserving transformations. The translation validation of instructions in isolation revealed 29 new bugs in McSema – a mature open-source lifter from x86-64 to LLVM IR. Towards the validation of full programs, our approach was able to prove the translational correctness of 2254/2348 functions taken from LLVM's single-source benchmark test-suite.

Dedicated to my parents (Benu & Pratima Dasgupta), sister (Mowsumi) & love of my life, Swetosree Sinha, for their unconditional love and support.

Acknowledgments

Coming up to this point is one of the biggest challenges I have ever pursued in my life. This thesis would not have been possible if I did not have the support of the following people.

I owe my deepest gratitude to my adviser, Professor Vikram Adve, whose guidance, patience, and encouragement have been pivotal in the successful completion of this thesis. He is one of the nicest & smartest people I have ever met in my life. He has an ocean-deep of patience to listen to all my ideas and never ceased to amaze me with his intelligent insights and thoughtful suggestions. Also, it is worth mentioning the vast amount of importance that he yields to the wellbeing of his students. I could not have imagined a better mentor during this journey. Thanks for providing me with this lovely memories!

I owe special thanks to Theodoros Kasampalis, Nathan Dautenhahn, Will Dietz, Daejun Park, Edward J. Schwartz, and Sushant Dinesh with whom I have collaborated at some point for research. I have enjoyed and hugely benefited from the many discussions we have had on our work and research life in general. I owe my thanks to all the (Ph.D.) committee members and my colleagues in the Low-Level-Virtual-Machine (LLVM) research group for their valuable feedback, thoughtful ideas, and insightful discussions. I am grateful to Alastair Reid and Matthew Fernandez for their invaluable feedback. Also, I want to thank the $\mathbb{K}$-Framework team, for their technical support throughout the project, and the Strata & Mcsema developers, for promptly confirming our reported bugs and answering all our questions in great detail. I am indebted to the Computer Science Department at Illinois for providing an extraordinary Ph.D. curriculum together with the flexibility conducive for research.

A very special word of thanks goes for my parents, Benu and Pratima, who have been supporting me unconditionally in all phases and ventures of my life and keeping me motivated over these years. Not to mention all the sacrifices that they made so that I can have a better place to live and a favorable environment conducive to my work. It's my father who always motivated me to believe in myself and had the biggest smile to know that I have made through this journey. I am so proud to realize his dream. My mother was always around at times I thought that it is impossible to continue and helped me to keep things in perspective. My Sister, Mowsumi (who is like my second mother), always protected me and letting me chase my dreams. A big thanks to you for taking care of everything back there at home and offering me the privilege to work without worrying. I owe thanks to the love of my life, Swetosree, for her continued and unfailing love, support, and understanding during my pursuit of the Ph.D. degree that made the completion of the thesis possible. I greatly value her contribution and deeply appreciate her belief in me. I consider myself the luckiest in the world to have such a supportive, lovely, and caring family, standing beside me all the time with their love and unconditional support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 VALIDATING BINARY LIFTERS: MOTIVATION

Analysis and reasoning about source code are one of the most pervasive concepts in computer science research. Analyzing the code to approximate the semantics of the program helps in determining the correctness of the program w.r.t some gold standard, determining illegal memory or control flow accesses, or proving/refuting various properties of interest to the users. Static analysis [1], model checking [2, 3], and abstract interpretation [4] are some of the well-known techniques used, widely and with ease, for the analysis of source code and have been deployed in many tools and processes that improve the software quality [5, 6, 7, 8, 9, 10, 11]. All such static source code analysis techniques are designed so that they are easier to apply, with greater precision, on human-readable source code written in high-level languages as opposed to low-level binary code.

Analysis at the binary level is complicated because of various reasons [12] including the fact that various source-level information, like symbol information, types, function boundaries, and their prototypes are completely or partially lost during the compilation process. The absence of symbol information and types in the binary code obscures the identification of variables, which are represented by reusable registers and the memory, which is addressable as a large continuous array. Registers and memory do not carry any type, and pointers of any type are indistinguishable from integers. Interestingly, the ability to reason about binary code allows analyzing the software even without the presence of source code but comes with a price because of the challenges mentioned above.

Despite the various challenges in analyzing machine code, understanding binary code is considered a pivotal task to guarantee the trustworthiness of critical systems spanning in various subfields of software engineering and security tools, including binary instrumentation [13, 14, 15, 16, 17], binary re-targeting [18, 19], software hardening [20, 21, 22, 23], software testing [24, 25, 26], CPU emulation [27, 28], automated reverse engineering [29, 30, 31, 32, 33, 34, 35], sand-boxing [36, 37, 38], profiling [39, 40], and automatic exploit generation [41].

Indeed, there are several compelling reasons to analyze at the binary level. We enumerate the most important ones as follows:

- One of the most favorable scenarios for doing the binary analysis is when source code is simply not available, and working with the binary is the only viable option. Such scenarios arise in the case of (1) legacy code when binary analysis allows to analyze or re-implement (or patch) the program, or (2) malware when binary analysis helps in security audits or malware detection [42, 43, 44, 45, 46].

- While analyzing source code, the libraries, which are not available in source format, are often replaced by coarse grained abstractions. These abstractions are manually created and, hence, are prone to error. In contrast, a binary analysis tool can analyze the library code directly [47]. Programs often require using libraries written in a different high-level language than the source program. This imposes a challenge in designing tool for analyzing source code because multiple languages need to be supported. Operating on the binary avoids these issues altogether since all high-level languages are translated into a hardware-specific, but single target language with no distinction between the source code or library code.

- There are other scenarios, e.g., security checking, high-assurance systems like aircraft, where it is not desirable to trust the compiler, even when the source code is available. Doing analysis directly at the binary level helps to achieve that by skipping the compilation step altogether and thereby excluding the compiler's correctness from the trusted computing base. During the compilation process, the source code undergoes various modifications, removal or additions, before translated to binary. Analyzing that binary is desirable because it is closest to what is actually executed on hardware [48].

1.2 CORRECTNESS CHALLENGES FOR BINARY LIFTERS

Binary analysis is generally performed by existing decompiler projects [33, 49, 50, 51, 35] using a three-step process.

Disassembly First, raw bytes in a binary blob are decoded, using a tool called *disassembler*, into a stream of assembly instructions. This step recovers many high-level primitives (like instructions, control-flow graphs, function boundaries, etc.) of the binary, which are otherwise lost during the compilation pipeline. Since, in any binary analysis of machine code, disassemblers represent the first link of a long chain of interconnected phases, a bug in the disassembly module would produce dangerous cascade effects on all the subsequent phases. Disassembly, in general, is undecidable [52]. The main reasons for such complexity are the impossibility of separating data from code, self-modifying code (i.e., code that modifies itself at run-time), and indirect control transfers (i.e., control transfers whose target is computed dynamically). To tackle the aforementioned problems, disassemblers assume that specific compilers conventions are respected, adopt several heuristics [53, 54, 55, 56], and work reasonably well. Nevertheless, when the code being disassembled violates the assumed conventions, when the heuristics fail, or even worse when code is obfuscated, disassemblers produce completely unreliable results.

Lifting Next, the assembly code is translated to an intermediate representation (IR), which precisely represents the operational semantics of the binary code. Such a process is often referred to as *binary lifting* and considered as one of the central steps in many binary analysis [33, 49, 50, 51, 35, 57, 58, 59, 60, 31, 61, 62] and instrumentation systems [16, 27] (refer to Table 1.1). The main reason that the binary analysis frameworks lift to a

Intermediate Representation (IR)	**Binary Analysis & Instrumentation Projects**
LLVM IR	McSema [33, 49], Dagger [63, 64], bin2llvm [65], fcd [57], rev.ng [66], llvm-mctoll [61], reopt [67], retdec [68], revgen [62], fracture [69]
BIL	BAP [51]
VEX	Angr [34, 50, 70, 71], PyVex [72], Valgrind [16]
ESIL	Radare2 [35]
TCG	QEMU [27]
Vine	BitBlaze [58]
DBA	BINSEC [73]
REIL	BinNavi [74]
Microcode	HexRay [59], Insight [75]
Dream	Dream [32]
TSL	CodeSurfer [76]

Table 1.1: Intermediate representations used in different binary analysis & instrumentation projects

platform-independent IR is to handle the complexity and diversity of modern instruction set architectures (ISAs) and thus enabling greater retargetability of the frameworks to multiple ISAs. Such an IR allows implementing analysis independently of (i) names and number of registers, (ii) instruction decoding, (iii) endianness of memory access, and (iv) instruction side-effects.

Post-Lifting Actions Finally, various analysis and transformations are performed on the lifted IR. This step is meant to serve one or more of the following goals: (i) recover higher-level constructs, such as functions, stack frames, variables, and types [61, 59, 31, 32, 77, 78, 67, 62], (ii) reason about the binary code [51, 50, 35, 64], (iii) re-target to a different ISA [33, 79, 80, 27, 81, 82], or (iv) instrument and recompile the binary for various purposes [83, 27, 16, 35]. Note that, these IR-level post-lifting actions are decided entirely by the ultimate goal of the decompilation framework.

Many binary analysis frameworks [51, 58, 33, 49, 50, 35, 57, 66] use such a lifter as a key step in their pipeline. Developing a lifter, especially for complex modern ISAs, is challenging and error-prone, mainly because of the following reasons:

- Manually encoding the effects of a vast number of instructions (and their variants) is hard. For example, x86-64 ISA has a large number of instructions, partly because of a large number of complex instructions and partly because it keeps most of the legacy and deprecated instructions for the sake of backward compatibility. In total, the x86-64 ISA consists of 996 unique mnemonics with 3736 variants, depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands.

- The informal specifications provided by the hardware manufacturers run into thousands of pages [84, 85] and have ambiguities, missing specifications, or even bugs [86]. This forces the developers of the binary lifter to consult with an actual processor implementation to clarify such details.

- Some instructions variants have divergent behaviors more than the difference of their type and size. For example, **`movsd`**, one of the 128-bit SSE instructions, has very different behaviors depending on whether the type of the source operand is register or memory; it clears the higher 64 bits of the target register only when the source type is memory. Such discrepancies prevent a naive generalization of the encoding of one instruction variant to others.

- The x86-64 standard also admits undefined behaviors that are implementation-dependent. Many instructions (32 out of 996 mnemonics) have undefined behaviors: their output values of the destination register or the %rflags register are undefined in certain cases. That is, the processor is free to choose any behavior in undefined cases. For example, the parity flag `%pf` is undefined in the logical-and-not instruction **`andn`**, where the processor implementation is allowed to either update the flag value (to 0 or 1), or keep the previous value.

- Once such a lifter is developed, the developers then run into the problem of not having a way to test their implementation thoroughly as generally, there are no formal, machine-readable semantics available for automated testing. For example, despite several explicit attempts [87, 88, 89] and other related systems [90, 49, 91, 92, 93], there does not exist any *complete* formal semantics of x86-64 up until we define a fully executable formal semantics of x86-64 [86].

- These lifters need to be updated and rechecked for correctness every time new instructions are added to an ISA.

1.3 OVERALL GOAL OF OUR WORK

Despite the correctness challenges in binary lifting, such lifters are sometimes used for tasks where correctness is especially important, e.g., when looking for security vulnerabilities in binary code, binary emulation of one processor ISA on another, or recompiling embedded software to run on new platforms. Any bug in the lifting would invalidate the binary analysis results. For example, a malware analysis system might miss vulnerabilities or a binary instrumentation system, instrumenting a buggy IR, might lead to failure or even crash in interpreting the instrumented program. Therefore, automatic validation tools are needed urgently to uncover hidden problems in binary lifters, which constitute the motivation and theme of our work. Beyond these more critical tasks, gaining confidence in binary lifters through more effective testing is also generally crucial for developers of decompilers, especially for complex ISAs.

The overall goal of our work is to develop formal and informal techniques to achieve high confidence in the correctness of binary lifting, from a complex machine ISA (e.g., x86-64) to a rich IR (e.g., LLVM IR), by leveraging the semantics of the languages involved.

Through the rest of this work, we fix our discussion to lifting x86-64 to LLVM IR using the most mature, open-source lifter McSema [33]. LLVM, being an industry standard compiler IR, many decompilation projects (refer to Table 1.1) prefer to employ LLVM as their lifted representation mainly because it enables various "out-of-the-box" analyses and optimizations which minimizes the effort in the post-lifting decompilation tasks. Moreover, LLVM IR is backed-up with its formal semantic models [94, 95], which assists formal reasoning on a program written in LLVM IR. Similarly, x86-64 instruction set architecture (ISA) is one of the most complex and widely used ISAs on servers and desktops. Hence, it is imperative to ensure the correctness of lifter targeting x86-64 program binary to LLVM IR. We note that our techniques are generally applicable to verify binary lifters from any ISA, e.g., x86, ARM, RISC-V, PowerPC, to an intermediate representation, such as LLVM IR [96], VEX IR [16] etc., as long as the formal semantics for both the ISA and the target IR are available.

1.4 SUMMARY OF RELATED WORK

In this section, we would provide a brief survey of the state-of-the-art validation techniques for binary lifting. As mentioned earlier, we would like to leverage the semantics of the source and target languages of the translation; one of the strongest ways to formally reason about the correctness of a lifter's translation. Hence it is equally important to investigate upon the current state of the available formal specifications of the concerned languages.

Existing Techniques for Validating Binary Lifters Surprisingly, there has been very limited work to date on validating the correctness of binary decompilers, and that work has focused on the translation of single instructions. All of this existing work falls short in at least one of the following criteria: (i) require random test-inputs which leads to incomplete coverage [97, 98, 99], (ii) do not scale to full program translation validation [100, 101], (iii) require modification of the lifting frameworks under test to emit additional information required to prove correctness [102], or (iv) need heavyweight symbolic execution to generate verified translation [103, 104].

Existing Semantics for x86-64 Heule *et al.* [87] presented a formal semantics of x86-64, but it covers only a fragment ($\sim$47%) of all instructions; as the authors of [87] candidly admitted, their synthesis methodology proved insufficient to add the remaining instructions primarily due to limitations of the underlying synthesis engine. Moreover, their semantics misses certain essential details (Section 4.1 & 3.5). Also, it would require extensive effort to translate their instruction-level semantics to use in a full-fledged theorem prover to be able to reason about the full functional correctness of the x86-64 binary.

Goel *et al.* [88, 89], on the other hand, specified a formal semantics in the ACL2 proof assistant [105], allowing to reason about functional correctness, but their semantics covers only a small fragment ($\sim$33%) of all user-level instructions.

There also have been several attempts [50, 51, 35, 93] to *indirectly* describe the x86-64 semantics, where they define an intermediate language (IL), specify the IL semantics, and translate x86-64 to the IL. This indirect semantics, however, may not be general enough to be used for different types of formal analyses, since the IL might be designed with specific purposes in mind, not to mention that the translation may miss certain important details of the instruction behaviors. Refer to Section 2.2 for a more detailed comparison to existing semantics.

1.5 INTUITION AND SUMMARY OF OUR APPROACH

Our current approach is inspired by a key observation that most of the decompilers [33, 63, 57, 66, 65, 61, 67, 68, 62, 69, 51, 34, 35, 58] are designed to perform simple instruction-by-instruction lifting (using a fixed and canonical representation of architectural state at the IR level), followed by standard IR optimization passes to achieve simpler IR code[1]. We capitalize on this observation by deriving the following insight:

Formal translation validation of single machine instructions can be used as a building block for scalable full-program translation validation.

With that insight, we define a fully executable formal semantics of x86-64, using $\mathbb{K}$ as the language engineering framework, to assist translation validation of individual instructions. This is the most complete formal semantics of x86-64 user-level instructions to date, which has been thoroughly tested using synthesized test inputs and the GCC torture tests. As our starting point, we took the semantics from Heule *et al.* [87] to avoid duplicating the formalization effort. We made several corrections or improvements to this semantics to improve both soundness and efficiency. We *automatically* translated their semantics into $\mathbb{K}$, and cross-checked the translated semantics against the original using an SMT solver. We *manually* specified, in $\mathbb{K}$, the semantics of the remaining instructions faithfully consulting the Intel manual [84] to obtain the complete semantics. We have also illustrated several potential uses of the semantics, which are realized by the formal analysis tools derived right from the $\mathbb{K}$ specification. The $\mathbb{K}$ framework also enables us to represent the semantics as SMT theories, which other projects can leverage for their purposes. We describe several practical lessons we have learned from our experience in developing the semantics,which could be useful for future formal specifications of processor ISAs.

Next, using the above semantics and a publicly available semantics of LLVM IR [94], we develop a single-instruction translation validation framework. The idea is to validate the translational correctness of a single instruction (single-instruction translation validation) by asserting equivalence of symbolic summaries of the x86-64 instruction and the lifted LLVM IR sequence using an SMT solver. If they are equivalent, the lifted LLVM IR sequence is correct for the instruction, else the equivalence check fails, and the solver generates a counter-example that we use to report a bug.

With single-instruction translation validation as building block, we propose a scalable approach for full-program translation validation; one that does not require heavyweight equivalence checkers or theorem provers and based on semantics-preserving transformations and isomorphic graph matching.

[1]To the best of our knowledge, we have not come across a lifter which deviates from this observation.

Specifically, for program-level validation, we compose these validated sequences of instructions using a tool we developed, called Compositional Lifter, to form a *reference translation* (`T′`). Next, we transform `T` (the IR program generated by the lifter we wish to validate) and `T′`, one function at a time, using semantics-preserving transformations, to prune any syntactic differences except for the names of virtual registers and the order of non-dependent instructions. Finally, we check if the data dependence graphs corresponding to the transformed function pairs `F` & `F′`, of `T` and `T′` resp., are isomorphic in which case the two lifted sequences (`T` & `T′`) are deemed as semantically equivalent[2].

1.6 CONTRIBUTIONS

Our contributions in this work can be broken down into two logical components, which also follows the chronological order in which we broke down and devised tools and techniques for solving the problem of validating binary lifters. Below we cite the primary contributions of our work.

Defining the Most Complete Formal Semantics of x86-64 The main contribution here is the design of a fully executable formal semantics of x86-64 using $\mathbb{K}$ as the language engineering framework.

1. **Complete.** We present the most complete formal semantics of x86-64 to date. Specifically, our semantics formalizes all the user-level instructions of the x86-64 Haswell ISA (that is, 3155 instructions covering 774 mnemonics [84]), except deprecated ones (336 instructions), the AES cryptography extensions (35 instructions), and the system & concurrency-specific instructions (210 instructions) (Section 3.4.1).

2. **Faithful.** Being executable, the semantics of *each* instruction has been thoroughly tested against $7,000+$ test inputs using the co-simulation method (Section 3.5). We further strengthened the faithfulness of our semantics by testing it using 1548 x86-64 programs from GCC C-torture tests [107].

3. **Revealed bugs.** Our extensive testing paid off in revealing 8 bugs [108] in Intel manual [84] (Volume 2: March 2018), the standard document on x86-64. All these findings were reported and acknowledged by Intel as bugs in the manual [108] and are fixed in the latest revision. Also, we found 42 bugs in existing semantics like Strata

[2]This equivalence claim follows from Horwitz et al. [106] and will be elaborated in Section 4.4.2.

and Stoke [109, 110]. All these bugs were reported to the authors, and have been acknowledged and some have been fixed [109]. The details are in section 3.5.

4. **Usable.** We illustrate the potential of our semantics to be used for formal analyses such as deductive program verification and program equivalence checking (Section 3.6). The $\mathbb{K}$ framework also enables one to represent our semantics as SMT theories, which allows others to easily reuse our semantics for their own purposes. Indeed, we have translated our semantics to Stoke [111] which can serve as a drop-in replacement of Heule *et al.*'s semantics [87] and can immediately benefit tools built on Stoke (e.g., [112]).

Developing Techniques for Scalable Validation of Binary Lifters The main contribution here is the development of a scalable approach to do full program level validation that does not require any heavyweight equivalence checkers or theorem provers.

1. **First single-instruction translation validation framework for x86-64.** We develop the first single instruction translation validation framework for x86-64. Our work applies the *most comprehensive formal semantics* for x86-64 known to date, and has the *most comprehensive* coverage in terms of the number of instructions tested when compared to earlier work [101].

2. **Single-instruction translation validation revealing bugs.** The single-instruction translation validation framework is experimentally verified to be effective in finding bugs. In particular, we find bugs in the lifting of 29 instructions in McSema [33, 49], a well-tested, mature [113], and open-source lifter for x86-64 to LLVM IR, clearly showing the effectiveness of our technique. All of these bugs have been acknowledged by the McSema developers, and some have been fixed.

3. **Compositional Lifter.** Given a lifter (D), we show that we can construct an alternate lifter, called *Compositional* Lifter ($\mathtt{D}'$), which essentially concatenates the lifted IR sequences for individual instructions (which are proven correct by part 1, above) to provide a reference translation $\mathtt{T}'$. We do not provide a formal guarantee that the above composition $\mathtt{T}'$, of validated lifted IR sequences, is the correct translation of $\mathtt{P}$. However, the tool is exceedingly simple to construct and we added it to our trust base. More importantly, the translation $\mathtt{T}'$ generated by the Compositional Lifter is syntactically very similar to the translation (say $\mathtt{T}$) generated by the lifter we aim to validate. Such code similarity serves as a foundation for scaling translation validation to full programs.

4. **Scalable Program-level validation.** We propose a scalable approach for full-program translation validation that does not require heavyweight equivalence checkers or theorem provers. Our key insight is that there exists a semantics-preserving transformation — dubbed a canonicalizer— for each pair of functions `F` and `F′` of `T` and `T′`, say `canonical(F)` and `canonical(F′)`, such that, stated informally, we can check if `F` and `F′` are semantically equivalent by checking whether `canonical(F)` and `canonical(F′)` have isomorphic data dependence graphs. If such a canonicalizer exists, then we can reduce the problem of program-level semantics checking to a much cheaper graph-isomorphism check! In this work, we construct an approximation of a canonicalizer, called Transformer, out of a very short sequence of 17 manually selected LLVM passes, of which 11 are transformation passes. The lifted functions `F` & `F′` being syntactically very close to begin with, we found such an approximation to work quite well. For example, the data dependence graphs extracted from the optimized versions when matched for graph isomorphism yield a low false-alarm rate of 7% based on our evaluation setup (Section 4.5). Moreover, we improved the matching results by employing an autotuner to automatically discover pass sequences, for each pair of `F` & `F′`, by searching over the manually-identified 17 optimization passes. This approach reduces the false-alarm rate from 7% to 4%, and with fewer passes, on average 8 instead of 17.

Our x86-64 formal semantics and the validation framework (built on top of the former) are publicly available at [114] and [115] respectively.

1.7 THESIS ORGANIZATION

The remaining chapters are organized as follows. In Chapter 2, we survey the current state of the art in validating binary lifters. We will also summarize the previous work in defining the formal semantics of x86-64 ISA. Chapter 3 gives details on our approach in defining the most complete formal semantics of x86-64. It elaborates on the challenges involved and how we ensured the faithfulness of the semantics and demonstrated its applicability. Chapter 4 presents the design and evaluation of core techniques for validating binary lifters. Chapter 5 presents the key take-aways from the presented work and Chapter 6 discusses some directions for future work. Finally, Appendix A provides a detailed description of the single-instruction translation validation phase, emphasizing more on various aspects of its design.

CHAPTER 2: RELATED WORK

This chapter gives context to this thesis by giving a literature review of the topics relevant to the presentation. The topics mainly include the state-of-the-art validation techniques for binary lifting and the current state of the available formal semantics of x86-64 ISA. Moreover, we would like to detail some of the pioneering work in the field of translation validation for compiler transformations. We note that the basic concepts of translation validation, when applied to validating binary lifting, will remain the same.

2.1 VALIDATION OF BINARY LIFTERS

Given the importance of establishing the faithfulness of the binary lifters, there exists a couple of efforts towards that direction, which we will elaborate on next. All the previous approaches can be broadly categorized to be based on (1) Simulation-testing, and (2) Formal Methods.

2.1.1 Testing-based Approaches

This approach is similar to black-box testing in software engineering. Most notable works include Martignoni et al. [97, 98, 100] and Chen *et al.* [99].

Martignoni et al. [97, 98] propose hardware-cosimulation based testing on QEMU [27] and Bochs [80]. Specifically, they compared the state between actual CPU and IA-32 CPU emulator (under test) after executing randomly selected test-inputs on randomly chosen instructions to discover any semantic deviations. Although, a scalable and straightforward approach, it's effectiveness is limited because many semantics bugs in binary lifters are triggered upon a specific input and exercising all such corner inputs, using randomly generated test-cases, is impractical.

In general, such testing based approaches are unsuitable for validating whole program (or even basic block) translations because even a correctly translated program may not always produce exactly the same output as the original program due to the differences in modeling of the architectural states in the translated program (or basic block) vs the original program. Although it is possible to engineer out such architecture-state-comparison problems, but still these approaches might not detect some intermediate mistranslated instructions which are course corrected at the end. As an example, suppose a register is assigned values twice in a program, and the first assignment is mistranslated, but the second assignment statement is translated correctly. Comparing the architecture states at the end of the program (or basic block) may not discover the mistranslation.

Chen *et al.* [99] proposed validating the static binary translator LLBT [79] and the hybrid binary translator [116], re-targeting ARM programs to x86 programs. First, an ARM program is translated offline to x86 program (via an intermediate translation to LLVM IR). Next, the translated x86 binary is executed directly on a x86 system while the original ARM binary runs on the QEMU emulator. During run time, both the ARM binary and the translated x86 binary produce a sequence of architecture states, which are compared at the granularity of single instruction after solving the architecture-state-comparison problem, as mentioned above. The validator is evaluated using the ARM code compiled from EEMBC 1.1 benchmark. Like the previous approach, the validation of single instruction's translation is based on testing and hence shares the same limitation of not being exhaustive.

Martignoni *et al.* [100] validate a "buggier and less complete" Lo-Fi emulator [27] by generating high-fidelity test-inputs creating using symbolic execution of a "faithful and more complete" (in terms of IA-32 ISA) Hi-Fi emulator[80] implementation of an instruction semantics. They execute each test instruction twice, once on a real hardware and next on the Lo-Fi emulator, and comparing the output states. However, the work [100] does not aim to validate the translation of full programs, which is one of our primary contributions. Note that an approach as above cannot scale naturally to binary function validation because a set of high-coverage test-inputs for all the constituent instructions of a function cannot trivially derive high-coverage test-inputs for the whole function.

Schwartz *et al.* [31] proposed control-flow structure recovery by employing semantics preventing schema and tested their binary-to-C decompiler, Phoenix on a set of 107 real world programs from GNU coreutils. Along similar lines, Yakdan *et al.* [32] presented a decompiler, DREAM, to offer a goto-free output. DREAM uses a novel pattern independent control-flow structuring algorithm that can recover the control constructs in binary programs and produce structured decompiled code without any goto statement. The correctness of their approach is demonstrated using GNU coreutils.

2.1.2 Formal Methods based Approaches

The following are the efforts to establish formal guarantees for binary translations using formal methods.

The closest work to ours in finding instruction-level bugs is MeanDiff [101], which proposed an N-version IR testing to validate three binary lifters, BAP [51], BINSEC [73], and PyVEX [72] by comparing their translation of a single binary instruction to BIL, DBA, and

VEX IRs respectively. The IRs are first converted to some unified IR representations, one at a time. The resulting IRs are then symbolically executed to generate symbolic summaries for comparison using an SMT solver. First, we formally validate the translation of an instruction, using a thoroughly-tested semantics [86], as opposed to comparing the translation to other potentially incorrect translations. Second, the IRs they support are simpler than LLVM and so it is unclear whether the approach would be effective if LLVM had to be translated to the unified representation. Third, we perform program-level validation, which is not addressed by MeanDiff.

Interestingly, the MeanDiff paper [101] says that one motivation for relying on differential testing was that no formal specification of x86-64 ISA was available at the time. We do not have that limitation because we have developed a formal and thoroughly tested x86-64 ISA specification [86, 117], and made it publicly available.

The work closest to ours, in terms of the goals, is the translation verifier, Reopt-vcg [102], which addresses verification challenges specific to the translator Reopt [67]. The verifier, which validates translations at basic-block level, is assisted by various manually written annotations, which are prone to errors. In future, they aim to generate such annotations automatically by instrumenting the lifter. Our approach does not need any such annotations, avoiding the overhead of maintaining instrumentation patches whenever the lifter is modified. Moreover, the validator uses the semantics of a small subset of x86-64, which limits its applicability to small programs. We incorporate a fairly complete x86-64 semantics [86], allowing translation validation for larger and more diverse programs.

Myreen et al. [103, 104, 118] (and later improved by Fox [119] w.r.t. the tool support) presented "decompilation into logic" — a framework for verified decompilation, where machine code is decompiled into tail-recursive functions defined in the language of the HOL4 theorem prover [120]. The decompiler proves a theorem stating that the function accurately describes the effect of the given machine code. Sewell at al. [121] proved correctness of compilation of the seL4 microkernel from C source down to ARM machine code by building on a formal model of ARM code generated by extending the above work of Myreen. Such a verified (de)compiler includes critical design decisions which need to be incorporated early in the design phase with the goal of verification in mind, and cannot easily be applied retroactively in existing (de)compilers.

2.2 DEFINING FORMAL SEMANTICS OF x86-64

There have been many projects that host a formal semantics of x86-64 either as their main contribution or as part of their infrastructure. This section summarizes such previous

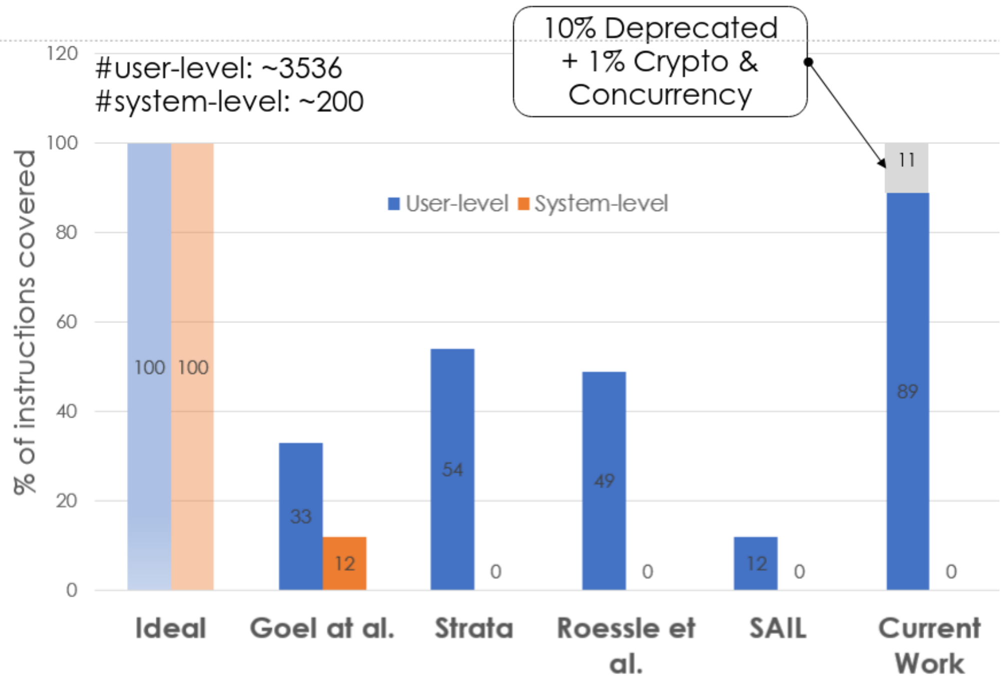

Figure 2.1: Projects hosting x86-64 ISA semantics. "Ideal" serves as a hypothetical baseline supporting **all** user-level and system-level instructions.

work and compares it to our formal semantics based on three directions that reflect the primary contributions of our work: completeness, in terms of supported user-level instructions; faithfulness, in terms of whether it is executable and hence can be evaluated with real code execution; and generality, in terms of its applicability to formal reasoning techniques.

Figure 2.1 shows a quick overview of the instruction coverage of our work compare against other related projects. The x-axis shows the various projects hosting x86-64 semantics, with "Ideal" being a hypothetical baseline project supporting *all* user-level and system-level instructions. The y-axis shows the percentage of instructions covered. For example the hypothetical "Ideal" project which supports all the instructions (user- plus system-level) with 95% of instructions being user-level and 5% of total instructions system-level. Our current work is the bar on the extreme right and the percentage of user level instruction that we formalize is pretty good as compared to other projects. However, we do not support system-level instruction, where as the x86-64 semantics in ACL2 [88] does support a subset of it. Moreover, we fall short of full 100% user-level coverage by 11%, which includes 10% deprecated (MMX and x87) and 1% cryptography and concurrency related instructions.

Strata [87] uses program synthesis to generate the instruction semantics of X86-64 as SMT bit-vector formulas. Automatically learning the formal semantics of 60% of the target x86-64 ISA is impressive, and we leverage this result in our work. However, the other 40% of the user-level instructions are not straightforward to automatically learn by their algorithm, mainly due to limitations of the underlying synthesis engine. Moreover, the specifications are executable only for non-floating-point (FP) instructions.

A contemporary work by Roessle *et al.* [112] presents a method to extract the big step semantics of a binary program using the small step instruction semantics extracted mostly from Strata[1] plus some manually drafted support for branching instructions and stack operations. Like Strata, their specification is executable only for the non-floating-point instructions. Moreover, their work does not aim for completeness of semantics, one of our primary goals.

Goel *et al.* [88] use the ACL2 theorem prover [105] to model the x86-64 ISA and they support 33% of all user-level instructions [122], plus some system-level instructions, paging, and segmentation. This list is far from a complete semantic definition of x86-64, but it is still the state-of-the-art in terms of formal analysis applied directly to x86-64 code. It is also an executable definition as demonstrated by its use for simulations. In our work, we do not leverage this definition, since Strata has defined many more instructions.

The CompCert verified compiler [90] includes semantics definitions for all intermediate and target languages used within the compiler, including a definition for 32-bit x86 assembly. The definition is specified in Coq [123] and has been used in a formal setting for proving the correctness of CompCert's compilation step to assembly, as well as outside CompCert, e.g., in proofs relating to the certified concurrent OS kernel CertiKOS [124]. However, this definition focuses on the 32-bit x86 instruction set, which is a subset of the x86-64 instruction set. Moreover, it is part of the trust base for CompCert and it is not clear whether or how it has been tested against an actual processor, whereas Strata and ours have been extensively tested.

TSL [91] is a system that can auto-generate tools for various machine code analyses given a semantics definition of the machine language written in the TSL specification. Such a semantics definition for the integer instructions (i.e., no floating-point instructions) of the 32-bit x86 instruction set is given as part of the project. It is used to generate various tools, including a machine code synthesizer [125]. This definition, to our knowledge, has not been used for formal verification proofs, i.e., to prove whether a given x86 program meets its specification.

Our semantics, like all the other work cited above, uses a sequential consistency memory model, and not weaker memory models. Existing efforts to specify weaker memory models for x86-64 such as Owens et al. [126] and Sarkar et al. [127], however, suffer from their limited support for instruction semantics (i.e., they consider only a small subset of 32-

[1]There are some minor omissions on immediate instructions with 8-bit operands for which Strata learns 256 brute force formulas.

bit x86 instruction set). We believe that integrating these two complementary efforts is a promising direction toward rigorously reasoning about real-world programs running on modern multiprocessors (e.g., using the Sail framework as we will describe below).

Sail is another language semantics framework, tailored for describing an instruction-set architecture semantics. Sail has been used to specify the semantics of ARMv8-A, RISC-V, and CHERI-MIPS [128], as well as the semantics of a small subset of x86 [129]. Sail is similar to the K framework we employed, but K is far more general-purpose than Sail. Also, the Sail x86 semantics is much more limited than ours. It describes the semantics of a fragment of 32-bit user-mode x86 instructions, while ours covers also the 64-bit counterpart as well as the floating-point instructions. Sail, however, allows us to integrate a semantic definition with their relaxed memory models [130, 131] for concurrency semantics. We believe that (automatically) translating our semantics into Sail[2] is a promising direction to obtain concurrency semantics and thus enable concurrency reasoning for x86 programs, which we leave as future work.

Degenbaev [134] formally specify the x86 instruction set architecture as part of the Verisoft XT project. The specification is developed to serve three primary goals: (1) to prove the correctness of a concurrent program for all possible interleavings, (2) to ensure the security of the hypervisor, and (3) to prove that the hypervisor emulates the guest instructions correctly. Unlike the scope of our model, their specification covers parts of system programming, concurrency, memory protection and virtualization related instructions. Their support for user level instructions, which is the main focus of our work, is quite limited. The work laid out a technique that could potentially validate their specification [34, Chapter 19]. Our semantics, being executable, is thoroughly tested using cosimulation against hardware where we executed the instructions twice, once on a real hardware and next on the interpreter provided by our model and the outputs are compared. Also, we validated our semantics by comparing it with related projects [87, 135] hosting x86-64 semantics. The co-simulation testing involves testing each supported instruction using 7000+ test-inputs, and testing 1548 programs from GCC C-torture tests [107].

Overall, the key differentiator of our effort compared to the existing work, as cited above, is that our semantics achieves (A) completeness of supported user-level instructions, (B) faithfulness, and (C) applicability to formal reasoning analyses. In Section 3.8, we elaborate on our novel approaches that allow us to achieve this unique combination.

[2]Indeed, the Sail ARMv8-A semantics is automatically generated from the ARM-internal specification of ARMv8-A [132] written in the ARM's architecture specification language, ASL [133], by using the ASL-to-Sail translator [128].

There are various binary analysis projects that target x86-64 binaries and lift them to a higher-level representation more suitable for the specific analysis. These include Angr [34] using the VEX IR of Valgrind [16], the QEMU [27] emulator using the TCG IR, the software fault isolation tool RockSalt [136] using its own RTL DSL, the disassembler and binary analyzer Radare2 [35] using the ESIL IR [137], the binary analysis tool BAP [51] using the BIL IR, and the static binary translator Remill [33] using LLVM IR [96]. We refer to these semantics as *indirect* because they give the semantics of the x86-64 binary via the translation to their IR, as opposed to a *direct* semantics such as ours and the others cited earlier. A direct semantics has significant advantages over an indirect semantics. For example, without the direct semantics of x86-64, we cannot even formulate the correctness of a translator from x86-64 to the IR. Analogously, many programming languages (C, C++, Java, etc.) have been given direct semantics, instead of indirect semantics by translation to other languages, for formal reasoning at the desired language granularity.

Hasabnis *et al.* [92, 93] also present an indirect semantics of x86-64, but in contrast to other indirect semantics, they use machine learning [92] and symbolic execution [93] to automatically learn the translation of x86-64 instructions to their IR, by extracting knowledge from the hard-coded translation logic of compilers such as GCC. However, as they admitted [93], their semantics omits some important details of x86-64 semantics (e.g., the effect of various instructions on CPU flags), and thus is not sufficient to serve as a solid foundation for rigorous formal analyses of x86-64 binary.

2.3 DEFINING FORMAL SEMANTICS OF ISA (OTHER THAN X86/x86-64)

Formal specification of instruction sets are critical in directly reasoning about low-level machine code [86, 89, 88, 112], verifying computer processor architectures [138, 139, 132, 140], compilers [141, 90, 142, 143], verifying assembly language functions against a specification [144], binary rewriting across architectures [82], verified binary decompilation [145, 78, 146, 103], abstract interpretation of binaries [147, 148], automated testcase generation [149, 100], synthesis of instruction semantics [87], and formalizing multiprocessor memory models [127, 126]. There exist a number of notable specifications of ISAs (e.g., RISC-V [128, 90, 141], MIPS [128, 119], CHERI-MIPS [128], PowerPC [143, 91, 90], SPARC [91], ARM [150, 151, 152, 119, 140, 132, 153, 154] etc.) other than x86/x86-64 that have hit many of the same problems that we and others in the x86-64 community have had to deal with. In this section, we will be focusing on some of the notable formal models of Arm [85] architecture, which is indeed comparable to x86-64 w.r.t. its size and complexity, and their similarities or dissimilarities with our model.

Fox and others from the University of Cambridge have a long history of contribution in formalizing the semantics of multiple versions of the Arm processor architecture [150, 151] using HOL4 proof system [120]. The semantics is used for a long time to propel verification efforts in various domains including formal verification of ARM processor [155, 156], decompilation [119, 103, 104], microkernels [121], hypervisors [157] and compilers [141, 143]. The specification, in its current form [151], is tested systematically against actual hardware for all usermode, integer instructions against three actual processors. However, the semantics precludes the full scope of the specification involving floating point and vector instructions, exceptions, interrupts, privilege checks, virtual memory, etc. Flur et al. [154] presents concurrency models for the the ARMv8 architecture and combines that with a complete application-level non-FP/SIMD ISA semantics. The former are validated by discussion with ARM and by black-box litmus testing; the latter by single-instruction testing. The specifications are written in Sail [128].

Like the above approaches [154, 151], our x86-64 model is tested against actual processors using directed tests. However, unlike [151], our semantics is not defined using a language of a theorem prover, but using $\mathbb{K}$ framework, which is comparatively much simpler and intuitive to write the semantics of a language. This is evident from the major sizable language semantics defined in K so far, which served as a great source of inspiration for our semantics: C [158], PHP [159], Python [160], Java [161], and JavaScript [162]. All these semantics are executable and they have been validated by a large volume of tests, and demonstrated their usefulness in formal analysis using the tools (auto-)generated by the $\mathbb{K}$ framework, similar to our x86-64 semantics. Moreover, unlike the language of a theorem prover which is usable to those familiar with the particular theorem prover, the $\mathbb{K}$ specifications does not compromise much on the readability of the specifications as evident from the fact that $\mathbb{K}$ specification has be used for formal documentation purposes [163] in the domain of blockchain. Sail [128], on the other hand, is a promising language semantics framework, tailored for describing an instruction-set architecture semantics and has been successfully used to specify the semantics of ARMv8-A, RISC-V, and CHERI-MIPS [128], as well as the semantics of a small subset of x86 [129]. Sail is similar to the $\mathbb{K}$ framework in the sense that it can auto-generate artifacts for runtime (emulators) and formal analysis tools given the specification of a language. One of the key differences is that $\mathbb{K}$ is far more general-purpose than Sail. Sail, however, allows us to integrate a semantic definition with their relaxed memory models [130, 131, 154] for concurrency semantics. We believe that (automatically) translating our semantics into

Sail[3] is a promising direction to obtain concurrency semantics and thus enable concurrency reasoning for x86-64 programs, which we leave as future work.

Unsurprisingly, creating a high-coverage and trustworthy specification from the documentation is hard because of the sheer size of instruction set and the fact that such documents are often informal. This is made even harder as those documentations are subjected to regular extensions or corrections. Reid [164, 153, 140, 132] made an noteworthy contribution to cope up with the problem by creating a methodology to automatically develop high quality processor, machine-readable, and executable architecture specifications from the same materials used to generate conventional documentation [132]. The specifications are written using an executable, strongly-typed, imperative style domain specific language (DSL) called the Arm Architecture Specification Language (ASL) and regarded as the most complete and throughly tested as compared to its predecessors. Such a single trusted specification has the benefit that each time one group finds and fixes a bug in the specification, it improves the utility of the specification for other groups. Moreover, the developed methodology involves building various tool chains around the specification to read, execute and test the specification. A major output of this work was the creation of formal specifications of Arm's v8-A [165] and v8-M [166] processor architectures, covering the semantics of the most security sensitive parts of the processor: the memory and register protection mechanisms and the exception mechanisms that trigger transitions between different modes [132]. A notable contribution is the development and application of multiple approaches with which the specification is thoroughly tested. That broadly includes (1) testing using test suites (including ARM's Architecture Validition Suite hosting billions of instructions and random instruction sequences) and simulators [132], (2) formal validation of processors against the specification [140], and (3) formal validation of the specification itself [153]. ISA-Formal [140] (cited at (2) above) uses mechanical translation of ARM's Architecture Reference Manuals to Verilog allowing the use of commercial model-checkers to compare the specification against the Verilog implementation of five ARM processors currently in development. The exercise is found to be effective at finding micro-architecture specific bugs, and able to scale all the way from simple 3-stage microcontrollers, through superscalar in-order processors up to out-of-order processors. As cited at (3) above, Reid [153] further enhances the trust in the model by writing high-level properties about the specification and formally verifying that the specification satisfies those properties. Specifically, the properties[4] are derived by examination

[3]Indeed, the Sail ARMv8-A semantics is automatically generated from the ARM-internal specification of ARMv8-A [132] written in the ARM's architecture specification language, ASL [133], by using the ASL-to-Sail translator [128].

[4]The properties are related to additional security features applied to ARM's v8-M microcontroller specification involving exceptions, privilege and security.

of natural language text in the reference manual, by examining recently discovered specification bugs, and by discussion with the architects of the specification. The properties are then translated into verification conditions along with the specification (throughly tested as part of [140, 132]) to be checked by an SMT solver. To the best of our knowledge, no realistic architecture specification has been subjected to such an extensive testing and formal verification before. Unlike the approach of defining the semantics in a custom DSL, we chose to define our x86-64 semantics using $\mathbb{K}$ framework. The choice was motivated by the fact that we do not have to worry about generating the runtime and formal analysis tooling as they are auto-generated "correct-by-construction" from the semantics itself. This indeed saves a lot of engineering effort in implementing the tooling from scratch. However, we note our x86-64 model falls short of others [89, 153, 132] in two aspects: (1) we do not support system-level instructions, and (2) the simulation speed of our model, on an average over the supported instructions, is approximately 54 x86-64 instructions/second, which is not nearly as optimized as other notable semantics (e.g., Goel et al. [89] executing $\sim$ 3.3 million x86 instructions/second and Sail [153] executing MIPS and Arm at $500,000 - 1,500,000$ and $53,000$ instructions /second respectively). The interpreter currently in use by our semantics is implemented in Java and more performant back end are under development.

2.4 TRANSLATION VALIDATION

Pnueli *et al.* [167] proposed the idea of translation validation as a new approach to the verification of translators (compilers, code generators). The idea is: Instead of verifying the compiler itself, one constructs a validation tool that, after every run of the compiler, formally confirms that the target code produced in the run is a correct translation of the source program. One of the critical ingredients to drive the translation validation process involves setting up a “simulation relation”. A simulation relation is a proof (or witness) of the equivalence between two programs and is represented as a pair of correlation and invariants. Given two programs, the correlation is a pair of program PCs or locations in the two programs, and the invariants are predicates in terms of the program variables (i.e., states) expected to hold at these respective PCs. A simulation relation is valid if it can be proved inductively as follows: (1) Inductive case: For a valid simulation relation, the invariants at each correlated location should be provable from the invariants at the predecessor correlated locations, and (2) Base case: At the pair of entry points of the two programs, the invariants must be provable using the equivalence condition at the input. Using the invariants of the simulation relation, if one can prove the equivalence of the required observables at the exit location (pair of exits of two programs), then the programs can be concluded as equivalent.

Translation validation has been employed heavily in the field of compiler correctness [168, 169, 170, 171, 172, 173]. Necula [170] proposed a technique where each of the original and the optimized programs is firstly evaluated symbolically into a series of basic block transfer functions. Next, the basic block and variable correspondence is inferred by a scanning algorithm that traverses the source and target functions in parallel. This algorithm generates both a relation between program points and the accompanying constraints, between program variables and memory at the program point, using the previously generated transfer functions.

Hawblitzel et al. [174] use a translation validation approach to determine whether assembly code produced by different versions of the CLR JIT compiler are semantically equivalent and thus report mis-compilations when there are differences. The versions include those across a seven-month time period, across two architectures (x86 and ARM), and across optimizations levels. The underlying validator encodes each assembly method body into a procedure in the Boogie [175] programming language and then invokes the SymDiff symbolic differencing tool [176] to compare the Boogie encodings for semantic equivalence. For code with loops, the validator simply eliminates loops by unrolling them n (= 2) times, ignoring any behaviors past the n^{th} iteration.

The translation validation technique by Rival [177] provides a unifying framework for the certification of compilation and of compiled programs. Similarly to Necula's technique, the framework is based on a symbolic representation of the semantics of the programs. Rival's technique extracts basic block and variable correspondence from the standard debugging information if no optimizations are applied. However, when some optimizations are involved in the compilation, the optimizing phase has to be instrumented further to debug the optimized code and generate the correspondence between the original and the optimized programs.

During our initial attempt to solve the problem of translation validation of the lifting of x86-64 program, we tried to borrow insights from the above mentioned efforts. However, to be effective, we believe our validator should not instrument the lifter mainly because lifters in an early development phase are updated and improved at a frantic pace. Without instrumentation, such simulation relations can be inferred by collecting constraints from the input and output programs using symbolic-execution (as demonstrated in Necula's work [170]). First, in the context of translation validation of binary lifting, such inference is not straightforward mainly because the two program (x86-64 binary and lifted IR) are structurally very different with potentially different number of basic blocks. For example, instructions like `adcq` generate additional basic blocks upon lifting, which are not explicit in the binary program. Indeed, a similar challenge poses a hard requirement of branch equivalence in Necula's approach. Second, checking program equivalence, in general, is an undecidable problem and using symbolic execution is very expensive. Hence, any solution which can avoid such overhead is of great importance in serving a practical validation approach. Consequently, we decided to move away from simulation-based validation approaches.

Value-graph translation validation for LLVM has been performed previously in two independent efforts, LLVM-MD [171] and Peggy [172] and it is imperative to differentiate our approach of establishing equivalence between two LLVM IR programs from such existing, similar approaches for validating LLVM IR-to-IR optimization passes. Like our approach, they move away from simulation proofs, and instead use graph isomorphism techniques to prove equivalence. Both LLVM-MD [171] and Peggy [172] build graphs of expressions for each program, transform the graphs via a series of "expert-provided" rewrite rules, and check for equality. The rewrite-rules mimic various compiler-IR optimizations and hence the technique is precise when the output program is an optimization of the input program and the optimizations are captured by the rewrite rules.

Compared to these approaches, our canonicalizer is simpler, requires no additional implementation effort, re-uses off-the-shelf, well-tested compiler passes, and still proves to be very effective in reducing two programs to isomorphic graphs, as demonstrated by our evaluations (refer to section 4.5).

CHAPTER 3: FORMAL SEMANTICS OF x86-64 USER-LEVEL ISA

In this chapter, we will present our published contribution [86] of the most complete and thoroughly tested formal semantics of x86-64 assembly instructions to date. Our semantics faithfully formalizes all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture. This totals 3155 instruction variants, corresponding to 774 mnemonics. The semantics is fully executable and has been tested against more than 7,000 instruction-level test cases and the GCC torture test suite. This extensive testing paid off, revealing bugs in both the x86-64 reference manual and other existing semantics. We will also illustrate potential applications of our semantics in different formal analyses, such as symbolic execution, deductive verification, and translation validation, and discuss how it can be useful for processor verification.

The developed semantics is for the assembly language notation of the binary program and do not include a formal model of the binary instruction decoder. As a result, any binary analysis using our formalism requires, as a prerequisite, converting the binary to the supported mnemonic notation using an off-the-shelf disassembler. This limitation is eliminated by a follow-up work [117], not included in the current thesis presentation, where we formally specify an x86-64 instruction decoder. We note that all future references of x86-64 "program(s)" or "instructions(s)", in the context of the presented x86-64 model, are meant to refer to the "assembly language programs(s)" or "assembly instruction(s)".

3.1 APPROACH OVERVIEW

We employed the $\mathbb{K}$ framework [178] (Section 3.3.2) as our formalism medium to leverage its capability of deriving various correct-by-construction formal analysis tools directly from the language semantics. We took Heule *et al.* [87]'s semantics (Section 3.3.3) as our starting point to avoid duplicating the formalization effort. We made several corrections or improvements to this semantics, to improve both soundness and efficiency. We *automatically* translated their semantics into $\mathbb{K}$, and cross-checked the translated semantics against the original using an SMT solver. We *manually* specified the semantics of the remaining instructions faithfully consulting the Intel manual [84] to obtain the complete semantics. A manual specification may sound like a daunting effort at first, but the fact that (1) x86-64 is largely stable and changes slowly over time, and (2) the state-of-the-art synthesis techniques for language semantics (notably, Strata [87] and Hasabnis *et al.* [92, 93]) suffer from scalability and/or faithfulness issues (see Section 3.4.2 and Section 2.2 for details), makes the effort worth undertaking. Moreover, an important message of this work is that complete formal semantics of x86 is possible, and that is not only useful in itself but also to generate formal analysis tools.

Like closely related previous work [88, 87], we omit the relaxed memory model of x86-64 and thus the concurrency-related operations. Modelling concurrency is a complex but relatively orthogonal problem in the presence of small-step operational semantics, as shown in prior work [127, 126], where they have integrated their memory model with a small subset of 32-bit x86 instruction set. We believe that integrating such a memory model into our instruction semantics is a promising direction toward rigorously reasoning about real-world programs running on modern multiprocessors. We leave it for future work.

3.2 CHALLENGES IN FORMALIZING x86-64

Size and Complexity The x86-64 ISA has a large number of instructions, partly because of a large number of complex instructions and partly because it keeps most of the legacy and deprecated instructions ($\sim 336+$) for the sake of backwards compatibility. It consists of 996 mnemonics, and each mnemonic admits several variants, depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands.

Inconsistent Instruction Variants Some variants have divergent behaviors more than the difference of their type and size. For example, `movsd`, one of the 128-bit SSE instructions, has very different behaviors depending on whether the type of the source operand is register or memory; it clears the higher 64 bits of the target register only when the source type is memory. Indeed, we revealed bugs in other semantics due to their incorrect generalization of the variants' behavior (Details in Section 4.1, Instruction Variants).

Ambiguous Documentation The x86-64 reference manual informally explains the instruction behaviors, leaving certain details unspecified or ambiguous, which required us to consult with an actual processor implementation to clarify such details. Completely formalizing the vast number of instructions with carefully identifying all the corner cases from the informal document, thus, is highly non-trivial.

Undefined Behaviors The x86-64 standard also admits undefined behaviors that are implementation-dependent. Many instructions (32[1] out of 996 mnemonics) have undefined

[1]These numbers are obtained by parsing the official manual "Volume 2: Instruction Set Reference" and cross checked with projects [135, 179] investing similar efforts.

behaviors: their output values of the destination register or the `%rflags` register are undefined in certain cases. That is, the processor is free to choose any behavior in undefined cases.

Many existing semantics, however, simply "define" the undefined behaviors by following a specific behavior taken by a processor implementation. This approach is problematic because they do not capture all possible behaviors of different processor implementations. Indeed, we found discrepancies between existing semantics in specifying the undefined behaviors, where different semantics are valid only for different groups of processors. That is, such semantics are not adequate to formally reason about universal properties (e.g., portability) of a program that need to be satisfied for all standard-conforming processors. For example, the parity flag `%pf` is undefined in the logical-and-not instruction **`andn`**, where the processor implementation is allowed to either update the flag value (to 0 or 1), or keep the previous value. We found, e.g., that Remill [49] updates the flag with 0, whereas Radare [35] keeps it unmodified. Identifying and faithfully specifying all of the undefined behaviors, thus, are desirable but challenging.

In our semantics, we faithfully modeled the undefined value as a unique symbol (called `undef`) whose value is nondeterministically decided each time within the proper range. These nondeterministic values are enough to capture and formally reason about all possible behaviors of the instructions for different processors (and even any future, standard-conforming processor). While performing instruction-level testing (Section 3.5), we consider the `undef` symbol to be matched with any concrete value provided by the hardware, so that we can test the instructions modulo the undefined behaviors.

3.3 PRELIMINARIES

First, we briefly explain pieces of x86-64 ISA necessary for our presentation. We also talk about $\mathbb{K}$ framework, a semantics engineering tool which we chose to formalize our semantics into and Strata which we use for our baseline semantics.

3.3.1 x86-64 Instruction Set Architecture

x86-64 is the 64-bit extension of x86, a family of backward-compatible ISAs. We briefly explain some details of the architecture.

x86-64 has the sixteen 64-bit general purpose registers (`%rax`–`%rdx`, `%rsi`, `%rdi`, `%rsp`, `%rbp`, `%r8`–`%r15`), and the two 64-bit special registers (`%rip` and `%rflags`). The lower 32-, 16- and 8-bit portions of the register are referenced by the sub-register variants, e.g.,

`%eax`, `%ax`, and `%al` for `%rax`, respectively. The Haswell x86-64 ISA additionally has sixteen 256-bit SIMD registers (`%ymm0`–`%ymm15`) along with the lower 128-bit sub-register variants (`%xmm0`–`%xmm15`).

The `%rflags` register stores the current state of the processor. Specifically, for example, the status flags such as the carry flag (`cf`), the parity flag (`pf`), the adjust flag (`af`), the zero flag (`zf`), and the sign flag (`sf`) are stored in `%rflags`. These status flags are set according to the result of arithmetic and logical instructions. These status flags used mostly in user-level x86-64 programs, and updated by arithmetic-logical instructions according to the result of the operation. Many control transfer instructions, e.g. **`jz`**, **`jge`** etc., are performed based on the values of these flags.

x86-64 ISA supports the addressing mode expressions that calculate a memory address to be read or written to. The addressing modes are used as the source or the destination of instructions that access the memory. The addressing mode expressions can be generalized as: `base` + `index` × `stride` + `offset`. In the assembly code, for example, **`-4(%rax, %rbx, 8)`** denotes the address mode expression "`%rax` + `%rbx` × 8 − 4".

The ISA has three types of instructions depending on the types of their operands: register instructions (with only register operands), memory instructions (with address mode operands), and immediate instructions (with constant operands). The same mnemonic can be used for the different types of instructions. For example, the mnemonic **`add`** can be used for the register instructions (e.g., **`addq %rax, %rbx`**[2]), the memory instructions (e.g., **`addq -4(%rax), %rbx`**), and the immediate instructions (e.g., **`addq $1, %rbx`**).

3.3.2 K Framework

$\mathbb{K}$ [178] is a framework for defining formal language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional cost. Using the interpreter, one can test their semantics immediately, which significantly increases the efficiency of semantics developments. Furthermore, the formal analysis tools facilitate formal reasoning about the given language semantics. This helps in terms of both applicability of the semantics and engineering the semantics itself.

We refer the reader to [178, 181] for details. In a nutshell, in $\mathbb{K}$, a language syntax is given using conventional Backus-Naur Form (BNF). A language semantics is given as a parametric transition system, specifically a set of reduction rules over configurations, which $\mathbb{K}$

[2]Throughout the presentation we will be using the AT&T assembly syntax [180] where the destination operand comes after source operands.

makes heavy use of.A configuration can be regarded as a snapshot of the program execution: it includes the remaining program, together with all the necessary semantic information to execute it. $\mathbb{K}$ configurations are organized as nested cell structures, whose leaves hold basic data organized in lists, maps, sets, etc. Cells can be referred to by their name; their order is irrelevant. In the contxt of x86-64, each cell in configuration represents a semantic component, such as the memory or the registers. A special cell, named `k`, contains a list of computations to be executed. A computation is essentially a program fragment, while the original program is flattened into a sequence of computations. A rule describes a one-step transition between configurations, giving semantics to language constructs. Rules are modular; they mention only relevant cells that are needed in each rule, making many rules far more concise and easy to read than in some other formalisms.

One of the most appealing aspects of K is its modularity. It is very rarely the case that one needs to touch existing rules in order to add a new feature to the language. This is achieved by structuring the configuration as nested cells and by requiring the language designer to mention only the cells that are needed in each rule, and only the needed portions of those cells. This modularity makes for compact and human readable semantics, and also helps with the overall effectiveness of the semantics development.

3.3.3 Strata Project

Strata [87] automatically synthesized formal semantics of 1905 instruction variants (representing 466 unique mnemonics) of the x86-64 Haswell ISA. The algorithm to learn the formal semantics of an instruction, say `IS`, starts with a small set of instructions, called base set `B`, whose semantics are known and trusted; a set of test inputs `T`, and the output behavior of `IS` obtained by executing `IS` on `T`. Then Stoke [135] is used to synthesize instruction sequences which contain instructions from `B` and match the behavior of `IS` for all test cases in `T`. Given two such generated instruction sequences `IS` and `IS`$'$, their equivalence is decided using an SMT solver and the trusted and known semantics from the base set. If the two sequences are semantically distinct, then the model produced by the SMT solver is used to obtain an input `t` that distinguishes `IS` and `IS`$'$, and `t` is added to `T`. This process of synthesizing instruction sequence candidates and accepting or rejecting them based on equivalence checking with previous candidates, is repeated until a threshold is reached, which in their implementation is based on the number of accepted instruction sequences.

Using the above technique, they first came up with the semantics of 692 register and 120 immediate instructions. Then they used "generalization" of the register instructions to get a total support count of 1905. Generalization is based on their hypothesis that the memory or immediate variants will behave identically with corresponding register variant (other than where the inputs come from) and hence can use the same formula as register variants. They validate this hypothesis using random testing.

For each instruction, Strata manifested its semantics in terms of two related artifacts. The first artifact is an instruction sequence and the second is a set of SMT formulas in the bit-vector theory, one for each output register. The second is obtained by symbolically executing the first.

3.4 FORMALIZATION OF x86-64 SEMANTICS

This section presents how we get the complete semantics of all the user-level instructions. Section 3.4.1 details the scope of our work. Section 4.1 mentions how we leverage the information available in Strata, our baseline semantics. Section 3.4.3 explains how we formalize our model in $\mathbb{K}$.

3.4.1 Scope of the Work

We support all but a few non-deprecated user-level instructions. The support includes 3155 total variants of the Haswell x86-64 ISA (representing 774 out of 996 unique mnemonics). The entire implementation took 8 man-months, with the lead author having prior experience in binary decompilation and strong familiarity with the x86-64 architecture and documentation. Below is a summary of the instruction categories that we support.

- *General-Purpose:* These implement data-movement, arithmetic, logic, control-flow, string operations (including fast- and repeated- string operations).
- *Streaming SIMD Extensions (SSE) & subsequent extensions (SSE-2, SSE-3, SSE-4.1, SSE-4.2):* Instructions in this category operate on integer, string or floating-point values stored in 128-bit xmm registers. Among other things, the category features instructions related to conversions between integer and floating-point values with selectable rounding mode, and string processing.
- *Advanced Vector Extensions (AVX) & subsequent extensions (Fused-Multiply-Add (FMA) & AVX2):* These instructions operate on integer or floating-point values stored in 256-bit ymm registers; a majority of which are promoted from SSE instruction sets. Additionally, the category features enhanced functionalities specific to AVX & AVX2,

like broadcast/permute, vector shift, and non-contiguous data fetch operations on data elements.

- *16-bit Floating-Point Conversion (or F16C):* These instructions implement conversions between single-precision (32-bit) and half-precision (16-bit) floating-point values.

Instructions which are *not included* in the current scope of work are:

- System-level instructions, which are related to the operating system, protection levels, I/O, cache lines, and other supervisor instructions;
- x87 & MMX instructions, consisting of legacy floating-point and vector operations, respectively, which are now superseded by SSE;
- Concurrency-related operations, including atomic operations and fences; and
- Cryptography instructions, which support cryptographic processing specified by Advanced Encryption Standard (AES).

We note that while there is no inherent limitation in supporting the above instructions with our approach, the system-level instructions require to formulate an abstraction of different architectures and operating systems, which is a significant effort that is orthogonal to the presented effort of formalizing the user-level instructions. Nevertheless, K framework makes it easy to add additional state components without modifying rules for operations that do not require those components. We expect our approach to work equally well compared with existing approaches, such as [89], which implements a subset of system-level instructions. On the other hand, the cryptography instructions are omitted mainly because they are not given a high priority.

3.4.2 Overview of the Approach

Briefly, our approach is as follows. We first defined the machine configuration and underlying infrastructure in the $\mathbb{K}$ framework, in order to define, execute and test the x86-64 semantics. To leverage previous work as much as possible, we took the semantic rules of all the instructions supported in Strata, which amounts to about 60% of the instructions in scope, in the form of SMT formulas. We corrected, improved or simplified many of the baseline rules. We then translated these SMT formulas from Strata into $\mathbb{K}$ rules using a script, and tested the resulting rules by comparing with the Strata rules using Z3. These steps give us a validated initial set of semantic rules in $\mathbb{K}$ for about 60% of the target instructions (our "baseline" set).

We attempted to extend the stratification approach in Strata to define additional rules automatically, in two ways: (i) augmenting their base set B, and (ii) constraining the search space manually using knowledge of instruction behaviors. Both these attempts failed; they worked only for a few instructions, but in general, we found them to be impractical. Specifically, we added 58 base instructions to the base set, and learned the semantics of 70 new instructions, which are variants of the added instructions, in 20 minutes, but no more even after we kept running for two days. Also, we tried constraining the search space by manually populating it with relevant instructions. The lesson we learned from these experiments is, getting the right set of base instructions or a constrained search space for a complex instruction need an insight about the semantics of that instruction itself. We found that the effort to extract such information from the manual is about the same as manually defining that instruction.

We then manually added $\mathbb{K}$ rules for the remaining 40% of the target instructions by systematically translating their description of the Intel manual into $\mathbb{K}$ rules, in some cases cross-referencing against semantics available in Stoke. The outcome was a complete formal specification of user-level x86-64 in $\mathbb{K}$.

We validated this semantics in three ways, as described in Section 3.5. First, we use the $\mathbb{K}$ interpreter to execute the semantics of *each* instruction for 7,000+ test inputs (each input is a processor state configuration) and compared the output directly with the hardware behavior for the same instruction. Second, we repeated this experiment using the applicable programs in the GCC C-torture tests [107]. Third, we compared against the semantics defined in the Stoke project for about 330 instructions that were omitted in Strata (thus not included in our baseline), using an SMT solver.

These validation experiments uncovered bugs in the Intel manual, in Strata's simplification rules, and in the Stoke semantics. All these bugs were reported to the authors, and most have been acknowledged and some have been fixed. The details are in Section 3.5.

3.4.3 Program Configuration

Defining a language semantics in $\mathbb{K}$ requires defining the program configuration, the semantics of how programs are evaluated (i.e., the execution environment), and the semantics of the statements or instructions. We begin with the configuration.

The $\mathbb{K}$ configuration of a running x86-64 program is shown in Figure 3.1. The cells are represented using angle brackets. The outer $\top$ cell contains the cells used during program

$$\Big\langle \langle \texttt{K} \rangle_{\textsf{k}} \;\; \langle \texttt{ID}_{\texttt{regname}} \mapsto \texttt{Value} \rangle_{\textsf{regstate}} \;\; \langle \texttt{Address} \mapsto \texttt{Value} \rangle_{\textsf{memstate}} \;\; \cdots \Big\rangle_{\textsf{T}}$$

Figure 3.1: Program Configuration

evaluation. The inner k cell contains a list of computations to be executed. Below we describe the two other inner cells.[3]

Register State. The regstate cell contains a map from registers or flag names to values. Note that, all the values or addresses, stored in registers, memory (described next) or flags, are represented as bit-vectors which are depicted as V_W, and interpreted as a bit-vector of size W and value V. The register names include the sixteen general purpose registers, %rip, and the sixteen SIMD registers. The value mapped to a register name is a 64-width bit-vector (or a 256-width one for the SIMD registers). Values for sub-register variants are derived from the register values by extracting the relevant bits. We store individual flag names (mapped to a bit-vector value of width 1) as opposed to a 64-bit rflags register. Every access (read/write) of %rflags retrieves the entries in the regstate map for the individual flags.

Memory State. Our memory model is inspired by previous efforts [91, 158]. The memstate cell is a map from 64-bit addresses to bytes, which specifies the byte-addressable memory[4], but our implementation is flexible enough to use alternative memory representations with addressing of 2-byte or 4-byte quantities. Our memory layout is "flat", in which all available memory locations can be addressed, but we do have logical partitions of the memory into sections like code, data and stack[5]. The following is an example snapshot of a memory state, holding a 4-bytes integer value 65535:

$$\left\langle \begin{array}{ll} 4 \mapsto \texttt{byte}(0,\ 65535_{32}) & 5 \mapsto \texttt{byte}(1,\ 65535_{32}) \\ 6 \mapsto \texttt{byte}(2,\ 65535_{32}) & 7 \mapsto \texttt{byte}(3,\ 65535_{32}) \end{array} \right\rangle_{\textsf{memstate}}$$

Here the memory address 4 stores the 0th byte of the bit-vector 65535_{32}, the address 5 stores the 1st byte, and so on. When memory is read, requested bytes are aggregated according to the size of the memory access.

[3]We omit other auxiliary cells (marked by "$\cdots$") for the simplicity of the presentation.

[4]Byte-addressability allows the model to specify both aligned and unaligned accesses in the same principle.

[5]These abstractions help in logical partitioning of the $\mathbb{K}$ memstate cell into nested cells like text, data and stack. For example, while executing x86-64 programs, the code (resp., data) section of the binary code is stored in text (resp., data) cells, and the local variables are allocated in stack cell. We note that these abstractions are purely cosmetic and can be skipped without affecting the faithfulness of the model.

3.4.4 Semantics of Execution Environment

We now give the reader a flavor of our semantics, by discussing a few of the roughly 5,200 rules[6] that we defined to model the entire semantics. We first explain the semantics of the execution environment, which involves all the machinery used for executing x86-64 programs. We will explain the semantics of individual instructions in the next section.

The execution of an x86-64 program begins with initialization of the configuration with the following contents of the `k` cell.

```
<k> $PGM:Instructions ↷ fetch </k>
```

The symbol ↷ is used to separate the computations in the `k` cell and "`:T`" to represent the type of a term.

Concisely, the semantics of execution of an x86-64 program involves initializing the memory by reading the program instructions (`$PGM`), from the `k` cell, one at a time until all the instructions are loaded in memory. The memory-loaded instructions are then fetched one at a time, using the `fetch` computation, to get executed. The instruction to be executed next is pointed to by the instruction pointer register `%rip`.

Next, we describe the rule applied to initialize memory with instructions one at a time.

```
rule  <k> OpC:Mnemonic OpR:Operands ⇒ . ...</k>
 <memstate> M:Map ⇒ M[L ← (OpC OpR)] </memstate>
 <nextloc> L:Address ⇒ L + instrSize(OpC OpR) </nextloc>
```

The `k` cell contains the instruction to be processed next. `Mnemonic` and `Operands` denote the types of the terms used to represent an instruction. The '⇒' symbol represents a reduction (i.e., a transition relation). A cell without the '⇒' symbol means that it is read but not changed by the rule. 𝕂 allows us to use "." to represent an empty computation and "..." to match the portions of a cell that are neither read nor written by the rule. The above rule essentially stores each instruction in memory, which is modeled as a map, at an address `L` given by the `nextloc`[7] cell. Subsequently, the `nextloc` cell gets updated to an appropriate address used for storing the next instruction. Once the entire program is loaded, the fetch-and-execute cycle starts, which is realized by the following rule:

```
rule <k> fetch ⇒ exec(OpC OpR) ↷ fetch ... </k>
  <memstate>... L ↦ (OpC OpR) ...</memstate>
  <regstate>... "RIP" ↦ ( L ⇒ L + instrSize(OpC OpR))  ...</regstate>
```

[6]Each rule is 17 LOC on average, and the total size is 15 KB of text.

[7]The `nextloc` cell is a auxiliary cell that holds the next memory location to store an instruction, which we omit in the program configuration (Figure 3.1) for the sake of simplicity.

The rule above says that if the next thing to be evaluated is a fetch computation (referred in the rule as `fetch`), then one should match `%rip` in the environment to find its value `L` in `regstate`, where `L` is matched in `memstate` to find the mapped instruction. The mapped instruction is then put at the head of the `k` cell to be computed next, using a rule `exec` for execution (defined later), along with the `fetch` computation to be executed in order. The rule also updates the value of `%rip` to point to the following instruction. The execution will be terminated when there is no instruction stored in the memory at the address pointed to by `%rip`.[8]

3.4.5 Semantics of Individual Instructions

Here we explain how we define the semantics of an instruction in $\mathbb{K}$ using a running example of logical-and-not **`andnq -4(%rsp), %rbx, %rax`**, which performs a bitwise logical AND of inverted source register operand (`%rbx`) with the source memory operand (`-4(%rsp)`) and writes the result to destination register `%rax`. Additionally the instruction affects all the 6 status flags (`%sf`, `%zf`, `%of`, `%cf`, `%af` and `%pf`).

The semantics of most of the instructions can be modeled broadly in 3 phases: (1) read the data from source operand(s), which could be a register, memory or constant value; (2) operate on the data based on the mnemonic; and (3) write the result(s) to destination operand(s), which could be a register or memory. An instruction may exercise some or all of the above phases.

Read from Source Operand(s) Instruction in the running example reads from register (`%rbx`) and memory (`-4(%rsp)`) operands. A read from register is modeled as a lookup with register name in the `regstate` map and subsequent read of the mapped value or, for a sub-register, a portion of it. The semantics of register read can be defined as:

```
rule <k> getRegisterVal(R:R64) ⇒ BVr ...</k>
  <regstate>... R ↦ BVr ...</regstate>
```

In the context of the running example, this rule is applied when the current computation (at top of the `k` cell) is a 64-bit register lookup, appeared as `getRegisterVal(%rbx)`, and `regstate` contains a register with name “`RBX`”. This rule resolves the register lookup to the mapped bit-vector value $\texttt{BV}_r$ (or $\texttt{BV}_{\texttt{RBX}}$ for the running example).

[8]While initializing the stack section of memory, we store an invalid address just before the entry-point function as return address. When the entry point function returned, the invalid return address is popped out of the stack and stored in `%rip` leading to program termination.

A read from memory involves computing the effective address in the memory, looking-up that address in memory, and reading requested bytes from memory if the memory access is within allowed range. The following rule is applied to compute the effective address:

```
rule <k>
 (Offset:Int (R:R64)):Mem ⇒ ( Offset64  + BVr):Address
...</k>
  <regstate> ... R ↦ BVr ... </regstate>
```

The term to the left of ⇒ shows the memory addressing expression, of type `Mem`, at the top of `k` cell, which gets reduced to an effective memory address (or EA). The EA for the memory operand used in the running example is (64'-4 + $\texttt{BV}_{\texttt{rsp}}$) and is used to do memory read access. The rule for memory read access is responsible to read a memory value of requested number of bits (64-bits for the current example) starting from the EA.

Operate on Data The rules for operating on operands will be different for each instruction based on the mnemonic. For example, the mnemonic **`andnq`** requires logical-and-not operation to be computed on the operands.

Write to Destination Operand(s) The example instruction writes the result to a destination register `%rax`. Also, the flags `sf` and `zf` are updated based on the result; `of` and `cf` are cleared, and `af` and `pf` are undefined. The rule, as shown below, realizes the destination write operation, where $\texttt{memVal}_{64}$ and $\texttt{BV}_{\texttt{r}}$ represents the 64-bit data values evaluated using the respective rules for reading register and memory operands (mentioned above).

```
rule <k>
  exec(andnq memVal64, BVr, R:R64) ⇒ .
...</k>
  <regstate>
    "R"  ↦ _ ⇒ (~BVr & MemVal64)
    "SF" ↦ (~BVr & MemVal64)[63:63]
    "ZF" ↦ (~BVr & MemVal64) = 064  ? 11 : 01
    "OF" ↦ 01
    "CF" ↦ 01
    "AF" ↦ undef // af and
    "PF" ↦ undef // pf are undefined.
  ...</regstate>
```

The operator "[i:j]" extracts bits i down to j from a bit-vector of size n, yielding another bit-vector of size i - j + 1, assuming that n > i ≥ j ≥ 0. The operator "&" implements bit-wise "and" operation. The rule associated with memory write is similar to that for memory read and is skipped here.

A x86-64 program is modeled as a list of instructions and its semantics is given by composing the semantics of its constituents.

3.4.6 Constructing the x86-64 Semantics

Systematic Translation of Strata Rules to K As mentioned in the introduction, we leverage the Strata [87] semantics to develop our complete semantics, to minimize the overall effort. We systematically translated their semantics into $\mathbb{K}$. Specifically, Strata offers the semantics of 1905 instruction variants as SMT formulas specifying the behavior of output registers. For each instruction, we converted the SMT formulas that Strata provides to a $\mathbb{K}$ specification using a simple script ($\sim$500 LOC). The ease of this porting is facilitated by the fact that SMT-LIB expressions use s-expression (or "symbolic-expression") notation [182] which eases the parsing effort significantly.

To validate the translation, we generated SMT formulas from the translated $\mathbb{K}$ specifications (using APIs provided by the $\mathbb{K}$ framework), and use the Z3 SMT solver to check their equivalence to the corresponding formulas provided by Strata. While translating and validating their semantics, we found various issues that we had to fix to establish our baseline semantics. Below we describe the issues we found in Strata.

Status Flags We found that Strata omitted to specify the `%af` flag behaviors, as the flag is not commonly used. However, we faithfully specified the semantics of all the status flags in the `%rflags` register, even if some of them are not commonly used, since they may affect the overall program's behavior in some tricky cases, and we do not want to miss any of such details when formally reasoning about the x86-64 programs.

Instruction Variants Strata essentially provides the semantics of the register instructions, assuming that the semantics of the memory and immediate instruction variants can be obtained by generalizing the register instructions[9]. However, we found that certain memory instructions cannot be inferred by simply generalizing their corresponding register instructions. For example, for **`movsd`**, one of the 128-bit SSE instructions, its register variant has quite different semantics from the memory variant. Figure 3.2 shows their pseudo-code semantics, where only the memory instruction clears the higher 64 bits of the destination register. This behavior cannot be inferred from the register variant that does not touch the

[9]Generalization is based on a hypothesis that the memory or immediate variants will behave identically, on their operands, with corresponding register variant.

Semantics of Register Variant (`movsd %xmm1, %xmm`)	Semantics of Memory Variant (`movsd (%rax), %xmm0`)
`S1.` XMM0[63:0] $\leftarrow$ XMM1[63:0]	`S1.` XMM0[63:0] $\leftarrow$ MEM_ADDR[63:0]
`S2.` XMM0[127:64] (Unmodified)	`S2.` XMM0[127:64] $\leftarrow$ 0

Figure 3.2: Pseudo-code semantics of the register and memory variants of `movsd`

higher bits at all. We found that another 128-bit SSE instruction, `movss`, has the same generalization issue. For the other instructions, we obtained the memory and immediate variants by generalizing the register variants, and validated the generalization by co-simulating the inferred semantics against a processor.

Immediate Instruction Variants There are 118 immediate instruction variants (over the 8-bit constants) that do not have corresponding register instructions. For those immediate instructions, Strata provides the instruction semantics for each individual constant, resulting in 30,208 ($= 118 \times 256$) formulae[10] for the immediate instructions' semantics. We generalized the set of formulae for each immediate instruction into a single semantic rule. We validated our generalization by cross-checking the generalized semantics with the original using the SMT solver.

Formula Simplification Due to the nature of the stratification, Strata provides complex formulae for certain instructions. We simplified those complex formulae by either applying additional 13+ simplification rules (Figure 3.3 mentions some of those) than what Strata originally had or manually translating into simpler ones. Then we validated the simplification by checking the equivalence between them using the SMT solver. For example, the original Strata-provided formula for `shrxl %edx, %ecx, %ebx` consists of 8971 terms (including the operator symbols), but we could simplify it to a formula consisting of only 7 terms.

3.5 VALIDATION OF SEMANTICS

A formal semantics is of limited use if one cannot generate confidence in its correctness. In this section, we describe how we establish that confidence in our model.

[10]Indeed, Strata explicitly provides only 19,783 formulae by randomly sampling ~168 constants out of 256, in average, for each immediate instruction, assuming that the remaining 10,425 formulae can be inferred.

```
/* Assume
**   A, B, C, D are symbolic bit-vector values of width W,
**   N_W  denotes a constant bit-vector value of width W & integer value N,
**   Cond is a symbolic boolean value,
**   I, J & K are symbolic integers s.t. K ≥ J ≥ I ≥ 0.
**
**   A ∘ B  denote bit-vector concatenation.
**   A[J:I] denotes bitvector-extract operation (assuming that J & I are within A's bounds).
**   '+'    denote addition over bit-vectors.
**   '⊗' denote  any of xor, or, add operators.
*/

 /* Eliminate redundant uninterpreted functions */
 • add_double(0_W, A)  ≡  A if A ≡ concat(0_X, N_Y) s.t.  X + Y = W

 /* Distribute over if-then-else */
 • (Cond ?  A : B) [J:I] ≡ (Cond ?  A[J:I] : B[J:I])
 • X ∘ (Cond ?  A : B) ≡ (Cond ?  X ∘ A : X ∘ B)
 • (Cond ?  A : B) ⊗ (Cond ?  C : D) ≡ (Cond ?  A ⊗ B : C ⊗ D)

 /* Distribute extract over addition */
 • (A + B) [J:0] ≡ A[J:0] + B[J:0]

 /* Merge consecutive  concatenations */
 • (A[K][J] ∘ (A[J][I] ∘ C)) ≡ A[K:I] ∘ C
```

Figure 3.3: Additional simplification rules over bit-vector logic

3.5.1 Co-Simulations against Hardware

One way of establishing trust in a specification is by testing specifications against existing implementations [128, 154, 151, 88, 132, 183, 87]. Empowered by the fact that we can directly execute the semantics using the $\mathbb{K}$ framework, we validated our model by co-simulating it against a real machine. During co-simulation, we execute a machine program on the processor as well as on our $\mathbb{K}$ model and compare the execution results after every instruction. In this work, we co-simulated our model against two Intel implementations that were available to the authors at the time of writing: "Intel Xeon CPU E3-1505M v6" and "Intel Xeon CPU E5-2640 v4". We admit that testing the model against other hardwares (such as AMD) would contribute to more thorough validation of our model, having the potential of revealing flaws in those implementations and/or additional imperfections in the manual as well, which we leave as future work.

We first describe our test-infrastructure and then talk about individual validation experiments and results.

Test Harness During co-simulations, we need to make sure that the program must be instrumented similarly both on our model and the real hardware. We use the GNU Debugger [184] to instrument programs on hardware. We developed instrumentation tools based on $\mathbb{K}$ framework to gain similar capabilities for our model. Using these tools we can record the output state (including memory) after the execution of each instruction. To facilitate debugging, in the event when the output states do not match, we developed a tool which points to the first instant when the output states diverge and this saves debugging time.

The co-simulation experiments are done in the following two phases: (1) Instruction level validation: testing individual instructions, and (2) Program level validation: testing a combination of instructions as a part of real-world programs.

Instruction Level Validation The goal here is to execute individual instructions both on hardware and our model using test inputs and then compare the output states.

$\mathbb{K}$ already has matured library support for bit-vector, integer and floating-point theories. We use bit-vectors to implement the values stored in registers or memory. Depending upon an instruction mnemonic, these values can be interpreted as integers (signed/unsigned) or floating-point values (with various precisions). We augmented the library support in $\mathbb{K}$ framework to interpret these bit-vectors accordingly. With that support, we can execute and hence test instructions implementing various floating-point operations including conversions (to and from integer/floating-point values) with selectable rounding modes (e.g. Nearest, +Inf, -Inf and Truncate).

Test Inputs A test input is a CPU state which includes values for all registers, flags and memory. Our test input set contains more than $7,000$ inputs, obtained from the following sources:

1. In section 3.3.3, we mentioned that Strata starts its algorithm with a set of test inputs which keeps on augmenting itself during the process of stratification. We used the final augmented test-suite of 6630 test inputs,
2. While testing instructions implementing floating-point operations, we found that many of the test inputs are representing a NaN or Infinity and it makes no sense to test with such instances. We did our best effort by manually generating more than 100 unique floating-point values by consulting the IEEE floating-point arithmetic standard [185],
3. We used the ($\sim$100) test-inputs offered by Remill [49], and
4. We manually implemented a regression test-suite worth of around 200 test-inputs which we accumulated over the course of the project.

The above test-inputs are publicly available at [186, 187]. Note that, each instruction semantics consists of one or more semantic rules, where those rules cover different cases of the instruction behaviors (including the undefined ones). We ensure that our test inputs are sufficient enough to trigger all of the semantic rules, achieving the full "semantic-rule" coverage.

Results For each immediate instruction with a constant operand of size 8-bits, we tested all the 256 variants of the instruction using the above set of test inputs. There are 62 immediate instructions with a constant operand width larger than 8-bits. Testing with all possible values of the constant (which could be 2^{32} for a 32-bit constant) is impractical, so we limited the constant operand to the first 256 values and other interesting values like all ones, setting or resetting the bits at the byte/word/quad-word boundaries etc.

Our current implementation of the fused-multiply-add operation[11] incorrectly rounds the operation twice (after multiplication and addition) as opposed to once. As a result, we encountered floating-point precision issues while testing instructions implementing those operations (for example **`vfmadd132pd`**). This is a limitation of the underlying $\mathbb{K}$ library and more details about this limitation can be found in Section 3.7.

While performing the validation tests, we encountered various cases where the output state obtained by executing the semantics on our model does not agree with that of the hardware execution. The instruction semantics in our model is either based on the Strata project (for the part we borrowed) or on the Intel manual. A difference in the output state could mean a bug in Strata's instruction semantics or in our interpretation of the Intel manual or in the Intel manual itself. We found many bugs in our interpretation which we fixed, but in other cases, we found issues in Intel manual and Strata project.

Inconsistencies Found in the Intel Manual Here are inconsistencies found during development and testing. According to the manual, the semantics of **`vpsravd %xmm3, %xmm2, %xmm1`** seems to depend on the lower 100 bits of `%xmm3`, whereas the actual hardware execution suggests that it should depend on the lower 128 bits. Similar inconsistencies are found in instructions with mnemonics **`vpsllvd`**, **`vpsllvq`**, **`vpsravd`**. Also, we found misleading typos related to instructions with opcodes **`vpsravw`**, **`vpsravd`**, **`vpsravq`**, **`packsswb`**. All these findings were reported and acknowledged by Intel as issues in the manual [108] (Volume 2: March 2018) and are fixed in the latest revision.

[11]According to the standard IEEE-754-2008 [185] (Definition 2.1.28), the operation `fused-multiply-add(x, y, z)` computes $x \times y + z$ as if with unbounded range and precision, rounding only once to the destination format.

Inconsistencies Found in Strata's Simplification Rules While testing the instructions specifications borrowed from Strata, we found inconsistent behaviors with the actual hardware. Moreover, the inconsistencies were discovered in the formulas of floating-point instructions. This is not surprising because Strata models the floating-point instructions as uninterpreted functions which cannot be executed or tested on hardware. Their semantics are executable in our definition though, and thus we were able to test them thoroughly. Note that Strata generates the formulas for these instructions by symbolically executing the corresponding learned instruction sequences followed by a formula simplification pass. Therefore, errors in those formulas can be due to bugs either in the symbolic execution engine or in the simplification stage. Our testing shows that the second is true with the following evidence. The simplification rule `add_double(A, 0) == A` does not hold for `A = ` -0.0. Same for `add_single`. These were reported [188]. Also, the simplification rule `sub_double(A, A) == 0` does not hold for `A = ` NaN. Same is true for `sub_single`. We found this bug in the branch of Stoke which is used in Strata. But this has been already fixed in the latest Stoke branch.

Program Level Validation The goal here is to test the combination of instructions as part of real-world programs and we chose to use GCC C-torture tests [107] for this purpose. Specifically, we used the tests inside the "testsuite/gcc.c-torture/execute" directory for GCC version 8.1.0. There are originally 1576 tests, which we compiled using the GCC switches "`-O0 -march=haswell -S -mlong-double-64 -mno-80387`". The last two switches avoid generating x87 instructions that are not in the scope of work. We had to exclude 6 programs containing system-level instruction **`prefetchnta`**, which require modeling caches, which we currently do not support. Many test-cases involve C-library functions, related to I/O operations (e.g., `printf`, `scanf`) and heap allocation (e.g., `malloc`, `free`), which we modeled in $\mathbb{K}$. We implemented the I/O related library functions, covering most of the their documented functionality, using $\mathbb{K}$ buitin I/O operators and the heap allocation related functions on top of our memory model. Note that most of such library functions use systems calls (e.g., open, write, sbrk etc.) which we do not support. As a consequence, we cannot execute the native x86-64 implementation of those library functions. Hence, we implemented simpler alternative models of those. As our support of C-library functions is not exhaustive, we have to exclude 22 programs containing un-supported functions like `vfprintf` and `vsprintf`, which we plan to support in future. This brings us to a grand total of 1548 viable tests, which are all tested. Out of those, we found that there are 293 cases where floating-point instructions are used covering 35 unique floating-point operations. Moreover, all the test-cases together cover about 963 instruction variants, covering 30% of our supported instructions. As before, we executed each program on the processor as well as on our model and compared the output state after every instruction, which matches in all the cases[12].

3.5.2 Comparing with Stoke

Stoke [135][13] contains manually written semantics for ∼1764 x86-64 instruction variants, a large fraction (81%) of which is also supported by Strata. The remaining fraction is exclusive to Stoke. Comparing with Stoke provides an additional crosscheck on our model. Moreover, these manually written formulas are based on a similar model of the CPU state to ours, which makes it easier to compare them against ours by using an SMT solver. While doing so we found inconsistencies between the two formalisms in a total of 16 mnemonics (42 instruction variants), and after careful analysis, identified these as errors in the Stoke specification of instruction semantics, as follows.

Inconsistencies Found in Stoke

1. For instructions like **`addsubpd %xmm1, %xmm2`** , the order of addition and subtraction specified by Stoke is opposite to the one specified in the Intel Manual. Same is true with the mnemonic **`addsubps`**. (Found in 12 instruction variants.)

2. The instruction **`pslld %xmm1, %xmm2`** implements a logical left shift of packed data by a count specified in `%xmm1`. Stoke's specification vectorized the operand `%xmm1` which is incorrect according to the manual. Similar issues were found in instructions implementing the logical right shift operations on packed data. (Found in 18 instructions.)

3. Instructions **`cvtsi2sdl %eax, %xmm1`** & **`vcvtsi2sdl %eax, %xmm0, %xmm1`** are respectively SSE- and AVX-versions of the instruction to convert a double-word (32-bit) integer to a scalar single-precision floating-point value. According to the manual, in the AVX-version, the destination bits $127 - 64$ of the register `%xmm1` are updated to the corresponding bits in the first source operand `%xmm0`. This is in contrast to the SSE-version of the instruction where the destination bits $127 - 64$ should remain unmodified. Stoke specifies the semantics of the AVX-version similar to the SSE-version, which is incorrect. (Found in 4 instruction variants.)

[12]Note that none of test-cases include floating-point instructions implementing fused-multiply-addition, which we already acknowledged to have precision issues.

[13]Recall that Stoke is a stochastic super-optimizer leveraged by Strata for stochastic search.

4. Some instructions, like `imulb %al`, which drive flag registers to an undefined state are not modeled correctly in Stoke. (found in 8 instruction variants)

All these errors were reported and confirmed [109, 110].

3.6 APPLICATIONS

In this section, we illustrate a few applications of our formal semantics, in addition to the reference model mentioned in the previous section. Our goal here is to explain that our semantics can be used for formal reasoning of x86-64 programs for a wide variety of purposes. For this reason, the applications are illustrative only, not meant to serve as a comprehensive evaluation or make any claim of scalability. Moreover, the reported performance of the applications is not optimized[14], and there is room for improvement, e.g., by providing custom abstractions and lemmas specific to x86-64, similarly to [189]. However, we believe that each application has the potential to be leveraged into a standalone tool, with its own user interface and case studies, but this is not our goal here. In fact, thanks to the language-parametric nature of $\mathbb{K}$, none of these reasoning approaches can be regarded as novel *per se*, because they are already used in the context of other languages defined in $\mathbb{K}$ and their implementation is language-semantics agnostic. We begin with a discussion of a use case for hardware verification.

3.6.1 Validating Processor Hardware

Verification is considered one of the most (if not the most) important challenges in modern processor design, for several reasons: (i) the enormous state spaces of modern systems; (ii) the lack of formal specifications in the state-of-practice, (iii) generating high-quality test inputs for simulation, (iv) quantifying/analyzing the extent of coverage of simulation, and (v) generating a complete set of properties for checking. For all these reasons, verification is estimated to use 70% of the resources and time, while design takes only 30% [190].

A fully executable formal ISA-level specification such as the one developed here can improve the state of practice in verification by providing a reliable specification of the functional behavior of hardware with respect to observable states. This increases confidence in the input tests, for both directed and random test generation. High confidence tests can reduce time and increase focus during debugging, triage and diagnosis efforts.

[14]All these applications are realized based on the generic tools offered by the employed $\mathbb{K}$ semantic framework and we do not work for improving their runtime, which is not impressive at time of implementation.

Second, since our method can symbolically execute instructions, it can be used to generate input tests that have high coverage. While such analyses have been done at the detailed RTL level [191, 192, 193, 139, 140, 194, 155, 138], there exists limited similar line of work at the x86-64 ISA level[15]. The most significant advantage of such symbolic execution is the ability to detect corner case or hard to detect bugs [191, 195]. This is analogous to finding security vulnerabilities due to corner-case software bugs, illustrated in Section 3.6.3, but applied to the hardware implementation instead of software. We expect that ISA level symbolic analysis will uncover such subtle and complex bugs due to the higher level of abstraction and greater scalability (in terms of execution lengths) than the RTL analyses.

A closely related challenge is checking the accuracy of ISA specifications, including reference manuals. By using such manual specifications to construct a formal specification, we may uncover errors in the manual specifications. This is explicitly demonstrated by the two bugs we discovered in the Intel x86-64 manual while performing the instruction-level validation tests described in Section 3.5. These bugs were discovered as a result of running test cases using both the formal semantics generated by reading the manuals and the hardware, and finding a mismatch, then checking the manual specification carefully to determine whether the bug lies in the manuals or in the hardware. At this point, it is imperative to cite the pioneering work by Alastair Reid [164] to mechanically transform the ARM processor specifications from documents intended for human consumption into trustworthy machine-readable specifications, which is heavily tested using billions of instructions from official ARM conformance tests [132]. The trust in the auto-generated specs is further enhanced by formally verifying the specification itself (Reid et al. [153]), where they develop high-level properties about the specification by translating natural language text in the reference manual to their property notation and formally verify that the specification satisfies those properties.

Given the formal semantics, a far more valuable strategy would be to *automatically generate human-readable documentation from the formal specification.* A basic version of this strategy is likely quite feasible today [163, 196], and we hope that much more sophisticated versions that synthesize illustrative examples and even explanatory text automatically could be possible soon, given the ongoing advances in concolic test generation, program synthesis, and natural language processing.

[15]It is worth mentioning the work by Martignoni et al. [100] about test-case generation for 32-bit x86 ISA by symbolically executing the instruction implementations in bochs [80] binary emulator. However, floating-point instructions are excluded because the underlying symbolic execution engine does not support them.

```
int s = 0; int n = N;
while (n > 0) { s = s + n; n = n - 1; }
return s;
```

(a) C source code

```
movl %edi, -20 (%rbp)    # %edi holds N
movl $0, -4 (%rbp)       # s = 0
movl -20(%rbp), %eax
movl %eax, -8(%rbp)      # n = N
L3: # loop header
  cmpl $0, -8(%rbp)      # check n <= 0
  jle L2  # if n <= 0, then jump to end, else continue
  movl -8(%rbp), %eax    # n > 0 at this point
  addl %eax, -4(%rbp)    # s = s + n
  decl -8(%rbp)          # n = n - 1
  jmp L3                 # jump back to loop header
L2:
  movl -4(%rbp), %eax    # n <= 0 at this point
  ret
```

(b) x86-64 assembly code

Figure 3.4: `sum-to-n` program

3.6.2 Program Verification

The $\mathbb{K}$ framework provides a language-parametric, reachability logic theorem prover [197, 198]. We instantiated it with our semantics to generate a correct-by-construction deductive verifier for x86-64 programs. Here, the functional correctness properties are specified as reachability specifications, essentially a pair of pre- and post-conditions for each function. The derived x86-64 verifier uses a sound and relatively complete proof system to prove the given specifications w.r.t. the x86-64 semantics. Like in other deductive verifiers, repetitive constructs such as loops and recursive functions need to be annotated with invariants. The verifier is automatic: it requires only the program, its specification, and the invariants.

To demonstrate that our semantics can be used to verify x86-64 programs, we use the x86-64 verifier to prove the functional correctness of the `sum-to-n` program as shown in Figure 3.4. It takes N as input and returns the sum from 1 to N. The functional correctness can be essentially described as: `%rax` $= \sum_1^N n = N(N+1)/2$. We present the actual specification that is fed to the x86-64 verifier. The specification has two parts: the top-level specification and the loop invariant.

```
<regstate>...
    "RDI" ↦ $N_{64}$
    "RBP" ↦ $56_{64}$
    "RIP" ↦ ($0_{64}$  => $-1_{64}$ )
    "RAX" ↦ (_ => $\frac{N*(N+1)}{2}_{64}$)
...</regstate>
<memstate>...
  // -8(%rbp): n
  48 ↦ (byte(0,_) => byte(0, $0_{32}$ ))
  49 ↦ (byte(0,_) => byte(1, $0_{32}$ ))
  50 ↦ (byte(0,_) => byte(2, $0_{32}$ ))
  51 ↦ (byte(0,_) => byte(3, $0_{32}$ ))
  // -4(%rbp): s
  52 ↦ (byte(0,_) => byte(0, $\frac{N*(N+1)}{2}_{32}$))
  ...
  55 ↦ (byte(0,_) => byte(3, $\frac{N*(N+1)}{2}_{32}$))
...</memstate>
  requires N ≥ 0 and N < 2^31 and (N*(N+1))/2 < 2^31
```

(a) Top-level specification

```
<regstate>... "RIP" ↦ (L3 => L2) ...</regstate>
<memstate>...
  // -8(%rbp): n
  48 ↦ (byte(0, A) => byte(0, $0_{32}$ ))
  49 ↦ (byte(1, A) => byte(1, $0_{32}$ ))
  50 ↦ (byte(2, A) => byte(2, $0_{32}$ ))
  51 ↦ (byte(3, A) => byte(3, $0_{32}$ ))
  // -4(%rbp): s
  52 ↦ (byte(0, B) => byte(0, B + $\frac{A*(A+1)}{2}_{32}$))
  ...
  55 ↦ (byte(3, B) => byte(3, B + $\frac{A*(A+1)}{2}_{32}$))
...</memstate>
  requires A >= 0 and A < 2^31 and B >= 0 and B < 2^31
       and B + ((A * (A + 1)) / 2) >= 0
       and B + ((A * (A + 1)) / 2) < 2^31
```

(b) Loop invariant

Figure 3.5: Specification of sum-to-n program

Figure 3.5(a) shows the functional correctness specification of the `sum-to-n` program. The `regstate` cell specifies the relevant registers used in the program, omitting the irrelevant ones denoted by "...". Specifically, it specifies that `%rdi` holds the value N without being updated during the program execution, and `%rax`, which is pre-conditioned on *any* symbolic value (denoted by `mi(64, _:Int):MInt`), is ensured to have the expected return value.

The `memstate` cell specifies the relevant part of the memory omitting others (denoted by "..."). It specifies the stack memory addresses `-8(%rbp)` and `-4(%rbp)` corresponding to `n` and `s`, respectively. Specifically, the stack memory addresses `-4(%rbp)` and `-8(%rbp)` start with *any* state and ensures to have respectively the correct sum and 0 when the program terminates. The `requires` clause specifies the condition of N that prevents the arithmetic overflow. Figure 3.5(b) shows the loop invariant specification. It specifies the behavior of an arbitrary loop iteration. That is, assuming the values of `n` and `s` be A and B, resp., in the beginning of an arbitrary loop iteration, it specifies their final values in the end of the entire loop execution, which are 0 and $B + A(A+1)/2$, respectively. Note that when $A = N$ and $B = 0$, i.e., the first loop iteration, the loop invariant captures the entire loop behavior. The $\mathbb{K}$ verifier takes a minute[16] to verify the `sum-to-n` assembly code satisfies the functional correctness specification.

3.6.3 Symbolic Execution

$\mathbb{K}$ automatically derives a correct-by-construction symbolic execution engine from the given semantics. Being instantiated with our semantics, the engine can be used to symbolically execute and explore all possible paths in the given x86-64 program. In this section, we demonstrate how this capability can be used to find a security vulnerability.

Consider the code snippet of the HiStar [199] kernel, as shown in Figure 3.6(a)[17], in which the KLEE [200] team found a security vulnerability. The `safe_addptr` function is supposed to compute the sum of two arguments `a` and `b`, setting the flag argument `of` when the arithmetic overflow occurs during the addition. That is, one of the functional correctness properties is that "`*of` $= 1$ if `a` $+$ `b` $>$ `r`", where $+$ is the mathematical addition (with no overflow). The functional correctness, however, is not satisfied when the source code is compiled to a 32-bit target, since the size of `r` becomes 32-bit (`uintptr_t`) while

[16]The application is for illustrative purpose and evaluating its scalability is left to future work. However, we note that the reported verification time would not be a major concern for scalability because of the modularity of deductive verification.

[17]For the simplicity of the presentation, in Figure 3.6(b), we highlight only the key computations of the assembly compiled from the source. However, in our experiment, the full unmodified compilation is used for the symbolic execution.

```
uintptr_t safe_addptr(int *of, uint64_t a, uint64_t b) {
  uintptr_t r = a + b;
  if (r < a) { *of = 1; return r; }  // "error state"
  else { return r; } }               // "safe state"
```

(a) C source code

```
# Address %ebp + 12  contains 64-bit value 'a'
# Adress  %ebp + 20 contains 64-bit value 'b'
# Let a[31:0]: lower 32 bits of 'a'
#     b[31:0]: lower 32 bits of 'b'

pushl   %ebp
movl    %esp, %ebp
subl    $32, %esp
movl    12(%ebp), %eax
movl    %eax, -24(%ebp)    # a[31:0] stored in -24(%ebp)
movl    16(%ebp), %eax
movl    %eax, -20(%ebp)    # a[63:32] stored in -20(%ebp)
movl    20(%ebp), %eax
movl    %eax, -32(%ebp)    # b[31:0] stored in -32(%ebp)
movl    24(%ebp), %eax
movl    %eax, -28(%ebp)    # b[63:32] stored in -28(%ebp)
movl    -24(%ebp), %edx    # a[31:0] moved to %edx
movl    -32(%ebp), %eax    # b[31:0] moved to %eax
addl    %edx, %eax         # r = a[31:0] + b[31:0]
movl    %eax, -4(%ebp)     # store r at -4(%ebp)
movl    -4(%ebp), %eax
movl    $0, %edx           # check if (0_32  ∘ r) < a
cmpl    -24(%ebp), %eax    #  ∘: denotes concatenation
movl    %edx, %eax
sbbl    -20(%ebp), %eax
jnc     .L2
movl    8(%ebp), %eax      # true branch: "error state"
movl    $1, (%eax)         # set *of to 1 and %eax to r
movl    -4(%ebp), %eax     # prepare return value r
jmp     .L3
.L2:
movl    -4(%ebp), %eax     # else branch: "safe state"; prepare return value r
.L3:
leave
ret
```

(b) x86-64 assembly code in a 32-bit target

Figure 3.6: A security vulnerability in the HiStar kernel

```
int popcnt(uint64_t x) {
  int res = 0;
  for (; x > 0; x >>= 1) { res += x & 0x1ull; }
  return res; }
```

Figure 3.7: `popcnt` program

the sizes of `a` and `b` are still 64-bit (`uint64_t`).[18] A suggested fix [200] is to change the conditional expression from `r < a` to `r < a || r < b`.

Using the symbolic execution engine derived from our semantics, we could find (in ~80 seconds) that, in the assembly code as shown in Figure 3.6(b), there exists a path that reaches `L2` (i.e., the else branch) even if the addition overflow occurs. The (simplified) path condition provided by the symbolic execution engine is $\mathtt{a}+\mathtt{b} \geq 2^{32} \ \wedge \ (\mathtt{a}+\mathtt{b} \mod 2^{32}) \geq \mathtt{a}$, where $0 \leq \mathtt{a} < 2^{64}$ and $0 \leq \mathtt{b} < 2^{64}$. We asked Z3 to solve the above path condition and it returned a solution (i.e., a concrete input to trigger the security vulnerability): `a` = `0x00000000ffffffff` and `b` = `0xffffffff00000000`.

3.6.4 Translation Validation of Optimizations

$\mathbb{K}$ also provides a program equivalence checker that can be used for the translation validation of compiler optimizations. We derived an x86-64 program equivalence checker from our semantics and used it to validate different optimizations. Figure 3.7 shows a program that we considered, `popcnt`, which counts the number of set bits in the given input.

We compiled the program with different optimizations: the GCC compiler optimizations (-O0, -O1, -O2, and -O3), and the Stoke super-optimization. On top of the baseline (-O0), the -O1 optimization produces a code obtained by performing the `mem2reg` optimization, the -O2 optimization produces one by factoring out the common statement over different branches,[19] and the -O3 optimization produces the same code with -O2. The Stoke super-optimization translates the assembly code into a single instruction: **`popcnt %rdi, %rax`**, where `%rdi` and `%rax` correspond to the input and the return values, respectively.

We validated these optimizations by checking the equivalence between the optimized programs. The equivalence checker symbolically executes each program and compares their return values (i.e., the symbolic expression of the `%rax` register value) using Z3. It is able to

[18] The function call `safe_addptr(*of, address, size)` is used to validate that an user is allowed to access the memory range specified by the arguments `address` and `size`. The access is denied if an overflow occurs. A bug in the overflow detection might be exploited by an attacker to gain an access to a memory region beyond the control of the running process.

[19] Specifically, by performing the common subexpression elimination, followed by certain statement reordering optimization, followed by the strength reduction.

prove successfully that all optimization variants are equivalent, i.e., to check the correctness of all these optimizations on `popcnt`.

Note that the symbolic execution of the `popcnt` program does not require an additional annotation about the loop because the number of loop iterations is bound to a constant (i.e., the bit-size of the input, 64).[20] In general, however, the equivalence checker may require us to provide an additional annotation about loops, which can be automatically generated by augmenting the underlying compiler.

3.7 LIMITATIONS

Our limitations mostly include missing features of the x86-64 and execution environment, as follows.

Floating-Point Operations. Our testing shows that we have FP precision issues with instructions implementing the fused-multiply-add operation. This is because the current $\mathbb{K}$'s floating-point library [201] implementation lacks support of the FMA capabilities of GNU MPFR library [202], which we plan to include in future.

Exceptions. We do not support exceptions, including the FP exceptions. Moreover, we do not distinguish between quiet and signaling NaN, i.e. all NaNs are quiet in our model. When the exception condition is encountered, execution proceeds after setting the exception flag.

Concurrency. Like the closest previous work [88, 87], we do not model concurrent semantics or the relaxed memory model as defined by other previous work [127, 126]. Our design, being parameterized on memory model, is amenable to accommodate others', which we plan to achieve in future.

Instruction Decoding. The instruction decoding semantics is *not* modeled in the current work. However, we want to note that we have formalized the Intel XED disassembler algorithm, and we are working on formalizing the instruction decoding.

3.8 LESSONS LEARNED

Here we present the lessons we learned during our semantics development, identifying important aspects to be considered, and clarifying best practices for developing a large ISA

[20]However, an additional annotation about the loop (i.e., a loop invariant) can be provided to improve the symbolic execution performance. For example, symbolic execution of the `popcnt` program without the loop invariant requires to iterate the loop 64 times, which takes ∼20 minutes, but it can be reduced to a minute if the loop invariant is provided.

semantics. We also discuss the novel aspects of our semantics development approach that allow us to obtain a complete and faithful semantics with a practical amount of effort.

Automatic semantics synthesis Most previous efforts in formalizing x86-64 semantics can be categorized based on whether the underlying approach is fully manual [88, 91, 90, 129] or fully automatic [87, 112, 92, 93]. We note that none of these approaches, when used in isolation, sufficiently scale to a complete and faithful semantics, as much as ours that combines these complementary approaches so that they benefit from each other.

Section 3.4.2 reports the challenges we encountered in achieving fully automatic synthesis of the entire x86-64 semantics. Specifically, in a vast instruction set like x86-64, it is common that many instructions can be grouped together where the instructions of each group are similar to each other except for a few differences. An automatic synthesis technique leveraging such a group, such as the stratification approach [87], would effectively synthesize such instruction variants' semantics, provided that the semantics of representative instructions in each group are given in advance.[21] The problem, however, is that it is non-trivial to properly partition all the instructions into such groups, providing the representative instruction semantics for each group, *without* a priori knowledge about the semantics of all instructions. The stratification approach [87] had been proposed to solve this dilemma, but it turned out to be not sufficient, leaving a substantial part of semantics unspecified. The vanilla stratification approach [87] turned out to be not sufficient to solve this dilemma, leaving a substantial part of the semantics unspecified. Thus, we decided to manually provide the information about the partition and representatives, for which we had to consult the manual to obtain knowledge about the remaining part of semantics. Once we obtained the required knowledge, however, we realized that it would be more straightforward to directly turn the knowledge into the semantics than going through the synthesis process, and thus we ended up manually specifying the remaining part of the semantics.

Another important step of the semantics synthesis is post-processing. The generated semantics is often verbose and not necessarily human-readable. The post-processing step is desired to simplify the generated semantics to be succinct, which helps to increase the human-readability as well as to improve the efficiency when being employed in other applications (e.g., the size of SMT formula encoding can be reduced, which can reduce the burden of SMT solvers). For our semantics development, we have written dozens of simplification rules that are fed to the $\mathbb{K}$ framework to simplify the synthesized semantics further (Section 4.1).

[21]For certain complex instructions, the size of their group is very small (i.e., they are quite different to each other), and thus the automatic synthesis would not yield a sufficient gain over the effort of specifying the semantics of their representatives, but we found that the number of such isolated instructions of x86-64 is small.

Modeling and executing implementation-dependent behaviors The x86-64 ISA standard admits *implementation-dependent behaviors* for certain operations on certain input patterns, that is, each processor implementation can freely choose the execution behavior for each such case (Section 3.2). Faithfully modeling the implementation-dependent behaviors is necessary for the correctness of the semantics. For example, as mentioned in Section 3.5.2, Stoke [135] does not faithfully model such behaviors, causing certain errors in their semantics that we revealed [110].

There are two natural, faithful ways of specifying implementation dependent behaviors. One is to parameterize the semantics over the implementation-dependent behaviors, and later instantiate it with a profile that describes specific behaviors taken by the processor of interest. This approach is desirable for validating the semantics using concrete execution. Another is to introduce non-determinism in the semantics, which captures a set of different possible behaviors in a single semantics, which is desirable during symbolic interpretation of the ISA code. We note that most of other existing *direct* x86-64 semantics employ approaches similar to the ones described above, faithfully modeling the implementation-dependent behaviors. For example, Goel *et al.* [89] models such behaviors using a constraint function which is guaranteed to be unique and non-deterministic, while they employ the aforementioned profile-based approach for concrete execution. TSL [91] makes both approaches available, from which their users can choose.

In our semantics, we faithfully modeled the undefined value as a unique symbol (called undef) whose value is non- deterministically decided each time within the proper range. For validating the semantics, we concretely executed the semantics while the non-deterministic behaviors are represented symbolically using the undef symbol and then we checked if the hardware output is matched by (an instance of) the simulated output.

Employing multiple semantic engineering frameworks We found that employing multiple semantic frameworks is helpful. Specifically, we employed the two semantic frameworks, $\mathbb{K}$ and Stoke, where we enjoyed all of their (executive) benefits that make it easier for us to write and validate the semantics, and utilize the semantics in various applications. For example, we wrote the semantics of certain complicated instructions (e.g., **`pcmpestri`**, **`pcmpestrm`**, and **`pclmulqdq`**) in $\mathbb{K}$, as $\mathbb{K}$ provides an easy way to specify behaviors with multiple cases, while Stoke would have required us to write a big nested if-then-else expression, which is not convenient. As another example of the benefits, we used Stoke to validate most of our instruction semantics as Stoke provides an infrastructure[22] for hardware co-

[22]Indeed, we contributed to their infrastructure as well [111, 203].

simulation, whereas we employed $\mathbb{K}$ to validate the semantics of floating-point instructions as Stoke does not support executing floating-point operations while $\mathbb{K}$ does.

In order to use the two frameworks interchangeably, we developed a translator between the semantics of the two frameworks. To check the correctness of the translation, we verified equivalence between the original and the translated semantics for each instruction using the Z3 SMT solver.

To summarize, employing multiple frameworks with validated translation between them improved both the ease of specification (using $\mathbb{K}$) and ease of validation (using Strata), which expedited our semantics development process and thus significantly contributed to the completeness of our semantics. Moreover, we immediately benefit from all of their formal analysis tools, increasing the applicability of the semantics in various formal reasoning tasks. Existing semantics development efforts (e.g., [88, 87]), however, employ a single framework without utilizing the potential of other frameworks, which otherwise might have improved completeness and/or faithfulness of their semantics with the same amount of effort.

CHAPTER 4: SCALABLE VALIDATION OF BINARY LIFTERS

In this chapter, we present our published contribution [146] to show that formal translation validation of single instructions for x86-64 is not only practical, but can be used as a building block for a scalable solution to full-program translation validation eliminating the expense of heavyweight equivalence checking.

The rest of this chapter proceeds as follows. The next section gives a high-level overview of our approach. Section 4.2 gives some background on building blocks used in our work. Section 4.3 describes our approach for formal single-instruction Translation Validation. Section 4.4 describes how we scale to full-program Translation Validation. Section 4.5 describes our experimental evaluation, including the bugs in McSema uncovered by our work, and the effectiveness of full-program Translation Validation. Section 4.6 briefly discusses some of the key limitations of this work to date, along with avenues for future work.

4.1 APPROACH OVERVIEW

In this section, we provide a high-level overview of the two main components of our approach, i.e., single-instruction translation validation and program-level validation. Before we begin, we first describe the scope to which our approach is currently applicable.

Applicability of our Approach. Our techniques are generally applicable to verify binary lifters from any ISA, e.g., x86, ARM, RISC-V, PowerPC, to an intermediate representation, such as LLVM IR [96], VEX IR [16] etc., as long as (a) formal semantics for both the ISA [1] and the target languages are available, and (b) the target language can be transformed to a canonical representation through a series of semantics-preserving transformations. Through the rest of this work, we fix our discussion to lifting x86-64 to LLVM IR using the most mature, open-source lifter McSema [33]. Our canonicalizer is approximated using a subset of LLVM optimization passes. LLVM, being an industry standard compiler IR, many decompilation projects [33, 49, 57, 67, 61] prefer to employ LLVM as their lifted representation mainly because it enables various "out-of-the-box" analyses and optimizations which minimizes the effort in the post-lifting decompilation tasks. Moreover, LLVM IR is backed-up with its formal semantic models [94, 95] which assists formal rea-

[1]Fortunately, this is not a bottleneck for some of the widely used ISAs as they are already backed-up with formally specified semantics (e.g. x86 [91, 90, 204], ARM [128, 205, 156], RISC-V [204, 90], CHERI-MIPS [128], PowerPC [90, 91]). For x86-64 ISA, despite several explicit attempts [87, 88, 89] and other related systems [90, 49, 91, 92, 93], there does not exist any *complete* formal semantics of x86-64 up until we define a fully executable formal semantics of x86-64 [86] (refer chapter 3).

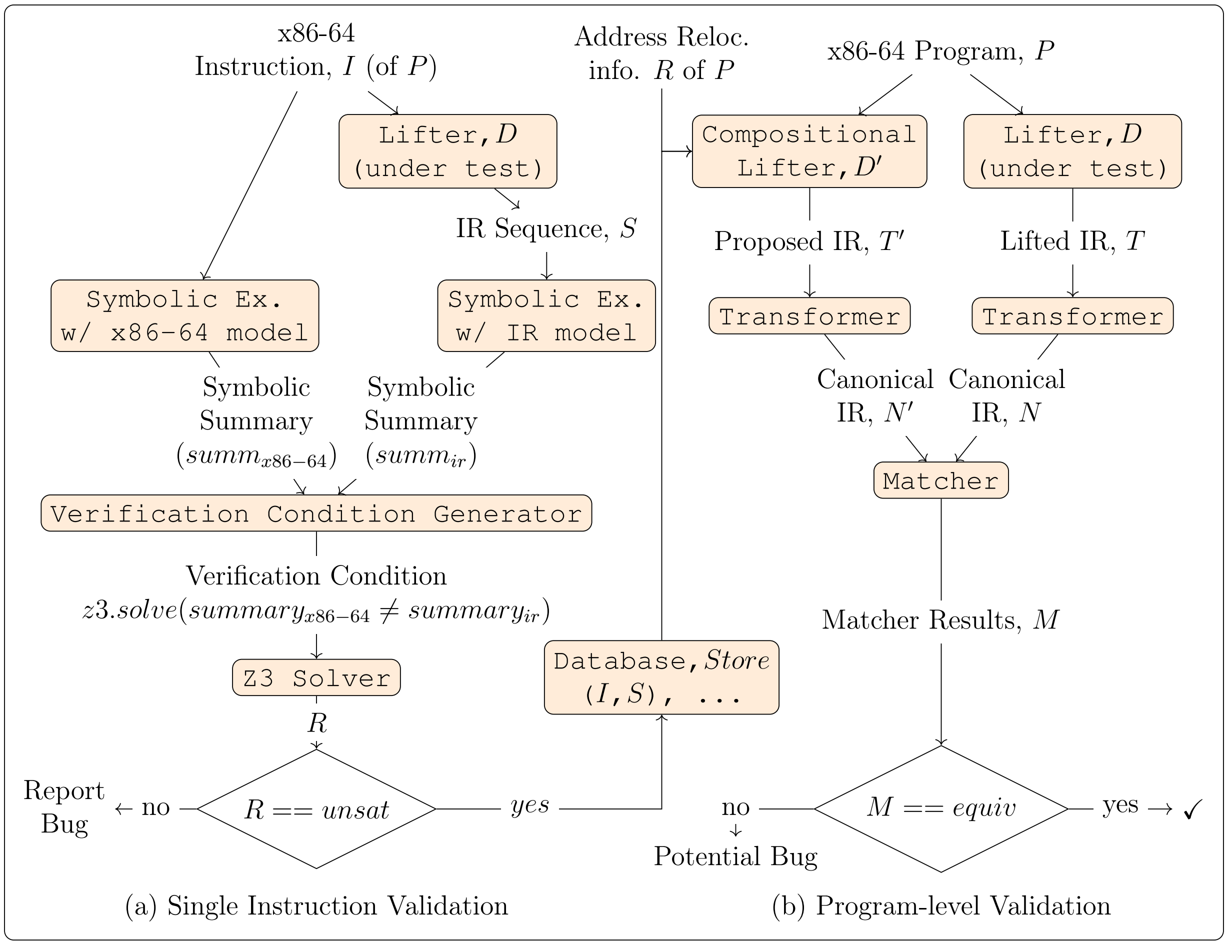
x86-64 Instruction, I (of P)
Lifter, D (under test)
IR Sequence, S
Symbolic Ex. w/ x86-64 model
Symbolic Ex. w/ IR model
Symbolic Summary ($summ_{x86-64}$)
Symbolic Summary ($summ_{ir}$)
Verification Condition Generator
Verification Condition $z3.solve(summary_{x86-64} \neq summary_{ir})$
Z3 Solver
R
$R == unsat$
Report Bug
no
yes
Address Reloc. info. R of P
Database, $Store$ (I, S), ...
x86-64 Program, P
Compositional Lifter, D'
Lifter, D (under test)
Proposed IR, T'
Lifted IR, T
Transformer
Transformer
Canonical IR, N'
Canonical IR, N
Matcher
Matcher Results, M
$M == equiv$
no
Potential Bug
yes → ✓
(a) Single Instruction Validation
(b) Program-level Validation

Figure 4.1: Overview diagram of the translation validation framework

soning on program written in LLVM IR. Similarly, x86-64 instruction set architecture (ISA) is one of the most complex and widely used ISAs on servers and desktops. Hence, it is imperative to ensure the correctness of lifter targeting x86-64 program binary to LLVM IR. Other notable lifters [57, 66] from x86-64 to LLVM IR may be directly supported in our framework through minimal engineering effort. Additionally, we restrict our work to the common case of compiler-generated binaries, and we do not consider binaries that are deliberately obfuscated to deter reverse engineering, which is in-line with previous work on translation validation. Lastly, as we aim to validate the lifted code and do not focus on finding bugs lower in the pipeline, e.g., in the loading and disassembly of binaries. This is orthogonal to our work and has been shown to be relatively mature for the typical case of compiler-generated binaries [206].

Our overall approach is a composition of two techniques, as shown in Figure 4.1, to validate the translation of an x86-64 program `P` to a lifted LLVM IR program `T` using a lifter `D`.

Single-Instruction Translation Validation The goal of single-instruction translation validation, as shown in Figure 4.1(a), is to formally validate the translation of individual instructions of `P` in isolation using the following steps: (i) Each x86-64 instruction `I` is lifted to an LLVM IR sequence `S` using the lifter `D` (McSema in our case), (ii) Next, we identify the input/output variable correspondence between `I` and `S`, i.e., we determine a mapping of registers/memory in `I` to IR entities in `S`, (iii) Using the formal semantics of the x86-64 and IR, we perform symbolic execution to generate symbolic summaries for `I` and `S`, (iv) Lastly, we say `S` is the correct translation of `I` if the corresponding summaries are semantically equivalent. We employ the $\mathbb{Z}3$ [207] solver for the equivalence checks. If the two summaries mismatch, meaning we find a bug, which is then reported. Otherwise, we add the pair `<I,S>` to a database (called *Store*), keyed by `I`, allowing reuse of the validation result.

Program-Level Validation The program-level validation, as shown in Figure 4.1(b), aims to validate that the lifted LLVM program `T`, generated by a lifter `D`, is the correct translation of binary program `P`. The key idea behind the validation strategy is to propose an alternate LLVM program $\mathtt{T}'$ as a reference translation to be compared against `T`. The translation $\mathtt{T}'$ is generated using a tool we developed, called Compositional Lifter, by carefully composing the validated lifted IR sequences corresponding to the individual binary instructions of `P`. The validated IR sequences are provided by the single-instruction translation validation technique above. *The composition T' preserves the data- & control-flow of the original binary-program `P` and, more importantly, is syntactically very close to the original lifted IR `T`.*

Next, we seek to compare `T` and $\mathtt{T}'$ *one function at a time*. Towards that goal, we use a set of 17 manually discovered LLVM optimization passes to close the syntactic gap between every pair of corresponding functions, `F` & $\mathtt{F}'$ of `T` & $\mathtt{T}'$ resp., except for the names of virtual registers and the order of non-dependent instructions. We compare the data dependence graphs extracted from the optimized pair of functions by a Matcher based on graph isomorphism (refer to Section 4.4.2). The isomorphism of data dependence graphs, for each pair of optimized functions, implies that the original lifted IR `T` is semantically equivalent to the reference translation $\mathtt{T}'$, and therefore `T` is the correct lifting of the input binary program. Otherwise, the automatic validation fails, and mismatch reported as a potential bug for further analysis. The reason that we can get false alarms (i.e., even if `F` and $\mathtt{F}'$ are semantically equivalent, the data dependence graphs extracted from their optimized versions are not isomorphic) because the selected LLVM passes may not be effective in reducing `F` and $\mathtt{F}'$ to isomorphic graphs.

Composing the Techniques In essence, the two techniques are independent, and their results do not depend on the other, with one minor caveat: the results from program-level validation, either a complete equivalence match, or a potential mismatch, are not sound until the IR instruction sequences used to construct `T′` are validated by the single-instruction translation validation. However, the ordering between the two techniques does not matter, i.e., single-instruction translation validation may be done offline; either ahead of time, when composing instruction during program-level validation, or done in a batch after program-level validation.

4.2 PRELIMINARIES

In this section, we provide background on various pieces used in our work: (i) The binary lifter under test, McSema, (ii) The formal x86-64 semantics, and (iii) The formal LLVM IR semantics.

McSema McSema [33] is the most mature, well tested, open-source lifter to raise binaries from x86-64 instructions to LLVM bitcode. At a high level, McSema is split into two parts: (a) front end, and (b) back end. The front end is responsible for parsing, loading, and disassembling a binary and exports an interface to the back end to query for the required information, e.g., the defined symbols, sizes of various binary sections, instruction listings, etc. The back end then uses this information and Remill [49] library to lift the individual instructions. McSema supports multiple different front ends with IDA Pro being the most robust, and supported option.

Conceptually, the implementation of McSema's back end is fairly straightforward: McSema exposes all of the architecture state, i.e., the program registers, conditional flags, and program memory, through an LLVM *struct*, aptly named *State*, which is passed as an argument to every lifted function. McSema simply scans through the disassembly of the binary and lifts each instruction one by one, emitting code to read and/or update the members of the struct based on the semantics of the lifted instruction. In essence, the code lifted by McSema simply encodes the operational semantics of the binary in LLVM IR.

x86-64 Formal Semantics Our current work uses state-of-the-art x86-64 semantics, developed in our previous open-sourced work [86], which presented the most complete, thoroughly tested formal semantics of x86-64 to date, and faithfully formalizes all non-deprecated, sequential user-level instructions of x86-64 Haswell instruction set architecture. The specification covers 774 mnemonics, and each mnemonic admits several variants (3155 in total), depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands. The semantics, defined in $\mathbb{K}$ [178], is executable (i.e. allows concrete execution), and comes with a symbolic execution engine automatically generated by the $\mathbb{K}$ framework[2].

LLVM IR Formal Semantics We use the LLVM formal semantics [94], defined in $\mathbb{K}$, which models LLVM types (integers, composite arrays, structs and their corresponding pointers), the `getelementptr` instruction (used to compute the address of an element nested within a aggregate data-structure), integer arithmetic & comparison operators, memory operations (`load`, `store`, and `alloca`), control flow instructions for unconditional and conditional branches, as well as function calls and returns. However, the semantics does not support floating-point, vector types, and most LLVM intrinsic functions. As a result, we cannot validate the translation validation of certain binary instructions whose lifted IR includes such unsupported constructs. This is a limitation of the available LLVM semantics and not a limitation of our work.

4.3 SINGLE-INSTRUCTION TRANSLATION VALIDATION

The single-instruction translation validation is responsible for validating the lifting (using McSema) of an x86-64 instruction `I` to LLVM IR sequence `S`. This is achieved by (1) Establishing variable correspondence between `I` and `S`, (2) Generating symbolic summaries individually for `I` and `S` for each output variable, (3) Generating verification conditions meant to establish semantic equivalence between the corresponding pair of summaries, and solving those using an SMT solver ($\mathbb{Z}3$). Next, we describe each one of these steps.

(1) Establishing variable correspondence: "Variable correspondence" between `I` and `S` refers to identifying the correspondence between the input/output variables of `I` and those of `S`. By input (resp., output) variables of an instruction we mean implicit and explicit register/memory/flags which are read (resp., written). By input (resp., output) variables of an lifted IR sequence `S` we mean the IR variables which are used to simulate the input (resp., output) variables of `I`. This information is valuable in setting up pre-conditions over corresponding input variables and post-conditions over output variables, thereby assisting the equivalence proofs between `I` and `S`.

As described in Section 4.2, McSema models the hardware architecture state using a *State* structure which holds all the simulated hardware registers at different offsets in the structure.

[2]Given a syntax and a semantics of a language, $\mathbb{K}$ automatically generates a parser, an interpreter, a symbolic execution engine, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional effort.

Hence, the input and output variables in the context of McSema are particular *struct* fields, identified by constant offsets. As an example, for an instruction **`adcq %rax, %rbx`**, the input variables are `%cf`, `%rax` & `%rbx`, and output variables are `%rbx`, `%cf`, `%pf`, `%sf`, `%zf`, `%of`, and `%af`. Figure 4.2 shows how these input/output registers are mapped to the McSema *State* structure in lifted LLVM code.

```
// State structure type with irrelevant fields replaced with "...".
// The nested type "struct.GPR", at offset 6, models the general-purpose
// simulated registers. Similarly, the type "struct.ArithFlags", at offset 2,
// models the simulated status flags.
%struct.State ↦ type { %struct.ArchState, ...,
    %struct.ArithFlags,..., ..., ..., %struct.GPR, ...}

// Pointers to simulated registers (or flags) are computed using LLVM's
// getelementptr instruction. The constant operands m and n are offsets
// to index to the different nested elements of an object pointed to
// by a base pointer "%state" of the above type, denoting field
// n within the nested struct at field m of structure "%struct.State".
getelementptr inbounds %struct.State, %struct.State* %state, i64 0,
        i32 m, i32 n, i32 0, i32 0

// Mapping of various simulated registers to getelementptr offsets.
    rax ↦ m = 6  n = 1;  rbx ↦ m = 6  n = 3
     cf ↦ m = 1  n = 1;   pf ↦ m = 1  n = 3
     af ↦ m = 1  n = 5;   zf ↦ m = 1  n = 7
     sf ↦ m = 1  n = 9;   of ↦ m = 1  n = 13
```

Figure 4.2: The mapping of input/output registers of **`adcq %rax, %rbx`** to the McSema *State* structure

We use the above architectural state representation of McSema to infer how the hardware registers or flags in the binary instruction corresponds to the simulated version of those in the corresponding lifted IR[3].

(2) Generating symbolic summaries: The $\mathbb{K}$ framework takes the $\mathbb{K}$-specification of x86-64 (resp., LLVM IR) as input and automatically generates a symbolic execution engine which we leverage to do symbolic execution of an x86-64 instruction (resp., the corresponding lifted LLVM IR sequence). The result of symbolic execution on an x86-64 instruction (resp., the corresponding lifted IR) is a set of summaries capturing the output behaviors corresponding to each register, flag, and clobbered memory (resp., the simulated version of those in the lifted IR), expressed using $\mathbb{K}$ builtin operators such as `add`, `concat` and

[3]Like McSema, fcd [57] has a similar approach to model the architectural state and infer variable correspondence. In case of Rev.Ng [66], the architecture registers are modeled as LLVM globals and variable correspondence refers to the mapping between x86-64 registers with those globals.

`extract`, over the symbolic values assigned to the input variables. For the running example of **`adcq %rax, %rbx`**, Figure 4.3 shows the symbolic summary corresponding to the output register `%rbx`[4] (summaries of other registers and flags are omitted).

```
// V_CF1, V_RAX64 and V_RBX64 are the symbolic values
// assigned to input variables. The subscript denotes
// the bit-width of the value. "extract" returns bits
// 1..64, where bit 0 is the most significant bit.
extract (
    add (
        (#if eq ( V_CF1 , 1_1 ) #then
            add ( concat ( 0_1 , V_RAX64 ) , 1_65 )
        #else
            concat ( 0_1 , V_RAX64 )
        #fi)
        , concat ( 0_1 , V_RBX64),
        )
    1 , 65 )
```

Figure 4.3: The symbolic summary corresponding to the output register `%rbx` of instruction **`adcq %rax, %rbx`**

Similar symbolic summaries will be obtained for the simulated registers and flags in the lifter IR sequence, which is omitted as well for brevity.

Most x86-64 instructions require a bounded (and small) number of operations. However, the x86-64 ISA includes instructions with Repeat String Operation Prefix (e.g. **`rep`**, **`repz`** etc.) to repeat a string instruction the number of times specified in the count register or until the indicated condition by the prefix is no longer met. That is, their specification involves a loop that the symbolic execution must handle. Conceptually, such loops can be realized using a *for* loop with index as the loop count decreasing by one in every iteration and the body consists of a *if* check which can break the loop if the indicated condition is met or the index reduces to zero. These loops are bounded by the maximum value the count register can hold and are simple as the index cannot change in any other ways; thus the x86-64 instruction will trivially terminate. In order to prove the equivalence of the translation of such an instruction, we first set up the precondition asserting that the register or memory value, corresponding to the loop trip count, and the corresponding simulated register in lifted IR are equivalent. Next, we symbolically execute the instruction and its corresponding lifted IR with a symbolic input state and comparing the summaries (using solver checks) of any single i^{th} iteration of the two loops. This suffices to establish equivalence between the two

[4]All the values or addresses stored in registers, memory or flags are implemented as bit-vectors and represented as V_W to be interpreted as a bit-vector of size W and value V.

loops, by co-inductive reasoning [208] to check the behavior of corresponding loops evolves in lock-step, and the fact that such loops are bounded by a constant thus must terminate [5].

(3) Generating & Solving the verification conditions: First, we convert the summaries written in $\mathbb{K}$ builtin operators to SMTLIB expressions. Given two symbolic summaries $\text{summary}_{x86-64}^{rbx}$ and $\text{summary}_{ir}^{rbx}$, for output x86-64 register `%rbx` and corresponding simulated register, we emit a satisfiability query as follows, to be solved by an SMT solver like $\mathbb{Z}3$,

$$\texttt{(assert (not (= summary}_{x86-64}^{rbx}\ \texttt{summary}_{ir}^{rbx}\texttt{)))}$$

Similar queries are generated for all registers, flags, and clobbered memory. Moreover, we add pre-conditions asserting the equivalence of input symbolic values assigned to the input variables of the binary instruction and its corresponding variables in the lifted IR. Note that we generate queries for all registers/flags, not just the ones clobbered because the registers and flags not modified by the instruction should have equivalent summaries (which is the unmodified value of the symbolic input value).

The verification condition queries are then dispatched to the $\mathbb{Z}3$ solver to prove equivalence between corresponding summaries. When the query of non-equivalence is satisfiable, the solver generates an example which can be used as a test input to trigger the mismatch. Any such mismatches are regarded as bugs in McSema and reported along with the associated test inputs.

Even though we are using solver checks during the first phase, this should not hamper the scalability of our program validation pipeline for the following reasons. First, the instruction-level validation is done for each instruction. Thus its verification condition is much simpler than that of whole program-level validation. Second, the validation result of each instruction can be reused within a program or across different programs; thus the validation cost can be amortized, or done offline. Note that the reuse of validation results is facilitated by the Store database (Figure 4.1).

Single-instruction translation validation of control-flow instructions: The single-instruction translation validation for control-flow instructions, e.g., jump (conditional/unconditional) and call, is critical in ensuring that the control-flow of the binary program is preserved in the McSema-lifted program. A conditional jump instruction, e.g., **`jcc rel-offset`** at program counter `pc`, evaluates the condition code `cc` as an appropriate expression over

[5] We manually inspect that the symbolic summaries corresponding to the loop trip count decrements by one in every iteration when the indicated condition is not met. Also, the count register is not modified in any other way.

the status flags, and updates the `%rip` with either the address of the target instruction (`pc + rel-offset`) or of the fall-through instruction (`pc + sizeof(jcc)`). Such an instruction is lifted to LLVM IR code with three goals: (1) computing the condition code value, `%cond`, matching the value of `cc`; (2) updating the value of the simulated register corresponding to `%rip`; and (3) transferring control-flow to the appropriate basic block, using an LLVM branch instruction (e.g., **`br i1 %cond, label <LT>, label <LF>`**), based on value of `%cond`.

The goals of single-instruction translation validation for the running example of **`jcc`** are twofold: (A) to ensure that the update of `%rip` by the binary instruction and of the corresponding simulated register in lifted LLVM IR are *equivalent*, and (B) the LLVM `br` instruction should preserve the control-flow semantics of the corresponding binary instruction, i.e., the **`jcc`** and **`br`** instructions should evaluate equivalent conditions (i.e., cc $\simeq$ value of `%cond`) and based on it's evaluation the control should jump to *corresponding* targets, i.e., if `%cond` is true, then the basic block with label `LT` should begin with an instruction corresponding to the target instruction at `pc + rel-offset`, else the basic block with label `LF` should begin with an instruction corresponding to the target instruction at `pc + sizeof(jcc)`.

To ensure (A), we symbolically execute **`jcc`** (resp., corresponding lifted IR) with concrete `pc` assigned to the `%rip` (resp., corresponding simulated resister) and symbolic values assigned to the status flags (resp., corresponding simulated flags) affected by the condition code `cc` (resp., `%cond`). We compare the resulting symbolic summaries, for the register `%rip` and its simulated counterpart, for equivalence using a solver preconditioned on the equivalence of respective symbolic inputs.

To achieve (B), we exploit the observation that the lifted IR encodes the target addresses of **`jcc`** instruction (or the potential values of `%rip`) in the branch labels of the LLVM **`br`** instruction[6]. An example of label `LT` is `%block-4004b4`, where `4004b4` is the target address when the condition code `cc` is satisfied. Moreover, we define an auxiliary state in LLVM semantics which captures the embedded address of the current block label. A simplified version of the syntax and semantics of this state cell (`aux-currentBlock`) is shown in Figure 4.4. The symbolic execution of the lifted IR, as mentioned above, provides the summary of this special state, which is compared for equivalence with the summary of `%rip` with similar preconditions, as mentioned above.

The single-instruction translation validation of other control flow instructions (like unconditional jump and call) are handled similarly.

[6]Similarly, for **`call`** instruction, the name of the lifted function call encodes the target address of the callee.

```
 // snippet of the LLVM configuration cell
 syntax LocalID ::= localID(String)
 <control>
    <currentBlock>  LocalID </currentBlock>
    // The following is the newly introduced auxiliary state cell
    <aux-currentBlock> mi(64, I:Int)  </aux-currentBlock>
 </control>

// The following rule is for conditional branch when the value of the branch condition cond
// is evaluated as false, in which case the control jumps to the basic block with label Lbl.
// The basic block, Lbl, hosts instructions Insts followed by terminator instruction TermInst
// like branch, return etc. The cell "basicBlocks" includes a mapping from basic-block labels to
// the details about the corresponding basic block. The cell "currentBlock" stores the label of
// the current basic block.
rule <k>
    brCond(cond:Int, _:LocalID, Lbl:LocalID) => Insts ~> TermInst
</k>
    <basicBlocks>...
        Lbl |-> bb(Lbl, Insts:Instructions, TermInst:TerminatorInstruction)
    ...</basicBlocks>
    <currentBlock> CurrentLbl:LocalID => Lbl </currentBlock>
    <aux-currentBlock>
       // ExtractAddrFromBlockLbl is a K-function which converts block labels
       // like %block-4004b4 to 4004b4
        _ => mi(64, String2Int(ExtractAddrFromBlockLbl(Lbl)))
    </aux-currentBlock>
requires cond ==Int 0
```

Figure 4.4: Syntax and semantics of the auxiliary state cell used to capture the the embedded address of the current block label.

4.4 PROGRAM-LEVEL VALIDATION

The goal of program-level validation is to validate the translation of the input x86-64 program `P` to the McSema-lifted LLVM IR program `T`. Towards that goal, the first step is to construct an alternative program `T′` generated using the Compositional Lifter (Section 4.4.1), which is then compared with `T` using the Matcher (Section 4.4.2).

4.4.1 Compositional Lifter

The Compositional Lifter is responsible for generating the proposed LLVM IR `T′` by composing the validated McSema-lifted IR sequences of the constituent binary instructions of the x86-64 program `P`. Importantly, the Compositional Lifter design (Algorithm 4.1) is simple and took us about three man-weeks to implement, mainly because it reuses the individual instruction translations performed by McSema. These are separately validated using single-instruction translation validation, as described in the previous section.

`P` is disassembled to identify function boundaries, and to decode instructions. If the decoded instruction I is already in Store, then its corresponding (validated) IR sequence is reused (line 13). Otherwise, I is lifted (using McSema) (line 5) to generate an LLVM IR sequence that is going to be validated using Phase 1 (lines 6-11). The validated IR sequences are then composed (line 15) following the data- and control-flow order of the binary program `P`.

Algorithm 4.1: Compositional Lifting

```
Inputs :
        P: x86-64 binary program.
        Store: Validated pairs (<I, S> ) of instruction I and lifted IR sequence S.
(possibly empty)
        R: Address Relocation information of binary P.
Output: Lifted IR Program T′
1  $T' \leftarrow \phi$
2  foreach function $F$ in $P$ do
3      foreach instruction $I$ in $F$ do
4          if $I$ not in Store then
5              $S \leftarrow$ McSema ($I$)
6              Perform Translation Validation of $I$ and $S$ (Phase 1)
7              if Validation successful then
8                  Add $< I, S >$ to Store
9              else
10                 Report Bug
11         else
12             Extract $S$ from Store for $I$
13         $T' \leftarrow$ Compose($T', S, R$)
14 return $T'$
```

The "Compose" step The code listings below explain the step "Compose" (line 15 in Algorithm 4.1) using an example. Listing 4.1 shows a binary code snippet which is lifted to LLVM IR using Compositional Lifter, shown at Listing 4.2. The parts irrelevant for the current exposition are omitted using "...". In the lifted IR, we have LLVM function calls one for each binary instruction. The callee (not shown in the listing below) includes the same validated IR sequence, implementing the operational semantics of an instruction, obtained during single-instruction translation validation.

```
        ...
400494: mov    %edi,-0x8(%rbp)
400497: cmpl   $0x1,-0x8(%rbp)
40049b: jge    4004ad
4004a1: movl   $0x1,-0x4(%rbp)
4004a8: jmpq   4004b4
4004ad: movl   $0x0,-0x4(%rbp)
        ...
```

Listing 4.1: Binary code snippet

```
define ... @composedFunc(%struct.State* %st, ...)  {
   ...
   <glue code>
   ; function sub_movl__edi__MINUS0x8__rbp_ implements
   ; the operational semantics of mov %edi,-0x8(%rbp).
   ; Same is true for the subsequently called functions.
   ... = call ... @sub_movl__edi__MINUS0x8__rbp_(...)

   <glue code>
   ; Code:  cmpl $0x1,-0x8(%rbp), RIP: 400497, bytes:  4
   ... = call ... @sub__cmpl__0x1__MINUS0x8__rbp_(...)

   <glue code>
   ; Code:  jge 4004ad, RIP: 40049b, bytes:  6
   ... = call ... @sub_jge_4004ad(...)

   ; %cond stores condition-code value computed in the
   ; previous instruction.
   br i1 %cond, label %block-4004ad, label %block-4004a1
block-4004a1:
   <glue code>
   ; Code:  movl $0x1, -0x4(%rbp), RIP: 4004a1, bytes:  7
   ... = call ... @sub_movl__0x1__MINUS0x4__rbp_(...)

   <glue code>
   ; Code:  jmpq 4004b4, RIP: 4004a8, bytes:  5
   ... = call ... @sub_jmpq_.L_4004b4(...)

   br label %block-4004b4
block-4004ad:
   <glue code>
   ; Code:  movl $0x0, -0x4(%rbp), RIP: 4004ad, bytes:  7
   ... = call ... @sub_movl__0x0__MINUS0x4__rbp_(...)

   ...
}
```

Listing 4.2: LLVM IR lifted using Compositional Lifter

For a control-flow instruction, like jump (resp., call), in addition to appending the (validated) IR sequence for the instruction, we generated the LLVM **`br`** (resp., **`call`**) instruction for the control-flow to jump to block(s) of code corresponding to the jump (resp., call) target address(es). Additionally, we have to use some logic, called *glue-code*, which is used to stitch the arguments and return-values of the called functions to the McSema `State` structure which is passed as an argument to the parent `composedFunc`. Note that this *glue-code* is specific to McSema (or, generally speaking, the lifter under test) and will be different for other lifters.

Below we provide details of the step "Compose" using a few example binary instructions. The composed program is initially empty. Upon encountering a function label, we append the following code to it[7], with irrelevant arguments omitted using "...".

```
define %struct.Mem* @composedFunc(%struct.State* %st, ..., %struct.Mem* %mem)
    {}
```

For an instruction **`adcq %rax, %rbx`**, McSema generates the following IR sequence when lifted in isolation, with irrelevant code portions omitted using "...".

```
define internal %struct.Mem* @ADCImpl(
    %struct.Mem*, %struct.State*, i64*, i64, i64) {
  ; Does adc computation and updates destination RBX
  ; and flags (omitted for brevity)
}

define %struct.Mem* @sub_adcq_rax_rbx(%struct.State* %st, ...,
        %struct.Mem* %mem) {
 %RIP = getelementptr ..., %st, ... ; compute simulated RIP addr
 %RAX = getelementptr ..., %st, ... ; compute simulated RAX addr
 %RBX = getelementptr ..., %st, ... ; compute simulated RBX addr
 %VAL_RBX = load i64, i64* %RBX
 %VAL_RAX = load i64, i64* %RAX

 ; RIP update based on instruction size
 %VAL_RIP = load i64, i64* %RIP
 %UPDATED_RIP = add i64 %VAL_RIP, 3 ; instr. len=3 bytes
 store i64 %UPDATED_RIP, i64* %RIP

 %retval = call %struct.Mem* @ADCImpl(
        %struct.Mem* %mem, %struct.State* %st, i64* %RBX,
        i64 %VAL_RBX, i64 %VAL_RAX)

 ret %struct.Mem* %retval
}
```

The above sequence is then validated using single-instruction translation validation unless it is already validated. Next, the validated IR sequence is appended to the composed program as shown below.

[7] *mem* is pointer to an opaque struct type which together with return type allows ordering of memory operations if required.

```
define %struct.Mem* @composedFunc(%struct.State* %st, ...,
        %struct.Mem* %mem)  {
    %MEM = alloca %struct.Mem*
    store %struct.Mem* %mem, %struct.Mem** %MEM

    ; Code: adcq %rax, %rbx
    %loadMem = load %struct.Mem*, %struct.Mem** %MEM
    %retval = call %struct.Mem* @sub_adcq_rax_rbx(
        %struct.State* %st, i64 0, %struct.Mem* %loadMem)
    store %struct.Mem* %retval, %struct.Mem** %MEM

    ret %struct.Mem* retval
}
; Definitions of called functions omitted for brevity
```

A similar composition happens for *all* the non control-flow instructions. For a control-flow instruction, like jump (resp., call), as mentioned earlier, in addition to appending the (validated) IR sequence for the instruction, we need to generate the LLVM **`br`** (resp., **`call`**) instruction for the control-flow to jump to block(s) of code corresponding to the jump (resp., call) target address(es). The composition for instructions accessing data-section are handled differently and elaborated next.

Composing data-section access instructions Instructions accessing the data section, like **`movq 0x602040, %rdi`** with the first operand being an address, cannot be lifted correctly in isolation because McSema does not have the full-program context to determine if the immediate operand is an integer or address. Depending on which section of the isolated binary executable the address belongs, it can be interpreted as a integer or an address[8]. However, the problem is the program-level validation may not use that lifting because the interpretations of the immediate operand, when lifted in isolation versus when lifted with full-program context, might be different. As a result, the composed IR, which consumes the translations of instructions in isolation, will be different from the one lifted by McSema. Upon optimization using LLVM passes, two such IRs will be optimized differently and eventually fail to match even when the translation of McSema is correct.

To aid in testing, we compile binaries with options to retain auxiliary information. To disambiguate between cases where an immediate operand is a reference into the data section (e.g., an `int*`) v/s a scalar (e.g., an `int`), we use relocation information, denoted by `R`

[8]During single-instruction translation validation, we validated the behavior of such instructions for both the possibilities of the constant operand by forcing the constant to belong to `.data` or `.text` section of the isolated binary.

in algorithm 4.1. Every immediate operand that is a reference has a corresponding entry in the relocation table. We allow McSema *to incorrectly lift such instructions in isolation when invoked by algorithm 4.1*, and then we course-correct the lifted IR by consulting the relocation information, R.

For example, the incorrect IR generated by McSema when lifting **`movq 0x602040, %rdi`** in isolation is:

```
define %struct.Mem* @sub_movq_0x602040___rdi(%struct.State* %st, ...,
        %struct.Mem* %mem) {
    ...
    %retval = call %struct.Mem* @MOVImpl(
        %struct.Mem* %mem, %struct.State* %st,
        ; data-section addr 0x602040
        ; lifted as a constant
        %i64* %RDI, i64 6299712)

    ret %struct.Mem* %retval
}
```

The address relocation information in the binary allows us to identify the address and correct the lifted output as shown below:

```
%G_0x602040_type = type <{ [8 x i8] }>
@G_0x602040= global %G_0x602040_type zeroinitializer
define %struct.Mem* @sub_movq_0x602040___rdi(%struct.State* %st, ...,
        %struct.Mem* %mem) {
    ...
    %retval = call %struct.Mem* @MOVImpl(
        %struct.Mem* %mem, %struct.State* %st,
        %i64* %RDI, i64 ptrtoint(%G_0x602040_type* @G_0x602040 to i64))

    ret %struct.Mem* %retval
}
```

We reiterate that Compositional Lifter only uses relocation information to strengthen the generated golden reference, T′, when such information is available, e.g., during test or development time. This allows for a tighter specification, allowing our technique to find bugs (e.g., if the lifter is not able to correctly disambiguate an address from a integer) at testing that would otherwise be missed. During the use of Compositional Lifter in the field to validate the lifting of McSema on an unknown, blackbox binary, we can function without the additional information, at the cost of potentially missing bugs described above. Note that this is a fundamental limitation because x86-64 semantics for an instruction has no notion of types, and therefore `T′`, which is based on x86-64 semantics, should allow for the ambiguity and cannot enforce stricter type requirements. McSema, on the other hand, is never given this additional information as it is expected to work in the field where relocation information is rarely available, except in library code.

4.4.2 Transformer & Matcher

Algorithm 4.2 summarizes our overall strategy to check equivalence between the IRs generated by McSema (`T`) and Compositional Lifter (`T′`). Due to the nature of the composition, `T` & `T′` are structurally very similar. We build on this observation to develop an inexpensive semantic equivalence checker that does not require heavyweight symbolic execution or theorem proving, instead using graph isomorphism, assisted by semantics-preserving transformations (lines 2-3). The algorithm is realized by a tool we develop called the Matcher (line 4).

At first, the function pair (`F` & `F′`) is transformed to ($\mathtt{F}_N$, $\mathtt{F}'_N$), using LLVM optimization passes[9], to prune any syntactic differences except for the names of virtual registers and the order of non-dependent instructions. There is clearly some important relationship between the syntactic code differences in `T` and `T′` and the choice of optimization passes with the aim of exploiting those differences. As a few examples of syntactic differences: (1) Program counter updates like `%rip` − `C` (`C` being a positive constant) are lifted in `T′` using addition (`%rip` + (−`C`)) versus subtraction used in `T`, and (2) As an optimization, `T` hoists the address computations of simulated registers to the entry block which are then dereferenced at every use-site. On the other hand, in `T′`, such addresses are both recomputed and dereferenced at every use-site. Above syntactic differences are eliminated using (1) `-instcombine` (a peephole optimization pass on LLVM IR), and (2) `-early-cse` or `-licm` respectively.

Next, the Matcher algorithm works on data dependence graphs, $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$, generated from $\mathtt{F}_N$ & $\mathtt{F}'_N$. A vertex of the graph represents an LLVM instruction, and an edge between two vertices captures SSA def-use edges or memory dependence edges between LLVM load and store instructions, extracted from LLVM MemorySSA [209] analysis. If the Matcher fails to match `T` & `T′`, there *may* be a bug in the lifter.

Checking Graph Isomorphism Our algorithm to check the isomorphism of $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$ is built on a subgraph-isomorphism algorithm from Saltz et al. [210], named *dual-simulation*

[9]The pass sequence (-mem2reg -licm -gvn -early-cse -globalopt -simplifycfg -basicaa -aa -memdep -dse -deadargelim -libcalls-shrinkwrap -tailcallelim -simplifycfg -basicaa -aa -instcombine) is determined by manually pruning the LLVM `-O3` sequence.

Algorithm 4.2: Matcher Strategy

```
Inputs :  T: McSema-lifted IR.
          T′: Compositional Lifter lifted IR.
Output: True $\implies$ T & T′ semantically equivalent
        False $\implies$ T & T′ may-be non-equivalent
1 foreach corresponding function pair (F,F′) in (T, T′) do
2     $F_N$ = Transformer (F)
3     $F'_N$ = Transformer (F′)
4     if !Matcher($F_N$, $F'_N$) then
          // A potential bug in McSema while lifting F
5         return false
6 return true
```

(refer Algorithm 4.3). The algorithm, in general, first retrieves an initial potential-match set, Φ, for each vertex in one graph based on semantic and/or neighborhood information in the other graph. In our case, the initial potential-match set for a vertex I_N in G_{F_N} contains all the vertices in $G_{F'_N}$ which satisfy the following three criteria: (1) they have the same instruction opcode, (2) they have identical constant operands, if any, and (3) they have the same number of outgoing data dependence edges[10] as I_N. Then, the algorithm iteratively prunes out elements from the potential match set of each vertex based on its parents/child relations until it reaches a fixed-point. Our overall algorithm, *Matcher*, checks that the graphs G_{F_N} and $G_{F'_N}$ are isomorphic and the instructions corresponding to the matching vertices are identical w.r.t. the instruction opcode and constant operands (note that all other operands are SSA variables, and so are validated by graph isomorphism).

Soundness of Equivalence via Graph Isomorphism Our argument that isomorphism of G_{F_N} & $G_{F'_N}$ implies semantic equivalence of the functions F and F′ is based on the pioneering work by Horwitz et al., which proved that *if the program dependence graphs of two programs are isomorphic then the programs are "strongly" semantically equivalent* [106]. Our dependence graph representations G_{F_N} and $G_{F'_N}$, which we check for isomorphism, only include the data dependences, and not the control dependences. Note, however, that the close structural similarity between the functions, F & F′, ensures the required equivalence of all control flow, as explained below. There is one minor exception to control-flow equivalence, which introduces no semantic differences between the programs, and is addressed below. We

[10]Checking outgoing but not the incoming edges is a design choice. The latter check will constrain the average size of the sets even more which in turn improve the runtime of the algorithm, but will not affect the soundness of the matcher in any way.

Algorithm 4.3: Dual Simulation

```
Inputs :
        $\mathtt{G}_{\mathtt{F}_N}$: data-dependence graph of N.
        $\mathtt{G}_{\mathtt{F}'_N}$: data-dependence graph of N'.
Output: Check if $\mathtt{G}_{\mathtt{F}_N}$ is subgraph-isomorphic to $\mathtt{G}_{\mathtt{F}'_N}$
1  changed ← true
2  while changed do
3      changed ← false
4      for $u \leftarrow G_{F_N}$ do
5          for $u' \leftarrow G_{F_N}.adj(u)$ do
6              $\phi'(u') \leftarrow \emptyset$
7              for $v \leftarrow \phi(u)$ do
8                  $\phi_v(u') \leftarrow \mathtt{G}_{\mathtt{F}'_N}.\mathrm{adj}(v) \cap \phi(u')$
9                  if $\phi_v(u') = \emptyset$ then
10                     remove $v$ from $\phi(u)$
11                     if $\phi(u) = \emptyset$ then
12                         return $\emptyset$
13                     changed ← true
14                 $\phi'(u') \leftarrow \phi'(u') \cap \phi_v(u')$
15             if $\phi'(u') = \emptyset$ then
16                 return $\emptyset$
17             if $\phi'(u')$ is smaller than $\phi(u')$ then
18                 changed ← true
19             $\phi(u') \leftarrow \phi(u') \cap \phi'(u')$
20 return $\emptyset$
```

first assume this exception does not occur, and informally prove semantic equivalence in three simple steps, as follows.

Let `PDG(f)` denote the program dependence graph of a function `f`.

(A) **The control flow graphs (CFGs) of F and F′ are isomorphic:** `F` and `F′` are both obtained by lifting the same binary, via instruction-by-instruction lifting (using identical IR sequences for each one). For `F`, we check that the order of lifted IR sequences is the same as the order of binary instructions within each corresponding basic block[11]. For `F′`, such an

[11]The checker is based on the fact that, for each binary instruction, McSema uses a specific templatized function in its lifting which realizes the operational semantics of that instruction. The implementation of the checker involves (1) Identifying the mapping between a binary instruction and the corresponding templatized function. Given the fact that (a) the lifted IR sequence `S`, for each binary instruction `I`, in `F′` is generated by McSema and hence includes the corresponding templatized function, and (b) `S` is the correct translation of `I` by virtue of the single-instruction translation validation, we identified the mapping by following how

order is already preserved assuming the correctness of `D′`. The control-flow edges are verified to be identical by the single-instruction translation validation of control-flow instructions (Section 4.3). Together, these facts ensure isomorphism of the CFGs of `F` & `F′`. We note that the requirement to preserve the order is stricter than necessary because data-independent instructions can be reordered safely.

(B) **The control dependence graphs of `F` and `F′` are isomorphic**: This is straightforward to derive using (A) and the definitions of control flow and control dependence [211], and we omit the explanation.

(C) **If the data dependence graphs $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$ are isomorphic, then $\mathtt{PDG(F}_N\mathtt{)}$ and $\mathtt{PDG(F}'_N\mathtt{)}$ are isomorphic**: By definition, the nodes of $\mathtt{G}_{\mathtt{F}_N}$ are identical to the nodes of $\mathtt{PDG(F}_N\mathtt{)}$, and similarly for $\mathtt{G}_{\mathtt{F}'_N}$ and $\mathtt{PDG(F}'_N\mathtt{)}$. The edges of a PDG are simply the union of the control dependence edges and the data dependence edges. Combining (B) with the isomorphism of $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$, it follows directly that $\mathtt{PDG(F}_N\mathtt{)}$ and $\mathtt{PDG(F}'_N\mathtt{)}$ are isomorphic.

The one exception mentioned above is that, as a custom optimization, some address computations for simulated registers are hoisted to the entry block by McSema (i.e., in `F`), to be reused by later instructions throughout the function, whereas this hoisting does not happen in `F′`. The addresses are computed using LLVM's `getelementptr` instructions whose operands are immutable throughout the function in both `F` and `F′` (the *State* pointer (Section 4.3) and some constant arguments). As a result, the results of these computations are unaffected by their location in the code. One requirement is that these address computations must dominate their uses since their results are assigned to SSA values: this property is enforced by McSema by running the LLVM `verify` pass (which we also consider trusted). Together with the isomorphism of the data dependence graphs, this guarantees that the potential difference in locations of these instructions does not introduce any differences in any uses of those values, and thus no differences in the semantics of the two functions.

Note that the entire above argument (indeed, the theorem of Horwitz, et al. [106]) is independent of the precision of any static analysis used to identify memory dependences. A highly imprecise analysis (e.g., one that says every store-load or store-store pair may be aliased) might lead to a failure to prove isomorphism between `T` and `T′`, but will not claim isomorphism if the two programs are not equivalent. In practice, we find in our experiments, described in Section 4.5, that the memory dependence edges from such a highly imprecise analysis do indeed reduce the success rate of the Matcher, but only by a small amount. A more precise analysis may improve the success rate, reducing the number of false alarms.

individual binary instructions are lifted in `F′`, and (2) Checking, for each corresponding basic blocks, if the order of binary instructions matches with the order of the corresponding templatized functions in `F`.

Autotuning-based Transformer As per our matching strategy, in order to prove that two functions `F` & `F'` are semantically equivalent, they need to be reduced to isomorphic graphs via semantic preserving transformations. For transformations, we initially used a custom sequence of 17 LLVM optimization passes, discovered manually by pruning the LLVM `-O3` search space. Later experimentation revealed that (1) changing the order of passes improves the number of functions that are successfully proved isomorphic (the phase-ordering problem of optimization), and (2) not all of the 17 passes are needed for every pair of functions under equivalence check. These two observations motivate us to frame the problem of selecting optimal pass sequences, one for every pair of candidate functions, as an application of program autotuning.

We used the OpenTuner [212] framework to implement the autotuner. OpenTuner requires the client to specify a search space to explore, and an objective function to maximize. Our search space is all permutations of passes from the 17-length pass sequence. The objective function in our case is to maximize the fraction of nodes in $\mathtt{G}_{\mathtt{F}_N}$ (or $\mathtt{G}_{\mathtt{F'}_N}$) having non-empty initial potential-match sets. The framework then uses various heuristic search techniques to find the best configuration that maximizes the objective function, within a given resource budget (a fixed number of iterations). Such an autotuning-based Transformer addresses the phase-ordering problem, improving the Matcher results (refer to Section 4.5) by lowering the false-alarm rate, and also by using much fewer than 17 passes on average.

Comparison with LLVM-MD & Peggy At this point, it is important to differentiate our approach to establish equivalence between two LLVM IR programs from existing, similar approaches for validating LLVM IR-to-IR optimization passes, e.g., LLVM-MD [171] and Peggy [172]. Like our approach, these tools eschew simulation proofs and instead use graph isomorphism techniques to prove equivalence. Both build graphs of expressions for each program, transform the graphs via a series of "expert-provided" rewrite rules, and check for equality. The rewrite-rules mimic various compiler-IR optimizations, and hence the technique is precise when the output program is an optimization of the input program, and the optimizations are captured by the rewrite rules.

Compared to these approaches, the implementation of our Transformer is simpler, requires no additional implementation effort, re-uses existing, well-tested compiler passes, and still proves to be quite effective in reducing two semantically equivalent programs to isomorphic graphs, as demonstrated by our evaluations.

4.5 EVALUATION

In this section, we present the experimental evaluation of single-instruction translation validation and program-level validation. All the experiments are run on an Intel Xeon CPU E5-2640 v6 at 3.00GHz and an AMD EPYC 7571 at 2.7GHz. We aim to address three questions through these experiments:

Q1. Is single-instruction validation by itself useful for finding bugs in a sophisticated decompiler, even though no context information is used during lifting?

Q2. What fraction of function translations are successfully proven correct by program-level validation, and what is the false alarm rate of the tool?

Q3. What is the runtime of our Compositional Lifter and Matcher-based approach?

Q4. Is program-level validation effective at finding additional real bugs in a complex lifter like McSema, beyond those found by single-instruction translation validation alone? We studied this question using artificially injected bugs because all real bugs were caught by single-instruction translation validation.

Usefulness of single-instruction translation validation: The goal here is to validate the lifting of individual x86-64 instruction to LLVM IR sequences using McSema. Haswell x86-64 ISA supports a total of 3736 instruction variants, of which 3155 are formally specified in [86]. McSema supports 1922 instructions, all supported by [86]. We had to exclude 573 instruction variants because of limitations of the LLVM IR semantics [94], which does not support vector and floating-point types and associated operations, and various intrinsic functions[12]. This brings us to a total of 1349 viable instruction variants, and we apply translation validation to each of them individually.

Out of the 1349 translation validations, 29 cases fail (hence are bugs), producing a counterexample for each failure, and 6 timed out. Except for timeouts, the solver found conclusive results in all the cases within a solver timeout of 30 secs. The solver time ranges from $0.25-29.89$ secs with median 0.46 secs. The max time, recorded for **`cmpxchgq %rcx, %rbx`**, is because of the complex summary of the corresponding lifted IR. Another significant factor dominating the performance of single-instruction translation validation is the time to generate the symbolic summaries. For generating those summaries we are using the symbolic execution engines auto-generated right from the x86-64 and LLVM IR semantics. Table 4.1

[12]However, we support an intrinsic called llvm.ctpop by implementing it in LLVM IR. This intrinsic is used pervasively in the lifted IR for updating the `%pf` flag.

presents the times (min/max/median/mean) spent during the different stages of the single-instruction translation validation pipeline, viz. creating symbolic summaries, and dispatching verification queries to solver, over 1349 validations. The reported performance of the symbolic executions are not optimized, and there is room for improvement. For example, the current symbolic-execution engines are implemented using a Java back end and more efficient back-ends (using LLVM & OCaml) are under development and we believe that they will make a difference in the performance.

	Symbolic Execution Time on x86-64 instruction (in secs)	**Symbolic Execution Time on LLVM IR (in secs)**	**Solver (in secs)**
Min	102.92	99.93	0.25
Median	503.195	199.295	0.46
Max	837.22	319.57	29.89
Mean	478.8483	204.4024	0.849

Table 4.1: Distribution of time during different stages of the single-instruction translation validation pipeline

Timeouts, declared based on a threshold of 24 hrs, correspond to **`paddb`**, **`psubb`**, and **`mulq`** family of instructions. On further investigation, we found that 4 out of 6 timeouts related to **`paddb`** and **`psubb`** are flaky: the $\mathbb{Z}3$ solver result toggled between *unknown* and *unsat* depending on the order in which other unrelated constraints are added (which is a known issue [213]). By removing the unrelated constraints[13], $\mathbb{Z}3$ concludes them to be equivalent. The remaining two cases (related to **`mulq`**) include solver constraints containing bit-vector multiplication, which the state-of-the-art SMT solvers are not very efficient at reasoning about. However, we manually inspected them to ensure that the generated code fragments are indeed semantically equivalent.

The 29 failures along with the test cases created from $\mathbb{Z}3$'s counterexamples were all reported and subsequently confirmed as bugs [214] by the McSema developers. The following are some brief examples of a few of the discrepancies we found.

- **`xaddq %rax, %rbx`** expects the operations (1) temp ← `%rax` + `%rbx`, (2) `%rax` ← `%rbx`, and (3) `%rbx` ← temp, in that order. McSema performs the same operation differently as (A) old_rbx ← `%rbx`, (B) temp ← `%rax` + `%rbx`, (C) `%rbx` ← temp, and (D) `%rax` ← old_rbx. This will fail to work when the operands are the same registers.

[13]The unrelated constraints refer to the verification queries related to registers/flags other than the one under verification.

- For instruction **`andnps %xmm2, %xmm1`**, the Intel Manual [84] says the implementation should be `%xmm1` ← ∼`%xmm1` & `%xmm2`, whereas McSema interchanges the source operands.

- For **`pmuludq %xmm2, %xmm1`**, both the higher and lower double-words of the source operands need to multiply, whereas McSema multiplies just the lower double-words.

- For **`cmpxchgl %ecx, %ebx`**, McSema compares the entire 64-bit `%rbx` (instead of just `%ebx`) with the accumulator `Concat(0x00000000, %eax)`.

- For **`cmpxchgb %ah, %al`**, the lower 8-bits of `%rax` should be replaced with the higher 8-bits at the end of the instruction, whereas McSema keeps them unchanged.

Program-level validation: Success rate and false alarms: The goal here is to validate the translation of programs, one function at a time, using the Matcher strategy (Section 4.4.2). For this purpose, we use programs from LLVM-8.0 "single-source-benchmarks". The benchmark suite consists of a total of 102 programs, of which 11 cannot be lifted by McSema due to missing instruction semantics. The remaining programs contain 3062 functions in total. We excluded 714 functions because the corresponding binary uses floating-point instructions, which are not supported in the LLVM formal semantics and hence could not be validated using single instruction validation. This brings us to a total of 2348 usable functions, which we compile using Clang[14] and feed the binaries to Compositional Lifter and McSema for lifting. The source LOC of the usable functions ranges from $1 - 1454$, with median 18, whereas the LOC of the corresponding lifted IR (`*.ll` assembly files) after inlining, obtained by lifting the binaries compiled from the source functions, ranges from $86 - 32105$ (median 656), with maximum LOC recorded for function "himenobmtxpa::jacobi".

The lifted function pairs are then optimized using the pass sequence (of length 17) and fed to the Matcher (Algorithm 4.2). Of the 2348 usable functions, the Matcher can prove the correctness of the translations for 2189 functions using graph isomorphism, i.e., a success rate of 93% (the inlined lifted IR ranges in size from $86 - 32105$, with the median as 611). We manually checked the remaining 159 and found them to be false alarms, with the following root causes:

- **Pass Selection & Phase-ordering problem (80% of false alarms)**: The fixed-length pass sequence is not able to converge functions into isomorphic graphs because either it missing some key passes or the order of application of individual passes is not

[14]We also used GCC-compiled binaries for the experiments, but most of them are not lifted by McSema due to unsupported instruction semantics.

effective. We addressed the above problems using autotuning of the pass-sequence, as described below.

- **Difference in Lifting globals (20% of false alarms)**: For data section addresses, McSema lifts a global with over-approximated size (as determined by IDA) which need not be equal to the actual source code size, whereas our Compositional Lifter determines the size as the width of the maximum access across all the instructions accessing that particular global. As a result, the lifted global sizes might be different from McSema. The memory dependence edges that we extract using LLVM IR `memory-ssa` analysis depend on the size of the globals, and hence the generated graphs will be different. A more accurate memory analysis might solve these issues.

Overall, a false alarm rate of about 7% is low enough that we believe our Matcher can be of practical use for validation and testing of a lifter. We can further reduce this rate by addressing the phase-ordering problem using an autotuner, as described in Section 4.4.2. We leveraged the experience and effort put into custom-designing the fixed-length pass sequence by including the constituent passes in the search space for autotuning. We avoided crafting the search space using all the LLVM passes (e.g., 187 passes of Clang's $-O3$ pass sequence) because our experiments showed that such a large search space was *less effective* at avoiding false negatives in a fixed number of iterations.

For 2254 out of the total 2348 functions, the autotuner is able to find custom pass sequences that lead to successful matching. These matches include 65 previously reported false alarms (out of total 159), reducing total false alarms to 94 (or 4% of 2348). All previously positive cases remain positive with the autotuner, as well. The autotuner runtime ranges from 10.7 secs - 19.97 mins, with a median of 6.67 mins. The length of the generated pass sequence has distribution of [min:- 3, median:- 7, max:- 243, mean:- 8][15].

Judiciously adding LLVM passes to the search might help remove false alarms further. We leave this as future work.

Performance of Program-level validation: The performance of this phase is dominated by the time to run the Compositional Lifter and Matcher.

We chose to validate the individual instructions offline after program-level validation. The number of those instructions, to be validated offline, amounts to approx. 50% of the total

[15] 43 out of 65 newly matched cases have an auto-tuned pass sequence of length greater than 17. For those cases, the search space, with 17 passes being ineffective, is composed differently out of multiple auto-tuned sequences derived from the other matching cases.

1349 variants we validated using single-instruction translation validation; implying that only a small subset of instructions are actually used in binary executables in practice.

The running time of the Compositional Lifter, on 2348 usable functions, ranges from $0.06s - 5.75s$, with a median of $0.63s$. Note that this performance depends heavily on the availability of instructions in the Store database for reuse, which in turn depends on the order in which the test functions are executed. For example, a large function with many commonly occurring instructions, if lifted first using the Compositional Lifter, will populate the Store sufficiently to create good reuse in later functions. Even with an arbitrary order of test execution, the Store reuse found is significant. Figure 4.5 shows the distribution of reuse percentage (both across & within functions) for an arbitrary order of lifter execution on 2348 test-functions.

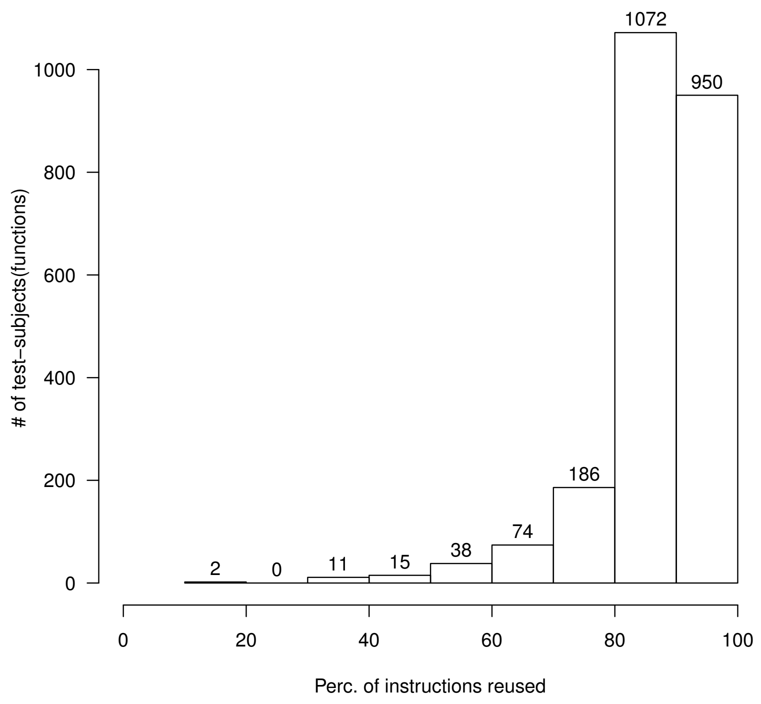

Figure 4.5: Distribution of reuse % by the Compositional Lifter in the Store database for an arbitrary execution sequence on 2348 test-functions.

The running time of Matcher, primarily the graph isomorphism algorithm, on 2348 usable functions ranges from $0.06s - 119.63s$, with a median of $4.91s$. We note that, for both Compositional Lifter & Matcher, the max time occurred for the largest function "himenobmtxpa::jacobi" (32105 LLVM assembly LOC).

Program-level validation: Effectiveness at finding bugs: In our experiments, all real bugs in McSema were caught by single-instruction translation validation and not program-level validation, which may be evidence that most of the complexity in lifting, by far, lies in lifting individual binary instructions to IR. Intuitively, this makes sense because of the large and diverse instruction set semantics, the simplicity of the compositional step, and the fact that every aspect of the composition logic is likely used hundreds of times per program. Nevertheless, bugs are possible in this aspect of the lifter.

We studied the effectiveness of program-level validation in finding bugs in McSema by artificially injecting bugs in the lifter's implementation. The injected bugs cover the following aspects of McSema's lifting: (1) *Instruction lifting*: McSema uses code templates to generate IR sequences for each instruction. The injected bug forces the tool to choose wrong templates. The injected bug is targeted to affect the translation of 491 unique instruction mnemonics that we collected from the compiled binaries of our evaluation test-suite. (2) *Inferring data-section access constants*: McSema uses information from IDA [215] to know if an immediate operand used in a data-section access instruction is a constant or a memory address. The introduced bug forces McSema to take the wrong decision. (3) *Maintaining correct dependences among instructions*: The injected bug changes the order in which instructions are lifted, potentially violating data and control dependences between instructions.

Each of the above bugs are injected one at a time and in combination and the resulting buggy lifter is tested against the Compositional Lifter on the same evaluation test-suite mentioned before. All the injected bugs are correctly detected by the Matcher, establishing program-level validation as a complementary technique to single-instruction translation validation in finding bugs during lifting.

Note that only the first of these bugs would be caught by single-instruction translation validation: the binary instruction semantics would not match with the LLVM IR sequence semantics in that case. The second and third cases would (in general) produce equivalent semantics between each X86 instruction and the LLVM IR sequence, and so single-instruction translation validation would not detect the bug.

4.6 LIMITATIONS

In this section, we discuss some limitations of our work.

Incomplete LLVM Semantics The LLVM IR semantics [94] is currently under development and does not support all LLVM abstractions, e.g., vector and floating-point types and their associated operations, and various intrinsic functions at the time of writing the thesis. This is a limitation of existing semantics and we believe the verification of lifted instructions that use such unsupported features will work out-of-the-box when semantic rules are added, assuming $\mathbb{Z}3$ supports the requisite features.

Formally Verifying Transformation Passes Our current implementation uses a small number of LLVM passes (17) to improve syntactic matching between the IR generated by McSema and by Compositional Lifter. For now, we trust the correctness of these passes to perform only semantics-preserving transformations. We also trust the LLVM `verify` pass to check SSA dominance for the McSema-lifted `getelementptr` instructions (discussed in Section 4.4.2 as part of the soundness proof of the Matcher algorithm). Formally proving correctness of arbitrary LLVM pass sequences is difficult. An alternative approach is to develop simple graph rewrites on SSA graphs that can be composed to mimic the transformations of LLVM passes and formally prove that these graph rewrites preserve program semantics. We leave this to future work.

CHAPTER 5: CONCLUSIONS

Binary analysis is a crucial task to guarantee the trustworthiness of critical systems spanning in various subfields of software engineering and security tools, including binary instrumentation, binary re-targeting, software hardening, software testing, CPU emulation, automated reverse engineering, sand-boxing, profiling, and automatic exploit generation. This is particularly necessary when the source code is not available (e.g., legacy code, closed-source software, or malware) or when it is not desirable to trust the correctness of compilers.

The need for semi-automatic analysis techniques for binary code has led to the development of several decompilation tools. To handle the complexity and diversity of modern instruction set architectures (ISAs), all such tools introduced a platform-independent intermediate representation that allows implementing analysis independently of (i) names and number of registers, (ii) instruction decoding, (iii) endianness of memory access, and (iv) instruction side-effects. Binary lifters, responsible for transforming binary code to the intermediate representation, are a complex piece of software, and it is non-trivial to implement them correctly. This is mainly because manual encoding the effects of a vast number of instructions (including their complex side effects) is laborious and error-prone. This is made even harder when the informal specifications provided by the hardware manufacturers run into thousands of pages, have mistakes, or allow for implementation-dependent undefined behaviors. Once such a lifter is developed, the developers then run into the problem of not having a way to test their implementation thoroughly as generally, there is no formal, machine-readable semantics available for automated testing. Lastly, to make it worse, these lifters need to be updated and rechecked for correctness every time new instructions are added to an ISA. Clearly, a lifter bug jeopardizes the soundness of all analyses done on the intermediate representation, and hence the soundness of the lifter should not be foregone.

In this work, we tackle the problem of increasing the assurance of binary lifters. Specifically, we develop formal and informal techniques to achieve high confidence in the correctness of a lifter from a complex machine ISA (e.g., x86-64) to a rich IR (e.g., LLVM IR).

Our work is inspired by a key observation that decompilers are usually designed to perform simple instruction-by-instruction lifting (using a fixed and canonical representation of architectural state at the IR level), followed by standard IR optimization passes to achieve a simpler IR code. We capitalize on this observation by exploiting the insight that *formal translation validation of single machine instructions can be used as a building block for scalable full-program translation validation.*

With that insight, we first define a fully executable formal semantics of x86-64 using 𝕂 as a language engineering framework, to assist translation validation of individual instructions. This is the most complete formal semantics of x86-64 user-level instructions to date, which have been thoroughly tested using synthesized test inputs and the GCC torture tests. While defining the semantics, we found 8 bugs in the Intel manual, the standard document on x86-64. All these findings were reported and acknowledged by Intel as bugs in the manual and are fixed in the latest revision. Also, we found 42 bugs in existing semantics like Strata and Stoke, which are all reported to the authors, acknowledged by them and some have been fixed. We have also illustrated several potential uses of the semantics, which are realized by the formal analysis tools derived right from the 𝕂 specification. The 𝕂 framework also enables us to represent the semantics in SMTLIB theories, which other projects can leverage for their purposes.

Next, we develop the first single-instruction translation validation framework for x86-64 built on top of the developed formal semantics. We validate the translational correctness of a single instruction by asserting the equivalence of symbolic summaries of the x86-64 instruction and the lifted LLVM IR sequence using an SMT solver. If they are equivalent, the lifted LLVM IR sequence is correct for the instruction, else the equivalence check fails, and the solver generates a counter-example that we use to report a bug. We experimentally verify that such single-instruction validation is effective in finding real bugs, and in particular, we find bugs in the lifting of 29 instructions in McSema, a well-tested, actively maintained, and open-source lifter for x86-64 to LLVM IR. The McSema developers have confirmed all of these bugs and fixed some of them. Finally, we demonstrated that validation of lifters without instrumentation or heavyweight equivalence checking is feasible. The design is based on a simple insight: Formal translation validation of single machine instructions can be used as a building block for scalable full-program validation, achieving scalability by using symbolic execution and theorem provers only for the single-instruction case. Our experimental evaluation shows that single instruction validation is valuable in finding real bugs in McSema, a popular open-source lifter from x86-64 to LLVM IR. We construct an alternate lifter by composing validated single-instruction translation sequences, with a small amount of custom logic to handle control flow sequences and global data. To check the McSema translation, we compare the outputs of the two lifters, using semantics-preserving program transformations together with graph isomorphism of data dependence graphs. We believe our approach can be easily modified to support other lifters from x86-64 to LLVM that are designed to translate individual instructions, simply by modifying how the alternate compositional lifter is constructed.

CHAPTER 6: FUTURE DIRECTIONS

This dissertation presented the need for developing scalable solutions to validate binary to IR lifting and demonstrated some techniques and tools to achieve that. We devote this chapter to discuss the opportunities that we foresee to improve the start-of-art in validation binary decompilers. The opportunities include (1) improving the current work in validating binary lifters, (2) extend and evaluate the efficacy of the current approach to other lifters, and (3) evaluating the correctness of disassemblers using ISA semantics.

6.1 IMPROVING THE CURRENT WORK

In this section, we enlist some limitations of our work and discuss avenues to improve on the current work targeting the performance of the single-instruction translation validation, faithfulness of the Transformer, and efficiency of the Matcher in reducing false alarms.

Improving Single-instruction translation validation The LLVM IR semantics [94] is currently under development and does not support all LLVM abstractions, e.g., vector and floating-point types and their associated operations, and various intrinsic functions at the time of implementation. This is a limitation of existing semantics, and we believe the verification of lifted instructions that use such unsupported features will work out-of-the-box when semantics are available.

Formally Verifying the Normalizer Our current implementation of the Transformer uses a small number of LLVM passes to facilitate matching between the McSema generated IR and the one proposed by Compositional Lifter. However, an unsound transformation pass can optimize two non-equivalent functions in a way allowing the Matcher to succeed, thereby erroneously claiming that the programs are equivalent. To prove soundness, these passes need to be formally verified to perform only semantic preserving transformations. Existing approaches like LLVM-MD [171] and Peggy [172] transform the graphs via a series of “expert-provided” rewrite rules, and check for equality. The rewrite-rules mimic various compiler-IR optimizations, and hence the technique is precise when the output program is an optimization of the input program, and the rewrite rules capture the optimizations. A promising approach is to develop or borrow similar simple graph transformations on SSA graphs to mimic the transformations of LLVM passes, but formally prove that the transformations preserve program semantics.

Efficient Matching Strategies In our current work, we use IR transformations, using 17 optimization passes, to prune away most of the syntactic differences of the candidate LLVM IRs. Next, the data dependence graphs, extracted from the syntactically-close-enough transformed outputs, are compared using isomorphic graph matching. The isomorphism of data dependence graphs, for each pair of optimized functions, implies that the original lifted IR `T` is semantically equivalent to the reference translation `T′`, and therefore `T` is the correct lifting of the input binary program. Otherwise, the automatic validation fails, and mismatch reported as a potential bug for further analysis. While evaluating the strategy on LLVM "single-source-benchmark", we found all the buggy cases to be false alarms. Investigating those false alarms revealed that the order of the passes in the fixed-length pass sequence is an important factor in effective IR transformation and subsequent matching. This is the well-known pass ordering problem that we addressed using an AutoTuner [212], which uses the manually identified 17 LLVM optimization passes as the search space to automatically find the optimization passes effective in lowering the false-alarm rate to 4%.

The primary root cause of the remaining false alarms is that the optimization passes are not able to converge candidate functions into isomorphic graphs. That is either due to the absence of some relevant passes in the autotuner search space or it is the very structure of the program, along with its alias relations between instructions, which prohibits effective optimization. We propose the following two ways to counter the false alarms.

- **Efficient Autotuning:** From our experience, including relevant passes in the search space might help match more cases; however, identifying such passes is nontrivial. We deliberately avoided creating the search space straight out of the 187 passes of *opt -O3* pass sequence because it makes the search space too big for the autotuning to be effective. We believe that there are research opportunities to efficiently narrow down such an ample search space to include as many relevant passes as possible.

- **Iterative Pruning based Matcher:** Another future direction could be to use an iterative matching and pruning strategy, which iteratively prunes the matched sub-graphs and look for more isomorphic matches after canonicalizing the residual graph. This direction is based on the insight that the residual graphs will be much straightforward, in terms of the aliasing relations among instructions, for the optimization passes to canonicalize effectively. We built a prototype of the strategy which shows promising results with small toy programs (with implementations of sorting, searching algorithms & tree-traversal algorithms). We note that one of the bottlenecks in pruning sub-graphs is the presence of spurious memory dependence edges, which might prohibit pruning an otherwise matched sub-graph.

6.2 EXTENDING TO OTHER LIFTERS

Our current work focuses on McSema; however, there are other lifters [57, 66] that are not formally verified. Extending our work to support these systems is vital for two reasons: (i) improving the trust in binary lifters, and (ii) the improvements made to our system would make it more generic enough for future binary lifters to get validation for (nearly) free.

To generalize the current method to a lifter translating a particular ISA to a specific IR, we need the semantics of both the ISA and IR defined formally, which is a sufficient requirement for single-instruction translation validation. Note that it does not matter which language semantics engineering framework is used. We used $\mathbb{K}$ mainly because of familiarity and ease-of-use. For lifters that are designed to translate binary instructions individually and compose the resulting LLVM IR, we believe that this could be done simply by customizing Compositional Lifter to capture the idiosyncrasies of each lifter. Such customization is mostly related to generating some custom glue code while IR composition, as detailed in Section 4.4.1. We believe that such an engineering effort for customization is quite practical and does not overshadow the benefit of abandoning a heavyweight equivalence checker.

6.3 EVALUATING CORRECTNESS OF DISASSEMBLER USING ISA SEMANTICS

Disassembly is the backbone of research in any static binary analysis, which gets benefited if the disassembly is accurate. The output of a disassembler serves many different uses-cases (e.g., debugging and reverse engineering). Therefore, disassemblers represent the first link of a long chain of stages on which any high-level analysis of binary code depends. It has been demonstrated that many disassemblers fail to decode certain instructions and thus that the first link of the chain is fragile.

There are a few prior works that directly address this problem and are based on testing a disassembler w.r.t to some notion of ground truth. Most notable pre-work is Paleari et al. [216], where the idea is to compare the output of n disassemblers against each other, where one of them is a special instruction decoder the author developed, that leverages the physical CPU to provide accurate results, while the others are other off-the-shelf disassemblers. Disassemblers usually have two phases: (I) instruction decoding, where a piece of code is translated into a single assembly instruction and (II) selection of the next instruction of the program to decode, where the disassembler selects the next piece of code to decode according to the format (or the semantics) of the instruction previously decoded. This process is repeated until all the code is disassembled. The work by Paleari et al. is focused on automated testing in the first phase only. However, there are various challenges specific to the second phase mainly because disassemblers adopt several heuristics, assuming certain compiler conventions are respected, to tackle problems related to disassembly [53, 56, 54, 55]. When the code being disassembled violates the assumed conventions (for example, in case of obfuscated code, handwritten assembly), disassemblers produce completely unreliable results.

The basis of Paleari et al. comparison is syntactic and works well if the disassemblers decode a single instruction at a time to a mnemonic which can then be compared syntactically against other disassembler outputs. But the problem comes if the disassemblers try to decode multiple instructions at a time. Because of the challenges mentioned above, the disassemblers might generate, for a given binary blob, syntactically different instruction sequences that may (or may not) be semantically equivalent. It would be interesting to evaluate off-the-shelf disassemblers by checking the semantic equivalence of their disassembled output using the formal semantics of x86-64 ISA.

APPENDIX A: SINGLE INSTRUCTION TRANSLATION VALIDATION

In this appendix, we will elaborate on various artifacts and tools used for single-instruction translation validation. The presentation will use a running example of a logical-and instruction **`andq -32(%rsp),%rbx`** to demonstrate the validation strategy of its lifting to LLVM IR (using McSema). The instruction performs logical AND of memory (`-32(%rsp)`) and register (`%rbx`) operands and writes the result to destination register `%rbx`. For example, Figure A.1 explains the output behavior of the instruction in terms of some logical formulas over the inputs values `RBX` (of register `%rbx`) and `Mem64` (of memory location `-32(%rsp)`). According to the Intel manual [84], flags `%zf`, `%sf` and `%pf` are updated based on result of logical AND computation, `%of` and `%cf` flags are cleared, and `%af` flag is *undefined*.

```

RBX ← RBX & Mem64

ZF ← RBX & Mem64 == 0 ? 1 : 0

SF ← (msb of RBX) & (msb of Mem64)

PF ← number of set bits in least significant 8−bits of
                    (RBX & Mem64) is even ? 1 : 0

OF ← 0
CF ← 0

AF ← undef
```

Figure A.1: Formal semantics of **`andq -32(%rsp),%rbx`** instruction

As discussed in Section 4.3, one of the prerequisites for the single-instruction translation validation approach is the formal specification of x86-64. For the current discussion, we will be emphasizing on the specification relevant to the running example of **`andq`**. Figure A.2 depicts the relevant portion of the formal syntax, declared using the conventional BNF notation. The syntax defines an x86-64 program as a list of instructions, where each instruction is composed of an opcode and list of operands. Operands can be further classifies based on its type (memory/register/immediate) or size (8-, 16-, 32-, or 64-bits).

Figure A.3 presents the semantic specification of **`andq`** instruction highlighting two rewrite rules. The first rule (lines 1 – 6), is responsible for reading a memory value of size 64-bits starting at memory offset `MemOff`. The read is accomplished using a $\mathbb{K}$ function `loadFromMemory` which can read a byte addressable memory implemented using $\mathbb{K}$ builtin `map`. In the rule text, the term to the left of ⇒ shows the instruction **`andn`** in execution

```

syntax Instructions ::= List{Instruction, ""}
syntax Instruction  ::=  Opcode Operands

syntax Opcode   ::=  "andq" | ...
syntax Operands ::=  List{Operand, ","}
syntax Operand  ::=  Register | Imm | Mem | MemOffset

// register operands
syntax Register ::= Rh | R8 | R16 | R32  | R64 | Xmm | Ymm
syntax Rh       ::= "%ah" | "%bh" | "%ch"| ...
syntax R8       ::= "%al" | "bl" | "cl" | ...
syntax R16      ::= "%ax" | "%bx" | "%cx" | ...
syntax R32      ::= "%eax" | "%ebx" | "%ecx" | ...
syntax R64      ::= "%rax" | "%rbx" | "%rcx" | ...
syntax Xmm      ::= "%xmm0" | "xmm1" | ...
syntax Ymm      ::= "%ymm0" | "ymm1" | ...

// immediate operands
syntax Imm    ::= "$" Int | HexConstant

// memory operands
syntax Mem ::=  Int "(" R64 "," R64 "," Int ")" // One of the memory
                                                // addressing expressions
                | MemOffset // an effective address computed
                            // from a memory addressing expression
syntax MemOffset ::= memOffset ( MInt ) // MInt represents a bit-vector
                                        // (or Machine Integer)
```

Figure A.2: A snippet of formal syntax relevant to **andq -32(%rsp),%rbx** instruction

mode, which gets reduced to a sequence of two tasks: a call to loadFromMemory followed by the same execution instance of the instruction. In the next rewrite rule (line 8 – 37), the terms to the left of ⇒ are the ones rewritten by the previous rule except that the loadFromMemory call is replaced by the read memory value, memVal_{64}. In this rewrite rule, the two tasks at the top of the k cell are reduced to empty computation ".". The regstate cell, which hosts a map, RSMap, from register names to their values, is updated with the new values of the registers. For example, the destination register R2 (= %rbx) is updated to the logical AND of the memory value, MemVal_{64} with the value from source register operand R2 (= %rbx). Similarly, control flags are updated as shown by the other map updates. The $\mathbb{K}$ function convToRegKeys is responsible for accessing the concrete register (%rbx in this case) pointed by the symbolic register (R2). The other $\mathbb{K}$ function, getParentValue(R2, RSMap), extracts the old value of the concrete register pointer by R2 (which, in this case, is value of %rbx) from the register map RSMap. The operators `andMInt`, `extractMInt`, and `eqMInt` are respectively the $\mathbb{K}$ built-in operators for bitwise *and*, *extract* and *equality check*.

```
rule <k>
  execinstr (andq:Opcode
      memOffset( MemOff:MInt):MemOffset, R2:R64, .Operands) ⇒
    loadFromMemory( MemOff, 64) ↷
    execinstr (andq memOffset( MemOff), R2, .Operands)
 ...</k>

rule <k>
  memLoadValue(MemVal64 :MInt):MemLoadValue ↷
    execinstr (andq:Opcode
        memOffset( MemOff:MInt):MemOffset, R2:R64, .Operands) ⇒ .
...</k>
  <regstate>
    RSMap:Map ⇒ updateMap(RSMap,
      convToRegKeys(R2) ↦ andMInt(
                                getParentValue(R2, RSMap),
                                MemVal64 )

      "ZF" ↦ (#ifMInt
                  eqMInt(
                    andMInt(
                      getParentValue(R2, RSMap), MemVal64 ),
                      mi(64, 0))
                #then mi(1, 1)
                #else mi(1, 0) #fi)
      "SF" ↦ andMInt(
                 // Extracts the most significant bit of register value.
                 extract( getParentValue(R2, RSMap), 0, 1),
                 // Extracts the most significant bit of memory value.
                 extract( MemVal64 , 0, 1))
      "PF" ↦ // omitted for brevity

      "OF" ↦ mi(1, 0)
      "CF" ↦ mi(1, 0)

      "AF" ↦ undefMInt
  </regstate>
```

Figure A.3: Formal semantics of **`andq -32(%rsp),%rbx`** instruction

Figure A.4 shows a snippet of McSema-lifted LLVM IR for the example instruction. The embedded comments, using ";", highlight the *stores* of values to various simulated registers and flags. The addresses of simulated registers (e.g., `%2` (for `%rbx`), `%9` (for `%cf`), `%16` (for `%pf`) etc.) are computed using LLVM `getelementptr` instruction and the values stored in them are based on the semantic of the **`andq -32(%rsp),%rbx`** instruction.

Figure A.5 shows the LLVM syntax and semantics corresponding to the LLVM **add** instruction for integer type values[1]. The rewrite rule at line 8 is responsible for adding two LLVM typed values and wrapping around the result, using $\mathbb{K}$ function `wrap` (line 13), to the bit-width of the operation type.

```
; %1: The state structure,   %2: pointer to rbx register
; %3: value of rbx register, %4: effective address (-32 + rbp)
define internal %struct.Memory* @ANDQImpl(%struct.Memory*, %struct.State*, i64*, i64, i64) {
  ; Store the added result in simulated register %rbx
  %6 = inttoptr i64 %4 to i64*
  %7 = load i64, i64* %6
  %8 = and i64 %7, %3
  store i64 %8, i64* %2, align 8

  ; Set carry flag to 0
  %9 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 1
  store i8 0, i8* %9, align 1

  ; Set parity flag (pf)
  %10 = trunc i64 %8 to i32
  %11 = and i32 %10, 255
  %12 = tail call i32 @my.ctpop.i32(i32 %11)
  %13 = trunc i32 %12 to i8
  %14 = and i8 %13, 1
  %15 = xor i8 %14, 1
  %16 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 3
  store i8 %15, i8* %16, align 1

  ; Set zero flag (zf)
  %17 = icmp eq i64 %8, 0
  %18 = zext i1 %17 to i8
  %19 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 7
  store i8 %18, i8* %19, align 1

  ; Set sign flag (sf)
  %20 = lshr i64 %8, 63
  %21 = trunc i64 %20 to i8
  %22 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 9
  store i8 %21, i8* %22, align 1

  ; Set overflow flag (of) to 0
  %23 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 13
  store i8 0, i8* %23, align 1

  ; Set auxiliary flag (af) to 0
  %24 = getelementptr inbounds %struct.State, %struct.State* %1, i64 0, i32 2, i32 5
  store i8 0, i8* %24, align 1

  ret %struct.Memory* %0
}
```

Figure A.4: McSema-lifted LLVM IR for **`andq -32(%rsp),%rbx`** instruction

Once we have the formal specification of the syntax and semantics of an instruction (or corresponding lifted LLVM IR), the $\mathbb{K}$ framework can generate, at no additional cost, a parser, an interpreter, as well as formal analysis tools such as symbolic execution engines,

[1]Refer to [94] for the semantics of other LLVM operators.

```
syntax Type ::= IntegerType | ...    // other LLVM types are omitted
syntax IntegerType ::= tInt(Int)
syntax Val ::= Int | ... // other kinds of LLVM values are omitted

syntax TypedVal ::= typedVal(Type, Val)

// addition between ints
rule iadd(typedVal(tInt(N:Int), I1:Int), typedVal(tInt(N), I2:Int)) =>
  typedVal(tInt(N), wrap(N, I1 +Int I2))

// Wraps around the result of an arithmetic operation to convert it back
// to an unsigned representation of the given bit width
syntax Int ::= wrap(/*Bit width */ Int,
                               /* Operation result */ Int) [function]

rule wrap(N:Int, OpRes:Int) => OpRes modInt (1 <<Int N)
```

Figure A.5: Formal semantics of LLVM **`and`** instruction

model checkers and deductive program verifiers. The next step is to use the symbolic-execution engine to symbolically execute the instruction (or corresponding lifted LLVM IR) in order to generate the symbolic summary of it.

The symbolic-execution engine, auto-generated using either the x86-64 semantics or LLVM IR semantics, needs a driver specification file for its execution, which includes the specification of the input symbolic values assigned to the various operands[2] of the instruction.

Figure A.6 shows a simplified snippet of the driver specification file to run symbolic execution on **`andq -32(%rsp),%rbx`** instruction. The computation term at the top of the `k` cell, "fetch", is responsible for fetching the top instruction from the "text" memory (contents of `text` cell). The term at the right side of $\Rightarrow$ represents the "target" state where the program successfully terminates. In the `regstate` cell, input symbolic values are assigned to various registers and flags. For example, the register `%rbx` is initialized with a 64-bit symbolic value `VX_RBX` and is ensured to be any symbolic 64-bit value `mi(64, _:Int)` at successful termination of the program. Similarly, the memory at address `-32(%rbp)` is initialed with 64-bit value `VX_MEM_64`. We use a similar driver specification file to drive symbolic execution on the lifted LLVM IR (details of which are omitted for the brevity of presentation). The symbolic executions result in symbolic summaries; one for the x86-64 instruction and the other for the lifted LLVM IR.

Next, the symbolic summaries are converted to SMTLIB formula in bit-vector logic, in order to compare them for equivalence. Note that, the x86-64 semantics is built using

[2]Note that, in the context of LLVM IR, operands means the simulated register or memory location.

```
rule
<k> fetch ⇒ exit_0 </k>
<regstate>
  "RBP" ↦ ptr ( 1032 )
  "RBX" ↦ (mi(64, VX_RBX:Int):MInt ⇒ mi(64, _:Int))
  "AF"  ↦ (mi(1,  VX_AF:Int):MInt  ⇒ mi(1,  _:Int))
  "CF"  ↦ (mi(1,  VX_CF:Int):MInt  ⇒ mi(1,  _:Int))
  "OF"  ↦ (mi(1,  VX_OF:Int):MInt  ⇒ mi(1,  _:Int))
  "PF"  ↦ (mi(1,  VX_PF:Int):MInt  ⇒ mi(1,  _:Int))
  "SF"  ↦ (mi(1,  VX_SF:Int):MInt  ⇒ mi(1,  _:Int))
  "ZF"  ↦ (mi(1,  VX_ZF:Int):MInt  ⇒ mi(1,  _:Int))
</regstate>

<memstate>
  <text>
    code (
      0 ↦ storedInstr ( andq -32(%rbp), %rbx, .Operands )
      1 ↦ storedInstr ( ret .Operands ))
  </text>

  <stack>
    // 64-bit symbolic value at -32(%rbp)
    1000 ↦ byte (0,  mi(64, VX_MEM_64)))
    1001 ↦ byte (1,  mi(64, VX_MEM_64)))
    1002 ↦ byte (2,  mi(64, VX_MEM_64)))
    1003 ↦ byte (3,  mi(64, VX_MEM_64)))
    1004 ↦ byte (4,  mi(64, VX_MEM_64)))
    1005 ↦ byte (5,  mi(64, VX_MEM_64)))
    1006 ↦ byte (6,  mi(64, VX_MEM_64)))
    1007 ↦ byte (7,  mi(64, VX_MEM_64))))
  </stack>
</memstate>
```

Figure A.6: A Snippet of the driver specification file to run symbolic execution on **`andq -32(%rsp),%rbx`** instruction

the 𝕂 bit-vector operators which makes the conversion of the summary to bit-vector logic SMTLIB formula trivial. This is mainly because the bit-width (or type) of the values are explicit in the summary expression, which can be used directly to generate the corresponding SMTLIB summary. For example, Figure A.7(a), lines $3-5$, shows a snippet of the symbolic summary corresponding to the output value of destination register `%rbx`. The summary is generated by symbolic executing of the example x86-64 instruction using x86-64 semantics. The summary expression is explicit in the type of the values. Figure A.7(b) shows the converted SMTLIB expression in bit-vector theory.

However, the LLVM IR semantics is developed using the 𝕂 unbounded-integer operators and, as a result, the corresponding summary does not capture the type of the values.

```
    // symbolic summary of the ouput value of destination %rbx
    // obtained using symbolic-execution on andq -32(%rsp),%rbx
    andMInt(
      mi(#token("64","Int"),VX_RBX),
      mi(#token("64","Int"),VX_MEM_64)

                                              (a)


    // SMTLIB summary of of the ouput value of destination %rbx
    VX_RBX     = z3.BitVec('VX_RBX',64)
    VX_MEM_64 = z3.BitVec('VX_MEM_64',64)
    xvar       = (VX_RBX & VX_MEM_64)

                                              (b)
```

Figure A.7: Summaries corresponding to the value of destination register `%rbx`

```
1        // symbolic summary of the simulated register %rbx
2        // obtained using symbolic-execution on lifted LLVM IR
3        `_modInt_`(`
4        `_&Int_`(VL_MEM_64, VL_RBX),
5        #token("18446744073709551616", "Int"))
6
7        (a)
8
9        // SMTLIB summary of the simulated register %rbx
10       VL_RBX     = z3.BitVec('VL_RBX',64)
11       VL_MEM_64 = z3.BitVec('VL_MEM_64',64)
12
13       lvar = (VL_MEM_64 & VL_RBX) & 18446744073709551615_64
14
15       (b)
```

Figure A.8: Summaries corresponding to the value of simulated destination register `%rbx`

For example, Figure A.8(a) (lines 3 – 5) shows the symbolic summary corresponding to the value of *simulated* destination register `%rbx`, obtained by symbolically executing the McSema-lifted LLVM IR. In the expression, all the operators are integer operators and the type of the symbolic variable, like `VL_RBX`, `VL_MEM_64`, are known. However, the constant like 18446744073709551616 has no designated type. Such types need to be inferred from the known types. In this particular case, the type of the constant is inferred as 64-bits while converting the expression to SMTLIB summary as shown in Figure A.8(b).

There exist a few other cases where the type inference is non-trivial. Consider the expression in Figure A.9(a), where a 64-bit value `VL_RBX` is added with 1 (expr1 in Figure A.9(a)) and similarly, a 8-bit value `VL_BL` is added with 2 (expr2 in Figure A.9(a)). Finally, the results of the two additions are added using the outermost addition operator. Note that the integer modulus operator is used to constraint the result of addition to the bit-width of the operation.

```
        // an example symbolic summary
        `_modInt_`(  // To constraint the resultant value to 64 bits
          `_+Int_`(`
            `_modInt_`( // To constraint the value of VL_RBX + 1 to 64 bits
               `_+Int_`(VL_RBX,   #token("1","Int")),   // expr1
               #token("18446744073709551616","Int")
             ),
             `_modInt_`( // To constraint the value of VL_BL + 2 to 8 bits
               `_+Int_`(VL_BL,    #token("2","Int"))), // expr2
               #token("256","Int")
             )
          ),
          #token("18446744073709551616","Int")
        )

                                         (a)

        // corresponding SMTLIB summary
        VL_RBX    = z3.BitVec('VL_RBX', 64)
        VL_BL     = z3.BitVec('VL_BL', 8)
        lvar      =  (
                        ( VL_RBX + 1_64 ) & 18446744073709551615_64 ) +
                        z3.Concat(0_56 , (( VL_BL + 2_8 ) & 255_8 )
                      ) & 18446744073709551615_64

                                         (b)
```

Figure A.9: Example symbolic summaries obtained by symbolically executing the McSema-lifted LLVM IR

VL_RBX and VL_BL has known types of size 64 and 8 respectively. From the known operand types and the requirement that the SMTLIB addition operator expects operands of equal size, the types of 1 and 2 can be individually inferred as 64 and 8 respectively; resulting a type mismatch on the operands on outermost addition operator. To handle the type mismatch, the type of expr2 need to be promoted to 64-bits. The converted SMTLIB summary is shown in Figure A.9(b)[3]. This type promotion makes sense for most of the bitwise operators except *concatenation* which might take operands of different types.

This conversion is achieved using a custom developed tool "spec-to-smt". The tools takes as input the symbolic summaries of a binary instruction and corresponding lifted LLVM IR and generates verification queries in SMTLIB format. Figure A.10 shows the verification query corresponding to the destination register %rbx of the running example instruction.

[3] We note that the integer modulus operator is used to constraint the value of the added result within the limits of the bit-width in the event of overflow.

Similar verification queries are generated for each register, flag and clobbered memory value and dispatched to the $\mathbb{Z}3$ solver for equivalence check. For a particular instruction, if the result of the equivalence check is `sat` for *any* verification query, then we declare the lifting of that instruction as buggy. Otherwise, the lifting is deemed as the correct translation of the binary instruction.

```
// verification query  corresponding to destination %rbx
    VX_RBX    = z3.BitVec('VX_RBX',64)
    VX_MEM_64 = z3.BitVec('VX_MEM_64',64)
    VL_RBX    = z3.BitVec('VL_RBX',64)
    VL_MEM_64 = z3.BitVec('VL_MEM_64',64)
    V_R       = z3.BitVec('V_R',64)

    s = z3.Solver()
    s.add(VX_RBX == VL_RBX)
    s.add(VL_MEM_64 == VL_MEM_64)

    s.push()
    lvar = (V_R == (VL_MEM_64 & VL_RBX) & 18446744073709551615_64 )
    xvar = (V_R == (VX_RBX & VX_MEM_64))

    s.add(lvar != xvar)
    if(s.check() == z3.sat):
      // report bug
    if(s.check() == z3.unknown):
      // report timeout
    s.pop()
```

Figure A.10: Verification condition corresponding to destination register `%rbx`

REFERENCES

[1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010. *[Cited on page 1.]*

[2] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logic of Programs, Workshop.* Berlin, Heidelberg: Springer-Verlag, 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=648063.747438 pp. 52–71. *[Cited on page 1.]*

[3] J.-P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," in *Proceedings of the 5th Colloquium on International Symposium on Programming.* London, UK, UK: Springer-Verlag, 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=647325.721668 pp. 337–351. *[Cited on page 1.]*

[4] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977. [Online]. Available: http://doi.acm.org/10.1145/512950.512973 pp. 238–252. *[Cited on page 1.]*

[5] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/940071.940115 pp. 327–336. *[Cited on page 1.]*

[6] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855760 pp. 267–280. *[Cited on page 1.]*

[7] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, "F-SOFT: Software Verification Platform," in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: http://dx.doi.org/10.1007/11513988_31 pp. 301–306. *[Cited on page 1.]*

[8] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser, "Formal Software Analysis Emerging Trends in Software Model Checking," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: https://doi.org/10.1109/FOSE.2007.6 pp. 120–136. *[Cited on page 1.]*

[9] D. Binkley, "Source Code Analysis: A Road Map," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: https://doi.org/10.1109/FOSE.2007.27 pp. 104–119. *[Cited on page 1.]*

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374 *[Cited on page 1.]*

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217943 pp. 73–85. *[Cited on page 1.]*

[12] X. Meng and B. P. Miller, "Binary Code is Not Easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2931037.2931047 pp. 24–35. *[Cited on page 1.]*

[13] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275. *[Cited on page 1.]*

[14] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 175–183. *[Cited on page 1.]*

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034 pp. 190–200. *[Cited on page 1.]*

[16] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44–66, 2003, rV '2003, Run-time Verification (Satellite Workshop of CAV '03). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104810429 *[Cited on pages 1, 3, 5, 17, and 53.]*

[17] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Cambridge, MA, USA, 2004, aAI0807735. *[Cited on page 1.]*

[18] C. Cifuentes and M. V. Emmerik, “UQBT: Adaptable Binary Translation at Low Cost,” *Computer*, vol. 33, no. 3, pp. 60–66, Mar. 2000. [Online]. Available: https://doi.org/10.1109/2.825697 *[Cited on page 1.]*

[19] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna, “Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis,” in *Information Security and Assurance*, T.-h. Kim, H. Adeli, R. J. Robles, and M. Balitanas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 72–86. *[Cited on page 1.]*

[20] S. K. Cha, M. Woo, and D. Brumley, “Program-Adaptive Mutational Fuzzing,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: https://doi.org/10.1109/SP.2015.50 pp. 725–741. *[Cited on page 1.]*

[21] B. Ford and R. Cox, “Vx32: Lightweight User-level Sandboxing on the x86,” in *USENIX 2008 Annual Technical Conference*, ser. ATC’08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404039 pp. 293–306. *[Cited on page 1.]*

[22] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC’13. USA: USENIX Association, 2013, pp. 337–352. *[Cited on page 1.]*

[23] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical Control Flow Integrity and Randomization for Binary Executables,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.44 pp. 559–573. *[Cited on page 1.]*

[24] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-vivo Multi-path Analysis of Software Systems,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950396 pp. 265–278. *[Cited on page 1.]*

[25] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing Symbolic Execution with Veritesting,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568293 pp. 1083–1094. *[Cited on page 1.]*

[26] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated Whitebox Fuzz Testing,” in *Proceedings of NDSS (Network and Distributed Systems Security)*, 2008, pp. 151–166. *[Cited on page 1.]*

[27] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401 pp. 41–41. *[Cited on pages 1, 3, 11, 12, and 17.]*

[28] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002. [Online]. Available: https://doi.org/10.1109/2.982916 *[Cited on page 1.]*

[29] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic Reverse Engineering of Input Formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455820 pp. 391–402. *[Cited on page 1.]*

[30] Z. Lin and X. Zhang, "Deriving Input Syntactic Structure from Execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453114 pp. 83–93. *[Cited on page 1.]*

[31] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534797 pp. 353–368. *[Cited on pages 1, 3, and 12.]*

[32] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations," in *NDSS*, 2015. *[Cited on pages 1, 3, and 12.]*

[33] A. Ruef and A. Dinaburg, "Static Translation of X86 Instruction Semantics to LLVM with McSema," 2014. [Online]. Available: https://github.com/trailofbits/mcsema *[Cited on pages 1, 2, 3, 4, 5, 7, 9, 17, 53, and 56.]*

[34] "Angr: A powerful and user-friendly binary analysis platform!" http://angr.io/, July 2018, last accessed: May 1, 2020. *[Cited on pages 1, 3, 7, and 17.]*

[35] S. Alvarez, "Radare2," https://rada.re/r/, July 2018, last accessed: May 1, 2020. *[Cited on pages 1, 2, 3, 4, 6, 7, 17, and 25.]*

[36] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure Execution via Program Shepherding," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=647253.720293 pp. 191–206. *[Cited on page 1.]*

[37] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298463 pp. 75–88. *[Cited on page 1.]*

[38] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: https://doi.org/10.1109/SP.2009.25 pp. 79–93. *[Cited on page 1.]*

[39] L. C. Harris and B. P. Miller, "Practical Analysis of Stripped Binary Code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1127577.1127590 *[Cited on page 1.]*

[40] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994. [Online]. Available: http://doi.acm.org/10.1145/178243.178260 pp. 196–205. *[Cited on page 1.]*

[41] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/SP.2012.31 pp. 380–394. *[Cited on page 1.]*

[42] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-Aware Malware Detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/SP.2005.20 pp. 32–46. *[Cited on page 2.]*

[43] A. Holzer, J. Kinder, and H. Veith, "Using verification technology to specify and detect malware," in *Computer Aided Systems Theory – EUROCAST 2007*, R. Moreno Díaz, F. Pichler, and A. Quesada Arencibia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 497–504. *[Cited on page 2.]*

[44] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting Malicious Code by Model Checking," in *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: http://dx.doi.org/10.1007/11506881_11 pp. 174–187. *[Cited on page 2.]*

[45] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive Detection of Computer Worms Using Model Checking," *IEEE Trans. Dependable Secur. Comput.*, vol. 7, no. 4, pp. 424–438, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2008.74 *[Cited on page 2.]*

[46] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855790 pp. 351–366. *[Cited on page 2.]*

[47] D. Gopan and T. Reps, "Low-level library analysis and summarization," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 68–81. *[Cited on page 2.]*

[48] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1749608.1749612 *[Cited on page 2.]*

[49] "Remill: Library for lifting of x86, amd64, and aarch64 machine code to LLVM bitcode," https://github.com/trailofbits/remill, July 2018, last accessed: May 1, 2020. *[Cited on pages 2, 3, 4, 9, 25, 38, 53, and 56.]*

[50] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," 2016. *[Cited on pages 2, 3, 4, and 6.]*

[51] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032342 pp. 463–469. *[Cited on pages 2, 3, 4, 6, 7, 12, and 17.]*

[52] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs." *The Computer Journal*, vol. 23, pp. 223–229, 1980. *[Cited on page 2.]*

[53] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," in *Proceedings Seventh International Workshop on Program Comprehension*, May 1999, pp. 192–199. *[Cited on pages 2 and 85.]*

[54] "Jump tables," https://www.hexblog.com/?p=68, last accessed: May 1, 2020. *[Cited on pages 2 and 85.]*

[55] "Simplex method in IDA Pro," http://www.hexblog.com/?p=42, last accessed: May 1, 2020. *[Cited on pages 2 and 85.]*

[56] J. Troger and C. Cifuentes, "Analysis of virtual method invocation for binary translation," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, Nov 2002, pp. 65–74. *[Cited on pages 2 and 85.]*

[57] "fcd: An optimizing decompiler," https://zneak.github.io/fcd/, July 2018, last accessed: May 1, 2020. *[Cited on pages 3, 4, 7, 53, 54, 58, and 84.]*

[58] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS '08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89862-7_1 pp. 1–25. *[Cited on pages 3, 4, and 7.]*

[59] "Hex-Rays Decompiler," http://www.hex-rays.com/decompiler.shtml, last accessed: May 1, 2020. *[Cited on page 3.]*

[60] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "SmartDec: Approaching C++ Decompilation," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, ser. WCRE '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: https://doi.org/10.1109/WCRE.2011.49 pp. 347–356. *[Cited on page 3.]*

[61] A. Smith and S. B. Yadavalli, "LLVM Based Binary Raiser: llvm-mctoll," 2018. [Online]. Available: https://github.com/Microsoft/llvm-mctoll *[Cited on pages 3, 7, and 53.]*

[62] V. Chipounov and G. Candea, "Enabling sophisticated analyses of x86 binaries with RevGen," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2011, pp. 211–216. *[Cited on pages 3 and 7.]*

[63] A. Bougacha, "Binary Translator to LLVM IR," https://github.com/repzret/dagger, 2017, last accessed: May 1, 2020. *[Cited on pages 3 and 7.]*

[64] J. Feichtner, D. Missmann, and R. Spreitzer, "Automated Binary Analysis on IOS: A Case Study on Cryptographic Misuse in IOS Applications," ser. WiSec '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3212480.3212487 pp. 236–247. *[Cited on page 3.]*

[65] K. Kirchner and S. Rosenthaler, "Bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3098954.3103152 *[Cited on pages 3 and 7.]*

[66] A. Di Federico, M. Payer, and G. Agosta, "Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3033019.3033028 pp. 131–141. *[Cited on pages 3, 4, 7, 54, 58, and 84.]*

[67] "reopt: A tool for analyzing x86-64 binaries," https://github.com/GaloisInc/reopt, 2014, last accessed: May 1, 2020. *[Cited on pages 3, 7, 13, and 53.]*

[68] J. Křoustek and P. Matula, "RetDec: An Open-Source Machine-Code Decompiler," [talk], July 2018, presented at Pass the SALT 2018, Lille, FR. *[Cited on pages 3 and 7.]*

[69] Draper-Laboratory, "An architecture-independent decompiler to LLVM IR," https://github.com/draperlaboratory/fracture, last accessed: May 1, 2020. *[Cited on pages 3 and 7.]*

[70] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," 2016. *[Cited on page 3.]*

[71] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," 2015. *[Cited on page 3.]*

[72] "Python bindings for Valgrind's VEX IR." https://github.com/angr/pyvex, 2013, last accessed: May 1, 2020. *[Cited on pages 3 and 12.]*

[73] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA Framework for Binary Code Analysis," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032318 pp. 165–170. *[Cited on pages 3 and 12.]*

[74] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," 2009. *[Cited on page 3.]*

[75] E. Fleury, O. Ly, G. Point, and A. Vincent, "Insight: An open binary analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 218–224. *[Cited on page 3.]*

[76] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *Compiler Construction*, R. Bodik, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 250–254. *[Cited on page 3.]*

[77] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011.* The Internet Society, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs *[Cited on page 3.]*

[78] M. Noonan, A. Loginov, and D. Cok, "Polymorphic Type Inference for Machine Code," *SIGPLAN Not.*, vol. 51, no. 6, pp. 27–41, June 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908119 *[Cited on pages 3 and 17.]*

[79] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang, “LLBT: An LLVM-based Static Binary Translator,” in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES ’12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2380403.2380419 pp. 51–60. *[Cited on pages 3 and 12.]*

[80] K. P. Lawton, “Bochs: A Portable PC Emulator for Unix/X,” *Linux J.*, vol. 1996, no. 29es, Sep. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=326350.326357 *[Cited on pages 3, 11, 12, and 43.]*

[81] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, “A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2465351.2465380 pp. 295–308. *[Cited on page 3.]*

[82] S. Bansal and A. Aiken, “Binary Translation Using Peephole Superoptimizers,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USA: USENIX Association, 2008, pp. 177–192. *[Cited on pages 3 and 17.]*

[83] “McSema-Research & Development,” https://www.trailofbits.com/research-and-development/mcsema/, last accessed: May 1, 2020. *[Cited on page 3.]*

[84] “Intel 64 and IA-32 Architectures Software Developer Manuals,” https://software.intel.com/en-us/articles/intel-sdm, 2019, published on October 12, 2016, updated on September 26, 2019. *[Cited on pages 4, 7, 8, 23, 75, and 86.]*

[85] “ARM Architecture Reference Manual,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html, 1996, last accessed: May 1, 2020. *[Cited on pages 4 and 17.]*

[86] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, “A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314601 pp. 1133–1148. *[Cited on pages 4, 13, 17, 23, 53, 56, and 73.]*

[87] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, “Stratified Synthesis: Automatically Learning the x86-64 Instruction Set,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908121 pp. 237–250. *[Cited on pages 4, 6, 7, 9, 14, 16, 17, 23, 24, 27, 35, 37, 49, 50, 52, and 53.]*

[88] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh, "Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14. Austin, TX: FMCAD Inc, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2682923.2682944 pp. 18:91–18:98. *[Cited on pages 4, 6, 14, 15, 17, 24, 37, 49, 50, 52, and 53.]*

[89] S. Goel, W. A. Hunt, and M. Kaufmann, *Engineering a Formal, Executable x86 ISA Simulator for Software Verification.* Cham: Springer International Publishing, 2017, pp. 173–209. [Online]. Available: https://doi.org/10.1007/978-3-319-48628-4_8 *[Cited on pages 4, 6, 17, 20, 29, 51, and 53.]*

[90] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, July 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814 *[Cited on pages 4, 15, 17, 50, and 53.]*

[91] J. Lim and T. Reps, "TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, pp. 4:1–4:59, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2450136.2450139 *[Cited on pages 4, 15, 17, 31, 50, 51, and 53.]*

[92] N. Hasabnis and R. Sekar, "Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872380 pp. 311–324. *[Cited on pages 4, 17, 23, 50, and 53.]*

[93] N. Hasabnis and R. Sekar, "Extracting Instruction Semantics via Symbolic Execution of Code Generators," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950335 pp. 301–313. *[Cited on pages 4, 6, 17, 23, 50, and 53.]*

[94] T. Kasampalis, "Translation Validation for Compilation Verification," PhD thesis, University of Illinois at Urbana Champaign (to be published), 2020. *[Cited on pages 5, 7, 53, 57, 73, 78, 82, and 89.]*

[95] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM Intermediate Representation for Verified Program Transformations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103709 pp. 427–440. *[Cited on pages 5 and 53.]*

[96] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. *[Cited on pages 5, 17, and 53.]*

[97] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU Emulators," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572303 pp. 261–272. *[Cited on pages 6 and 11.]*

[98] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing System Virtual Machines," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831730 pp. 171–182. *[Cited on pages 6 and 11.]*

[99] J.-Y. Chen, W. Yang, B.-Y. Shen, Y.-J. Li, and W.-C. Hsu, "Automatic Validation for Binary Translation," *Comput. Lang. Syst. Struct.*, vol. 43, no. C, pp. 96–115, Oct. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2015.05.002 *[Cited on pages 6, 11, and 12.]*

[100] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 337–348. *[Cited on pages 6, 11, 12, 17, and 43.]*

[101] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing Intermediate Representations for Binary Analysis," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155609 pp. 353–364. *[Cited on pages 6, 9, 12, and 13.]*

[102] J. Hendrix, G. Wei, and S. Winwood, "Towards Verified Binary Raising ," in *Workshop on Instruction Set Architecture Specification (co-located with ITP 2019)*, 2019. *[Cited on pages 6 and 13.]*

[103] M. O. Myreen, M. J. C. Gordon, and K. Slind, "Machine-code Verification for Multiple Architectures: An Application of Decompilation into Logic," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '08. Piscataway, NJ, USA: IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1517424.1517444 pp. 20:1–20:8. *[Cited on pages 6, 13, 17, and 18.]*

[104] M. O. Myreen, M. J. C. Gordon, and K. Slind, "Decompilation into logic — Improved," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2012, pp. 78–81. *[Cited on pages 6, 13, and 18.]*

[105] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach.* Norwell, MA, USA: Kluwer Academic Publishers, 2000. *[Cited on pages 6 and 15.]*

[106] S. Horwitz, J. Prins, and T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988. [Online]. Available: https://doi.org/10.1145/73560.73573 pp. 146–157. *[Cited on pages 8, 69, and 71.]*

[107] "C Language Testsuites: C-torture version 8.1.0," https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html, 2018, last accessed: May 1, 2020. *[Cited on pages 8, 16, 30, and 40.]*

[108] "Bug Reported in Intel Developer Zone: Possible errors in instruction semantics," https://software.intel.com/en-us/forums/intel-isa-extensions/topic/773342, April 2018, last accessed: May 1, 2020. *[Cited on pages 8 and 39.]*

[109] "Bug Reported in Stoke: Semantic bugs," https://github.com/StanfordPL/stoke/issues/983, April 2018, last accessed: May 1, 2020. *[Cited on pages 9 and 42.]*

[110] "Bug Reported in Stoke: Modelling the behavior of flags which may or must take undef values," https://github.com/StanfordPL/stoke/issues/986, May 2018, last accessed: May 1, 2020. *[Cited on pages 9, 42, and 51.]*

[111] S. Dasgupta, "Defining semantics of instructions unsupported in Strata/Stoke," https://github.com/StanfordPL/stoke/pull/996, 2019, last accessed: May 1, 2020. *[Cited on pages 9 and 51.]*

[112] I. Roessle, F. Verbeek, and B. Ravindran, "Formally Verified Big Step Semantics out of x86-64 Binaries," in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2019. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3293880.3294102 pp. 181–195. *[Cited on pages 9, 15, 17, and 50.]*

[113] "Comparison with other machine code to LLVM bitcode lifters," 2018, last accessed: May 1, 2020. [Online]. Available: https://github.com/lifting-bits/mcsema#comparison-with-other-machine-code-to-llvm-bitcode-lifters *[Cited on page 9.]*

[114] S. Dasgupta, "A Scalable Validator for Binary Lifters," https://github.com/sdasgup3/validating-binary-decompilation, 2020, last accessed: May 1, 2020. *[Cited on page 10.]*

[115] S. Dasgupta, "Semantics of x86-64 in K," https://github.com/kframework/X86-64-semantics, 2018, last accessed: May 1, 2020. *[Cited on page 10.]*

[116] B. Shen, J. You, W. Yang, and W. Hsu, "An LLVM-based hybrid binary translation system," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012, pp. 229–236. *[Cited on page 12.]*

[117] A. H. Miranti, S. Dasgupta, and G. Roşu, "Formalizing x86-64 Instruction Decoder in K," in *Workshop on Instruction Set Architecture Specification (co-located with ITP 2019)*, 2019. *[Cited on pages 13 and 23.]*

[118] M. O. Myreen and M. J. C. Gordon, "Hoare logic for realistically modelled machine code," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 568–582. *[Cited on page 13.]*

[119] A. Fox, "Improved Tool Support for Machine-Code Decompilation in HOL4," in *Interactive Theorem Proving*, C. Urban and X. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 187–202. *[Cited on pages 13, 17, and 18.]*

[120] K. Slind and M. Norrish, "A Brief Overview of HOL4," in *Theorem Proving in Higher Order Logics*, O. A. Mohamed, C. Muñoz, and S. Tahar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 28–32. *[Cited on pages 13 and 18.]*

[121] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation Validation for a Verified OS Kernel," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462183 pp. 471–482. *[Cited on pages 13 and 18.]*

[122] "X86isa: Implemented-opcodes: Opcodes Supported by the x86 Model," http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/X86ISA____IMPLEMENTED-OPCODES, July 2018, last accessed: May 1, 2020. *[Cited on page 15.]*

[123] "The Coq Proof Assistant," https://coq.inria.fr/, 2018, last accessed: May 1, 2020. *[Cited on page 15.]*

[124] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026928 pp. 653–669. *[Cited on page 15.]*

[125] V. Srinivasan and T. Reps, "Synthesis of machine code from semantics," *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pp. 596–607, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2737924.2737960 *[Cited on page 15.]*

[126] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: X86-TSO," in *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03359-9_27 pp. 391–407. *[Cited on pages 15, 17, 24, and 49.]*

[127] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The Semantics of x86-CC Multiprocessor Machine Code," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480929 pp. 379–391. *[Cited on pages 15, 17, 24, and 49.]*

[128] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290384 *[Cited on pages 16, 17, 18, 37, and 53.]*

[129] "Sail x86 ISA model," https://github.com/rems-project/sail/tree/sail2/x86, 2017, last accessed: May 1, 2020. *[Cited on pages 16, 18, and 50.]*

[130] "rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86," https://github.com/rems-project/rmem/, 2019, last accessed: May 1, 2020. *[Cited on pages 16 and 18.]*

[131] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 19:1–19:29, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3158107 *[Cited on pages 16 and 18.]*

[132] A. Reid, "Trustworthy Specifications of ARM V8-A and v8-M System Level Architecture," in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '16. Austin, Texas: FMCAD Inc, 2016, pp. 161–168. *[Cited on pages 16, 17, 18, 19, 20, 37, and 43.]*

[133] A. Reid, "ARM's Architecture Specification Language," https://alastairreid.github.io/specification_languages/, 2016, last accessed: May 1, 2020. *[Cited on pages 16 and 18.]*

[134] U. Degenbaev, "Formal specification of the x86 instruction set architecture," 03 2012. [Online]. Available: https://core.ac.uk/download/pdf/10697957.pdf *[Cited on page 16.]*

[135] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451150 pp. 305–316. *[Cited on pages 16, 24, 27, 41, and 51.]*

[136] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, “RockSalt: better, faster, stronger SFI for the x86,” *PLDI: Programming Languages Design and Implementation*, pp. 395–404, 2012. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111{%}5Cnhttp://dl.acm.org/ft{_}gateway.cfm?id=2254111{&}type=pdf *[Cited on page 17.]*

[137] “Evaluable Strings Intermediate Language,” https://radare.gitbooks.io/radare2book/content/disassembling/esil.html, July 2018, last accessed: May 1, 2020. *[Cited on page 17.]*

[138] W. A. H. Jr., “Microprocessor Design Verification,” *J. Autom. Reasoning*, vol. 5, no. 4, pp. 429–460, 1989. [Online]. Available: https://doi.org/10.1007/BF00243132 *[Cited on pages 17 and 43.]*

[139] S. Goel, A. Slobodova, R. Sumners, and S. Swords, “Verifying X86 Instruction Implementations,” in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372885.3373811 pp. 47–60. *[Cited on pages 17 and 43.]*

[140] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, “End-to-End Verification of Processors with ISA-Formal,” in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 42–58. *[Cited on pages 17, 19, 20, and 43.]*

[141] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “CakeML: A Verified Implementation of ML,” *SIGPLAN Not.*, vol. 49, no. 1, pp. 179–191, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2578855.2535841 *[Cited on pages 17 and 18.]*

[142] X. Leroy, “Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111042 pp. 42–54. *[Cited on page 17.]*

[143] M. O. Myreen and M. J. C. Gordon, “Verified LISP Implementations on ARM,Âăx86ÂăandÂăPowerPC,” in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 359–374. *[Cited on pages 17 and 18.]*

[144] S. Maus, M. Moskal, and W. Schulte, “Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving,” in *Algebraic Methodology and Software Technology*, J. Meseguer and G. Roşu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 284–298. *[Cited on page 17.]*

[145] A. Mycroft, “Type-Based Decompilation (or Program Reconstruction via Type Reconstruction),” in *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ser. ESOP ’99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 208–223. *[Cited on page 17.]*

[146] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, “Scalable Validation of Binary Lifters,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. ACM, 2020. [Online]. Available: http://doi.acm.org/10.1145/3385412.3385964 *[Cited on pages 17 and 53.]*

[147] J. Regehr and U. Duongsaa, “Deriving Abstract Transfer Functions for Analyzing Embedded Software,” *SIGPLAN Not.*, vol. 41, no. 7, pp. 34–43, June 2006. [Online]. Available: https://doi.org/10.1145/1159974.1134657 *[Cited on page 17.]*

[148] J. Regehr and A. Reid, “HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems,” pp. 133–143, 2004. [Online]. Available: https://doi.org/10.1145/1024393.1024410 *[Cited on page 17.]*

[149] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *Queue*, vol. 10, no. 1, pp. 20–27, Jan. 2012. [Online]. Available: https://doi.org/10.1145/2090147.2094081 *[Cited on page 17.]*

[150] A. C. J. Fox and A. Fox, “A HOL specification of the ARM instruction set architecture,” 2001. *[Cited on pages 17 and 18.]*

[151] A. Fox and M. O. Myreen, “A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture,” in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–258. *[Cited on pages 17, 18, and 37.]*

[152] A. Fox, “Directions in ISA Specification,” in *Interactive Theorem Proving*, L. Beringer and A. Felty, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 338–344. *[Cited on page 17.]*

[153] A. Reid, “Who Guards the Guards? Formal Validation of the Arm v8-m Architecture Specification,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133912 *[Cited on pages 17, 19, 20, and 43.]*

[154] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2837614.2837615 pp. 608–621. *[Cited on pages 17, 18, and 37.]*

[155] A. Fox and C. Fd, "Formal verification of the ARM6 micro-architecture," *Technical Report, UCAM-CL-TR-548, University of Cambridge, Computer Laboratory*, 12 2002. *[Cited on pages 18 and 43.]*

[156] A. Fox, "Formal Specification and Verification of ARM6," in *Theorem Proving in Higher Order Logics*, D. Basin and B. Wolff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 25–40. *[Cited on pages 18 and 53.]*

[157] M. Dam, R. Guanciale, and H. Nemati, "Machine Code Verification of a Tiny ARM Hypervisor," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, ser. TrustED '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2517300.2517302 pp. 3–12. *[Cited on page 18.]*

[158] C. Ellison and G. Roşu, "An Executable Formal Semantics of C with Applications," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12).* ACM, January 2012, pp. 533–544. *[Cited on pages 18 and 31.]*

[159] D. Filaretti and S. Maffeis, "An Executable Formal Semantics of PHP," in *ECOOP 2014 – Object-Oriented Programming*, R. Jones, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. *[Cited on page 18.]*

[160] D. Guth, "A formal semantics of Python 3.3," 2013. *[Cited on page 18.]*

[161] D. Bogdănaş and G. Roşu, "K-Java: A Complete Semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15).* ACM, January 2015, pp. 445–456. *[Cited on page 18.]*

[162] D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).* ACM, June 2015, pp. 346–356. *[Cited on page 18.]*

[163] "The EVM Jello Paper," https://anon293409324.github.io/, 2017, last accessed: May 1, 2020. *[Cited on pages 18 and 43.]*

[164] A. Reid, "Defining interfaces between hardware and software: Quality and performance," Ph.D. dissertation, School of Computing Science, University of Glasgow, March 2019. *[Cited on pages 19 and 43.]*

[165] "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (DDI0487))," https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile, 2013, last accessed: May 1, 2020. *[Cited on page 19.]*

[166] "Armv8-M Architecture Reference Manual (DDI0553))," https://developer.arm.com/docs/ddi0553/latest/armv8-m-architecture-reference-manual, 2016, last accessed: May 1, 2020. *[Cited on page 19.]*

[167] A. Pnueli, M. Siegel, and E. Singerman, "Translation Validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. Berlin, Heidelberg: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646482.691453 pp. 151–166. *[Cited on page 20.]*

[168] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg, "VOC: A Translation Validator for Optimizing Compilers," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 2, pp. 2–18, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104803931 *[Cited on page 20.]*

[169] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck, "TVOC: A Translation Validator for Optimizing Compilers," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 291–295. *[Cited on page 20.]*

[170] G. C. Necula, "Translation Validation for an Optimizing Compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/349299.349314 pp. 83–94. *[Cited on pages 20 and 21.]*

[171] J.-B. Tristan, P. Govereau, and G. Morrisett, "Evaluating Value-graph Translation Validation for LLVM," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993533 pp. 295–305. *[Cited on pages 20, 22, 72, and 82.]*

[172] M. Stepp, R. Tate, and S. Lerner, "Equality-based Translation Validator for LLVM," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032364 pp. 737–742. *[Cited on pages 20, 22, 72, and 82.]*

[173] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu, "Translation and Run-Time Validation of Loop Transformations," *Form. Methods Syst. Des.*, vol. 27, no. 3, pp. 335–360, Nov. 2005. [Online]. Available: https://doi.org/10.1007/s10703-005-3402-z *[Cited on page 20.]*

[174] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth, "Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491442 pp. 191–201. *[Cited on page 21.]*

[175] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A Modular Reusable Verifier for Object-oriented Programs,” in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, ser. FMCO’05. Berlin, Heidelberg: Springer-Verlag, 2006. [Online]. Available: http://dx.doi.org/10.1007/11804192_17 pp. 364–387. *[Cited on page 21.]*

[176] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, “SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs,” in *Proceedings of the 24th International Conference on Computer Aided Verification*, ser. CAV’12. Berlin, Heidelberg: Springer-Verlag, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_54 pp. 712–717. *[Cited on page 21.]*

[177] X. Rival, “Symbolic Transfer Function-based Approaches to Certified Compilation,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/964001.964002 pp. 1–13. *[Cited on page 21.]*

[178] T. F. Şerbănuţa, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, “The K primer (version 3.2),” Tech. Rep. *[Cited on pages 23, 26, and 57.]*

[179] “x86 and amd64 Instruction Reference (UnOfficial),” http://www.felixcloutier.com/x86/, July 2018, last accessed: May 1, 2020. *[Cited on page 24.]*

[180] “x86 assembly language,” https://en.wikipedia.org/wiki/X86_assembly_language/, July 2018, last accessed: May 1, 2020. *[Cited on page 26.]*

[181] G. Roşu and T. F. Şerbănuţă, “An Overview of the K Semantic Framework,” *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010. *[Cited on page 26.]*

[182] G. L. Steele, Jr., *Common LISP: The Language.* Newton, MA, USA: Digital Press, 1984. *[Cited on page 35.]*

[183] X. Shi, “Certification of an Instruction Set Simulator,” Theses, Université Grenoble Alpes, July 2013. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00937524 *[Cited on page 37.]*

[184] “GDB: The GNU Project Debugger,” https://www.gnu.org/software/gdb/, July 2018, last accessed: May 1, 2020. *[Cited on page 37.]*

[185] “IEEE Std 754-2008 - IEEE Standard for Floating-Point Arithmetic,” https://standards.ieee.org/findstds/standard/754-2008.html, Aug 2008. *[Cited on pages 38 and 39.]*

[186] S. Dasgupta, “Test Inputs borrowed from Strata project,” https://github.com/kframework/X86-64-semantics/tree/master/tests/single-instruction-tests/test-inputs, 2018, last accessed: May 1, 2020. *[Cited on page 38.]*

[187] S. Dasgupta, "Test Inputs (either manually generated or borrowed from related projects," https://github.com/kframework/X86-64-semantics/tree/master/tests/single-instruction-tests, 2018, last accessed: May 1, 2020. *[Cited on page 38.]*

[188] "Eric Schkufza. Personal communication," June 2018. *[Cited on page 40.]*

[189] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A Formal Verification Tool for Ethereum VM Bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3236024.3264591 pp. 912–915. *[Cited on page 42.]*

[190] H. D. Foster, "Trends in Functional Verification: A 2014 Industry Study," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744921 pp. 48:1–48:6. *[Cited on page 42.]*

[191] L. Liu and S. Vasudevan, "Scaling Input Stimulus Generation Through Hybrid Static and Dynamic Analysis of RTL," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 1, pp. 4:1–4:33, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2676549 *[Cited on page 43.]*

[192] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6. *[Cited on page 43.]*

[193] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1538–1543. *[Cited on page 43.]*

[194] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Putting It All Together – Formal Verification of the VAMP," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 411–430, Aug. 2006. *[Cited on page 43.]*

[195] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An Empirical Study on the Correctness of Formally Verified Distributed Systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 328–343. *[Cited on page 43.]*

[196] D. A. Burke and K. Johannisson, "Translating Formal Software Specifications to Natural Language," in *Logical Aspects of Computational Linguistics*, P. Blache, E. Stabler, J. Busquets, and R. Moot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 51–66. *[Cited on page 43.]*

[197] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based Program Verifiers for All Languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984027 pp. 74–91. *[Cited on page 44.]*

[198] G. Roşu and A. Ştefănescu, "Checking Reachability using Matching Logic," in *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, Oct 2012, pp. 555–574. *[Cited on page 44.]*

[199] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267327 pp. 19–19. *[Cited on page 46.]*

[200] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756 pp. 209–224. *[Cited on pages 46 and 48.]*

[201] "MPFR Java Bindings," https://github.com/kframework/mpfr-java, July 2018, last accessed: May 1, 2020. *[Cited on page 49.]*

[202] "The GNU MPFR Library," https://www.mpfr.org/, July 2018, last accessed: May 1, 2020. *[Cited on page 49.]*

[203] S. Dasgupta, "Improving Stoke ability to debug a circuit," https://github.com/StanfordPL/stoke/pull/997, 2019, last accessed: May 1, 2020. *[Cited on page 51.]*

[204] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, "Verified Compilation on a Verified Processor," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314622 pp. 1041–1053. *[Cited on page 53.]*

[205] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596 pp. 207–220. *[Cited on page 53.]*

[206] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16).* Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse pp. 583–600. *[Cited on page 54.]*

[207] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766 pp. 337–340. *[Cited on page 55.]*

[208] D. Sangiorgi, *Introduction to Bisimulation and Coinduction.* New York, NY, USA: Cambridge University Press, 2011. *[Cited on page 60.]*

[209] "MemorySSA," https://llvm.org/docs/MemorySSA.html, last accessed: May 1, 2020. *[Cited on page 68.]*

[210] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "DualIso: An Algorithm for Subgraph Pattern Matching on Very Large Labeled Graphs," in *2014 IEEE International Congress on Big Data*, June 2014, pp. 498–505. *[Cited on page 68.]*

[211] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. *[Cited on page 71.]*

[212] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf *[Cited on pages 72 and 83.]*

[213] "Z3's behavior seems to depend on the order in which formulas are asserted," https://github.com/Z3Prover/z3/issues/1106, June 2017, last accessed: May 1, 2020. *[Cited on page 74.]*

[214] "A few discrepancies in x86-64 Instruction Semantics," https://github.com/lifting-bits/remill/issues/376, November 2019, last accessed: May 1, 2020. *[Cited on page 74.]*

[215] I. Guilfanov, "Decompilers and Beyond," In Black-Hat USA, July 2008. *[Cited on page 78.]*

[216] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, “N-version Disassembly: Differential Testing of x86 Disassemblers,” *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA ’10*, p. 265, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1831708.1831741 *[Cited on page 84.]*