Formally Verified Big Step Semantics out of x86-64 Binaries

Ian Roessle
Virginia Tech, USA
iroessle@vt.edu

Freek Verbeek
Virginia Tech, USA
freek@vt.edu

Binoy Ravindran
Virginia Tech, USA
binoy@vt.edu

Abstract

This paper presents a methodology for generating formally proven equivalence theorems between decompiled x86-64 machine code and big step semantics. These proofs are built on top of two additional contributions. First, a robust and tested formal x86-64 machine model containing small step semantics for 1625 instructions. Second, a decompilation-into-logic methodology supporting both x86-64 assembly and machine code at large scale. This work enables black-box binary verification, i.e., formal verification of a binary where source code is unavailable. As such, it can be applied to safety-critical systems that consist of legacy components, or components whose source code is unavailable due to proprietary reasons. The methodology minimizes the trusted code base by leveraging machine-learned semantics to build a formal machine model. We apply the methodology to several case studies, including binaries that heavily rely on the SSE2 floating-point instruction set, and binaries that are obtained by compiling code that is obtained by inlining assembly into C code.

CCS Concepts • **Theory of computation → Equational logic and rewriting**; **Abstraction**;

Keywords x86-64, semantics, theorem proving

ACM Reference Format:
Ian Roessle, Freek Verbeek, and Binoy Ravindran. 2019. Formally Verified Big Step Semantics out of x86-64 Binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3293880.3294102

1 Introduction

This paper targets bottom-up formal verification, i.e., verification of binaries where source code is unavailable. Our aim is to use formal methods to analyze legacy systems, or systems where source code is unavailable due to proprietary reasons. Various safety-critical systems in automotive, aerospace, medical and military domains are built out of components where the source code is not available [38]. In such a context, certification plays a crucial role. Certification can require a compliance proof based on formal methods [32, 41]. Bottom-up formal verification can aid in obtaining high levels of assurance for black-box components running on commodity hardware, such as x86-64.

A binary typically consists of *blocks* composed by *control flow*. In this paper, a block is defined as a sequence of instructions that can be modeled with only if-then-else statements (no loops). A formal model of a binary can be obtained by translating, e.g., loops to recursive functions, and blocks to sequences of state updates. Each state update corresponds to the semantics of one instruction, dictated by a *machine model*. We call this the *small-step semantics* of that block. This approach is called *decompilation-into-logic* (DiL) [29, 30]. The model obtained by DiL can, e.g., then be used to prove correspondence to source code.

In a context where source code is unavailable, however, small-step semantics do not suffice. A block can consist of dozens of lines of machine code that is unintelligible and not suitable for further analysis. This paper presents a methodology that largely automatically derives a formal model for a block in a binary that is on the same level of abstraction as C. We call this the *big-step semantics* of that block. This provides insight into the semantics of the binary and enables use of the generated formal model for correctness proofs. Moreover, it provides a user with insight into the branching conditions in the binary, which can aid in building test suites.

Of particular note within any formal verification effort is the trusted code base (TCB) [21]. Specifically, the machine model is part of the TCB. The x86-64 architecture presents a unique challenge, since hand-writing a machine model of the x86-64 architecture is inherently based on semi-formal Intel manuals. Therefore, Heule et al. applied machine learning to *infer* semantics from live x86-64 hardware [17]. Their approach produced semantics that are more reliable than the Intel manuals. We provide a DiL framework that maps instructions in a binary to machine learned semantics. This has an additional advantage that it learns a set of test cases that cover intricate corner cases. We automatically prove millions of lemmas using these test cases, which validate our machine model against live x86-64 hardware.

CPP '19, January 14–15, 2019, Cascais, Portugal
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6222-1/19/01…$15.00
https://doi.org/10.1145/3293880.3294102

This paper presents the following contributions: 1.) a largely automated way to generate big-step semantics of blocks in a binary, plus a formal proof of equivalence between big- and small-step semantics, based on 2.) a machine learned and formally tested x86-64 machine model containing 1625 instructions variants (IVs), and 3.) a DiL implementation for x86-64 and the Isabelle/HOL theorem prover [9, 31]. The latter applies to both Objdump disassembled machine code and symbolized assembly, making it possible to leverage recent advances in *reassembly* [40].

The challenge of binary verification is the semantical gap between a source language and a binary. This introduces some limitations. Specifically, we are not able to extract typing information. Local variables and values in memory have a known size, but their type is unknown. Our machine model does not deal with concurrency. We cannot deal with self-modifying code. To support calls to library functions and indirect calls, a more advanced memory model is needed, allowing assumptions on where loaded libraries are stored. Finally, we have targeted x86-64 specifically, instead of making the methodology generic.

The methodology is applied to several examples to show that it is able to deal with, e.g., if-then-else structures, floating-point operations and pointers. We verify an example where the binary has been obtained by compiling C code mixed with inline assembly. We also verify a binary containing the remainder function from the FDLIBM floating point library[1]. For each case study, we show that the big-step semantics lifted out of the binary is a close match to the original source code. All case studies and the Isabelle/HOL proofs are publicly available at: https://filebox.ece.vt.edu/~iroessle/cpp_2019.zip

2 Methodology

The first step is disassembly (see Figure 1). *Reassembly* builds on disassembly by also performing *symbolization*, where memory references are translated from concrete addresses to labels. Various reassembly tools exist, e.g., IDA Pro [11], Ramblr [40], and Codesurfer [3]. Symbolization supports formal verification that is agnostic of memory layout. We use Ramblr. Ramblr provides sufficient symbolization, while also ensuring recompilability.

We perform a *deep* embedding of the reassembled binary into Isabelle/HOL. A deep embedding is a simple syntactic translation requiring only a parser. This reduces the TCB, as it prevents semantical errors in the translation. The result is a *binary model*: a populated data structure in Isabelle/HOL, which contains the text-, data- and bss-sections of the binary. Section 3.3 provides more details.

To build a machine model, we leverage Strata [17]. Strata provides trustworthy semantics of x86-64 instructions that have been obtained by machine learning. Strata demonstrates trustworthy semantics for 692 instructions, which through generalization arguments expands to 1625 IVs. An additional 119 instructions with 8-bit immediate operands are supported by providing 256 formulas per instruction (one formula per immediate value).

We *extract* generalized semantics out of Strata. The semantics in Strata are stored either as assembly code fragments that derive off a base set of instructions, or as functions within C++. Strata has an application that can translate the aforementioned into bit-vector formulas (bvf's) for a specific instruction. However, there was no support for memory operands and the output was specific to the supplied set of operands. We developed a tool in the Strata C++ namespace that implements the generalization arguments for memory and immediate arguments, and outputs formulas for 1625 IVs in a form generic with respect to operands (see Section 4). For instructions unsupported by Strata (e.g., jumps), we define hand-written semantics.

We developed a formal language called *Chum* that can be used to express x86-64 instruction semantics. Chum serves as an intermediary between Strata and Isabelle/HOL. It combines standard bvf's (such as in QF_BV in SMT-LIB [6]) with our machine model. It thus contains operators to update the machine state, such as memory read/writes, and register- and flag assignments. We extract Chum from Strata. The result – a file containing instruction semantics written in Chum – is then deeply embedded into Isabelle/HOL. Essentially, this methodology reduces the problem of giving semantics to the full x86-64 instruction set to giving semantics to a small bit-vector (bv) language. To minimize the TCB, a testing framework is set up to test the formal instruction semantics on an actual x86-64 machine (see Section 6).

The result of these steps is a trustworthy syntactical representation of a binary in Isabelle/HOL, with trustworthy semantics for each individual instruction. The next step is to derive big-step semantics of blocks within the binary (see Section 5). We restrict the semantics of Chum to the executable subset of the logic of Isabelle/HOL wherever possible. Moreover, we provide a library of rewrite rules proven correct within Isabelle/HOL. This allows formal symbolic execution, which automatically rewrites the per-instruction small-step semantics to big-step block semantics.

3 Overview of Formal Model

3.1 Machine Model

The machine model $\mathcal{M}$ consists of a state automaton with instructions as labels. The set of states S is a defined using a record. Let R_n denote the set of n-bit registers and F denote the set of flags. We use the Isabelle datatype $'\alpha$ *word* [9] to define bv's of length $'\alpha$.

$$S \equiv < regs :: R_n \mapsto n\ word, mem :: 64\ word \mapsto 8\ word, \\ flags :: F \mapsto \mathbb{B} > \quad \text{with } n \in \{64, 256\}$$

[1] http://www.netlib.org/fdlibm/

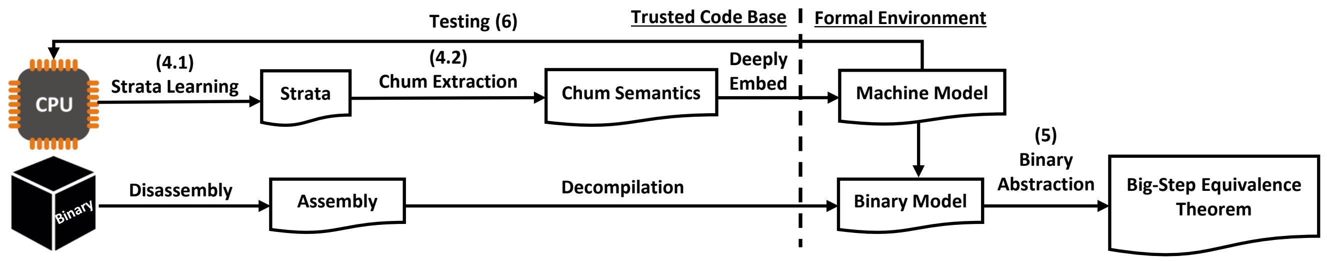

Figure 1. Methodology to lift abstract specifications out of x86-64 binaries

The state stores the contents of the registers, provides a 64-bit address space of bytes, and stores the flags. To access a part of the state σ, read and write functions r and w are defined. For example, $r(\texttt{rip}, \sigma)$ reads the instruction pointer register and $w(a, 255, \sigma)$ writes at address a the byte 255 into memory. Note that there are no registers smaller than 64 bits, nor are there 128-bit registers. For example, the 32-bit register `eax` is a part of the 64-bit `rax` register. The semantics of instructions concerning `eax` are thus expressed in terms of operations on the 64-bit `rax` register. For example, writing the 32-bit register `eax` will additionally zero out the upper 32 bits of `rax`, whereas writing to the 16-bit `ax` will leave the upper 48 bits of the lower part untouched. Similar semantics exist for 128-bit registers, which are actually part of the 256-bit ones. Finally, we introduce functions r_{mem} and w_{mem} to read and write blocks of bytes to the memory at once. For example, $w_{mem}(v, a, s, \sigma)$ writes – in little-endian fashion – value v into the memory at address a, split into a list of s bytes. Depending on the size, the value is possibly truncated or zero-extended.

The machine model is a step function over these states labeled with assembly instructions. Let I denote the set of instructions:

$$\mathcal{M} :: I \times S \mapsto S$$

An instruction is determined by its IV and its *operands*. For example:

Instruction:	`add rax, rbx`
IV:	`add r64, r64`
Operands:	`[rax,rbx]`
Instruction:	`cmp dword ptr [rbp - 0x14], 0x7F`
IV:	`cmp m32, imm32`
Operands:	`[dword ptr [rbp - 0x14],0x7F]`

3.2 Chum: Instruction Semantics

Central to the machine model is a function *getChum* :: $V \mapsto$ *Chum*, where V is the set of IVs. The semantics of an instruction are fully determined by its IV. They are expressed by a datatype *Chum* assigning bvf's to registers and flags (see Figure 2). The sequence of bvf's of a certain IV are executed *independently*, i.e., all assignments occur simultaneously on the state.

$$\begin{array}{lll}
\texttt{chum} & \longrightarrow & \texttt{assignee} := \texttt{bvf} \mid \texttt{semantic}; \texttt{semantic} \\
\texttt{assignee} & \longrightarrow & \texttt{reg} \mid \texttt{mem} \mid \texttt{flg} \mid \texttt{var} \\
\texttt{mem} & \longrightarrow & (\texttt{loc}, \mathbb{N}) \\
\texttt{loc} & \longrightarrow & \texttt{loc} \,\square_a\, \texttt{loc} \mid 64\ \mathit{word} \mid [\texttt{reg}] \mid \texttt{label} \\
\texttt{bvf} & \longrightarrow & \texttt{bvf} \,\square_b\, \texttt{bvf} \mid \square_u(\texttt{bvf}) \mid \texttt{bvf}\ !!\ \mathbb{N} \mid \texttt{val} \\
 & & \mid \langle \mathbb{N}, \mathbb{N} \rangle \texttt{bvf} \mid \underset{\mathbb{N}}{\underline{\texttt{bvf}}} \mid \texttt{if}\, \texttt{Bbvf}\ \texttt{then}\ \texttt{bvf}\ \texttt{else}\ \texttt{bvf} \\
\texttt{Bbvf} & \longrightarrow & \texttt{bvf} \,\square_B\, \texttt{bvf} \\
\texttt{val} & \longrightarrow & \texttt{var} \mid \texttt{closed} \\
\texttt{var} & \longrightarrow & \mathit{OP1} \mid \mathit{OP2} \mid \mathit{OP3} \\
\texttt{closed} & \longrightarrow & r(\texttt{reg}, \sigma) \mid r_{mem}(\texttt{mem}, \sigma) \mid \texttt{imm}
\end{array}$$

where

$$\square_a \in \{+, -, *, :\}, \square_b \in \{+, -, \wedge, \vee, \smile, <<, +^f, -^f, \ldots\}$$
$$\square_u \in \{\text{zxtend}, \text{sxtend}, \neg, \text{parity}, |\_|^f\}, \square_B \in \{=, \neq, \geq, \leq, \ldots\}$$

Figure 2. Chum grammar

Example 3.1. The semantics of the instruction `add rax, rbx` are the same, regardless of which 64-bit registers are used. Function *getChum* returns:

$$\mathit{getChum}(\texttt{add r64, r64}) = (\mathit{OP1} := \mathit{OP1}{+}\mathit{OP2}; \mathit{ZF} := \ldots; \ldots)$$

The instruction writes the sum (a function over bv's) of the operands to its first operand, and sets various flags.

At the top-level, Chum expresses semantics by assigning bvf's to parts of the state: registers, memory, or flags. The assignee can also be left open. A memory location is expressed by an address and a size. The x86-64 instruction set allows address computation within an instruction: addresses can be computed using immediate values, values stored in registers, or labels. A bvf consists of standard bv operations such as logical and arithmetic operators, concatenation ($\smile$), shifting, etc. The notation $\langle h, l \rangle$ denotes bit slicing: it takes the part of the bv starting at bit l (from right to left) and ending at bit h. The notation $\underset{n}{\underline{b}}$ is used to denote that a bvf b is in n-bit mode. The parity function is used to express whether the number of set bits in the given bvf is odd. Floating point operations are indicated by ^{f}, e.g., $|a|^f$ denotes the floating point absolute function. The unary operator !! expresses the nth bit of the given bvf, starting at the right.

This grammar is the basis for several artifacts: first, Chum extraction code writes a plain-text file containing a list of pairs of IVs and Chum semantics, based on this grammar. Second, the grammar is defined as a Chum datatype in ML. Third, the grammar is mechanized as a grammar file for the MLTON compiler. MLTON then generates a parser that reads in the plain-text file, and produces a populated Chum datastructure in ML. Effectively, this deeply embeds the semantics extracted from Strata into ML. Fourth, the grammar is defined as a Chum datatype in Isabelle/HOL. The Chum datastructure in ML is then deeply embedded into the Isabelle/HOL datastructure. The list of pairs is embedded as a map, producing the function *getChum*.

3.2.1 Floating Point Operations

To the best of our knowledge, there is no word-level floating point library in Isabelle/HOL. Operations are needed such as: $+^f \;::\; 64\ \texttt{word} \times 64\ \texttt{word} \mapsto 64\ \texttt{word}$. Defining a library for these functions is outside of the scope of this paper. These operations are introduced as *constrained functions*. In Isabelle, a *locale* is added [4], which is a method for providing a context where some functions are introduced without a function body. One can then formulate constraints over these functions. To ensure that these constraints are not internally inconsistent, a witness is provided.

The floating point locale defines functions $+^f, -^f, *^f, \div^f$ over 64-bit words, representing double precision floating point operations. Besides these, the constants $0^+, 0^-, \infty^+, \infty^-$ and functions sign, isNaN, and $|\_|^f$ are introduced regularly, i.e., with a concrete meaning. Respectively, they return the sign bit (bit 63), check whether the exponent consists solely of 1's and the mantissa is non-zero, and compute the absolute value by setting the sign bit to 0. The constraints are based on the IEEE 754-2008 standard [1]. Examples are:

$$\begin{array}{rcl} x +^f 0^+ & \equiv & x \\ x *^f 0^+ & \equiv & \text{if sign}(x) \text{ then } 0^+ \text{ else } 0^- \\ x \notin \{0^+, 0^-\} \Longrightarrow x \div^f 0^+ & \equiv & \text{if sign}(x) \text{ then } \infty^- \text{ else } \infty^+ \\ \{x, y\} \subseteq \{0^+, 0^-\} & \Longrightarrow & \text{isNaN}(x \div^f y) \end{array}$$

The effect of using this locale instead of concretely defined functions, is that they are not executable. For the constrained functions, we can symbolically execute and simplify floating point formulas only based on the rules introduced by the locale. We will show in Section 5 that we can derive floating point formulas out of a binary. However, floating point constants are simply bv's represented by hexadecimal numbers.

3.3 Decompilation-Into-Logic

The binary model $\mathcal{B}$ consists of the instructions to be executed, an initial state, and a termination condition. Let A denote the address space, i.e, $A = 64$ *word*.

$$\mathcal{B} \equiv <\ \mathit{fetch} :: A \mapsto I, \sigma_0 :: S, \mathit{is\_final} :: A \mapsto \mathbb{B}\ >$$

Function *fetch* provides the current instruction to be executed, based on the current `rip`. The initial state is obtained by loading all data- and bss sections of the binary into memory. The termination condition decides when the binary is finished based on an address.

In order to build the binary model in Isabelle/HOL, a datatype is built that can store a deep embedding of the binary. The datatype consists of datatypes for instructions, registers, flags, address computations, labels, and immediates. Using the MLTON parser generator, a parser for x86-64 `.s` assembly files is built. Via ML, this parser then generates a populated data structure within Isabelle/HOL. We have added a command to Isabelle/HOL called `x86_64_parser` which takes as input the name of an assembly file and loads it into the theorem prover.

We source the `.s` assembly files from an x86_64 binary using the Ramblr disassembler. Ramblr is modified to automatically add annotations in comment fields, which are deeply embedded along with the assembly. The instruction size is added in bytes. This is used to increment `rip` and to compute relative offsets for branching instructions. The opcode is added as well. An opcode is what tells the hardware exactly what operation to perform. Within the Intel instruction set architecture (ISA) there are cases where the hardware has multiple opcodes which support the same instruction, often for optimization purposes. For example instruction `ror ax, 1` (Rotate Right) can be supported by two different opcodes (0xD1 and 0xC1) as there is an optimized version of the ror instruction for shifts of value 1. Instruction `ror ax, 2` has support by only one opcode (0xC1). While it is generally assumed that multiple opcodes supporting the same instruction behave identically, this additional information is necessary for a full deep embedding of the binary within Isabelle/HOL.

What it means to run a binary on top of a machine can now be defined:

$$\begin{array}{rcl} \mathit{start} & = & \mathit{run}(\mathcal{B}.\sigma_0) \\ \mathit{run}(\sigma) & = & \text{let } \mathit{rip} = r(\texttt{rip}, \sigma) \text{ in} \\ & & \quad \text{if } \mathcal{B}.\mathit{is\_final}(\mathit{rip}) \text{ then } \sigma \\ & & \quad \text{else } \mathit{run}(\mathcal{M}(\mathcal{B}.\mathit{fetch}(\mathit{rip}), \sigma)) \end{array}$$

4 Extraction of Chum from Strata

4.1 Strata and Stoke Introduction

We leverage the semantics machine learned by Strata [17] from x86-64 hardware. The search space in which the instructions are learned is a subset of the assembly language (strataAL) consisting of sequential permutations of 51 assumed correct hard-coded register-only instructions and 11 pseudo-instructions (i.e., the *base set*). Strata uses a compiler optimization tool called Stoke [34] to learn IVs as strataAL fragments. Stoke requires as input a set of test cases (input/output pairs). Initially it uses random test cases, with some additional special test cases to cover common corner cases or register values. Strata uses Stoke to learn multiple strataAL fragments for a given instruction. Based on the stochastic nature of the search, each of these learned strataAL fragments can produce different results. Strata then uses the Z3 SMT solver [10] to prove equivalence between these learned strataAL fragments. If the SMT solver proves non-equivalence, a counterexample is then fed back into the test cases, and the process repeats. Ultimately the test cases we use to validate our machine model are concolic versions of these test cases Strata learned as part of this counterexample guided refinement, along with the initial random test cases, and heuristically interesting cases.

In total, Strata learns 692 instructions (one per each register-only IV). Specifically excluded from their learning were MMX, cryptography, x87, `loop`, string instructions (including the `rep` prefix), and any post-Haswell ISA instructions. Included are SSE (2 – 4.1), AVX, AVX2, FMA3, BM1, BM2, as well as legacy x86 instruction sets.

Strata's x86-64 semantics are stored in two different forms. The learned semantics are stored as non-looping strataAL fragments. The initial base set and pseudo instructions are stored as manually written semantics composed as C++ code. Stoke has an application called `stoke_debug_circuit` that can produce a bvf for a learned instruction given its associated strataAL fragment. All formulas produced operate on either whole 64-bit or 256-bit registers. For example, the bvf for the 32-bit `add eax, ebx` will provide a formula that writes to the 64-bit `rax` register.

It is not possible to directly leverage either the learned semantics or the `stoke_debug_circuit` application, to extract all the semantics from Strata. This is because:

1. The semantics are not in a form that can be directly parsed and leveraged into logic. As mentioned, they are stored as non-looping register-only assembly code fragments (strataAL), or C++ code.
2. The `stoke_debug_circuit` application lacks support for production of bvf's that write to memory. For example, it is unable to produce any bvf's for the following IV: `sal m64, imm8`.
3. The `stoke_debug_circuit` application produces formulas for *instructions*. For tractability, we require formulas generalized to IVs. When one considers instructions with immediate values, this generalization is more complex than a simple symbolic match-and-replace. Consider the IV `sal r64, imm8`. Stimulating the application, for this IV, with operands `rbx` and 1, produces the following bvf:

$$\texttt{rbx} \quad := \quad \langle 63, 0\rangle(\underbrace{0}_{1} \smile \texttt{rbx} << \underbrace{0}_{57} \smile \underbrace{1}_{8})$$

Trivially, `rbx` can be replaced with symbol `OP1` to represent operand 1. In terms of the immediate, one might assume that substituting the 8-bit value 1 with symbol `OP2` would suffice. Consider, however the same IV when stimulated with `rbx` and an immediate value of 0xFF:

$$\texttt{rbx} \quad := \quad \langle 63, 0\rangle(\underbrace{0}_{1} \smile \texttt{rbx} << \underbrace{0}_{57} \smile \underbrace{0x3F}_{8})$$

This case demonstrates that generalization cannot be achieved by a simple match-and-replace, as value 0xFF is not found in the formula produced. This example will be revisited in the next section, as we discuss our approach for extracting semantics out of Strata.

4.2 Chum Extraction

We extract bvf's from Strata, per IV, into a serialized version of Chum. For register-only IVs, generating the bvf is accomplished by stimulating `stoke_debug_circuit` with an instruction that consists of the IV instantiated with *safe registers*. A match-and-replace is then done from the safe registers to open variables.

In choosing specific operands for the instruction to be learned, registers were chosen such that they would *generalizable*. For example, `rbx` and `rcx` were the chosen operands utilized when learning binary register-only 64-bit IVs. These are safe registers, as these registers are never directly written to unless specified as operands. As a counterexample, `rax` is an unsafe register as `cmpxchg` directly writes to it regardless of whether `rax` is provided as an operand.

Consider the following example, which generates the bvf for `add r64, r64`. Application `stoke_debug_circuit` is stimulated with `add rbx, rcx`, which produces the following bvf:

$$\texttt{add rbx, rcx}: \quad \texttt{rbx} \quad := \quad \langle 63, 0\rangle(\underbrace{0}_{1} \smile \texttt{rbx} + \underbrace{0}_{1} \smile \texttt{rcx})$$

This is match-and-replaced to:

$$\texttt{add r64, r64}: \quad \texttt{OP1} \quad := \quad \langle 63, 0\rangle(\underbrace{0}_{1} \smile \texttt{OP1} + \underbrace{0}_{1} \smile \texttt{OP2})$$

For non-register-only IVs, the extraction process is more involved. Each IV, *iv*, is mapped to a register-only IV that *supports* it, *iv′*. An IV, *iv′*, supports *iv* if *iv′* has the same mnemonic, number of operands, and each operand meets specific criteria for generalization. An operand in *iv* does not require generalization support if it is already a register. For a non-register operand, the criteria are based on its type and bit-length, as well as those of the corresponding register operand of *iv′*. The semantics for instruction *iv* are obtained by stimulating `stoke_debug_circuit` with *iv′* with safe registers, which are then match-and-replaced to open variables. Lastly, any operand size mismatches between the supporting and supported operands are resolved. We will now discuss the criteria for operand generalization and any required size mismatch resolution.

4.2.1 Generalization to Immediate Operands.

The register operand providing support for the immediate must be of equal or greater size. Consider the example from the previous section `sal r64, imm8`. This variant finds support from the variant `sal r64, r8`. Application `stoke_debug_circuit` is stimulated with `sal rbx, cl`, which produces the following bvf:

$$\begin{array}{l}\texttt{sal rbx, cl}:\\ \quad \texttt{rbx} := \langle 63,0\rangle(\underbrace{0}_{1}\smile \texttt{rbx} \ll \underbrace{0}_{57}\smile(\langle 7,0\rangle\texttt{rcx} \wedge \underbrace{0x3F}_{8}))\end{array}$$

Doing the match-and-replace yields:

$$\begin{array}{l}\texttt{sal r64, imm8}:\\ \quad \texttt{OP1} := \langle 63,0\rangle(\underbrace{0}_{1}\smile \texttt{OP1} \ll \underbrace{0}_{57}\smile(\langle 7,0\rangle\texttt{OP2} \wedge \underbrace{0x3F}_{8}))\end{array}$$

Operator $\wedge$ denotes standard bv conjunction. As there was no size mismatch between `cl` and `imm8` nothing further is required. In case of a size mismatch, a sign extension is introduced into the bvf after it is returned from `stoke_debug_circuit`. Consider `add r32, imm8`. It is supported by `add r32, r32`. Semantics are extracted by generalizing to open variables, and introducing sign-extension:

$$\begin{array}{l}\texttt{add ebx, ecx}:\\ \texttt{ebx} := \underbrace{0}_{32}\smile\langle 31,0\rangle(\underbrace{0}_{1}\smile\langle 31,0\rangle\texttt{ecx} + \underbrace{0}_{1}\smile\langle 31,0\rangle\texttt{ebx})\\ \texttt{add r32, imm8}:\\ \texttt{OP1} := \underbrace{0}_{32}\smile\langle 31,0\rangle(\underbrace{0}_{1}\smile\langle 31,0\rangle\underbrace{\texttt{sxtend(OP2)}}_{32} + \underbrace{0}_{1}\smile\langle 31,0\rangle\texttt{OP1})\end{array}$$

4.2.2 Generalization to Memory Operands.

Similar to the immediate case, the register operand providing support for the memory operand must be of equal or greater size. As memory operands can be written to, this generalization has another case to consider in resolving operand size mismatches. If the supporting operand is being written to, a slicing operator is introduced, truncating the bv to the appropriate size. Consider IV `movapd m128,xmm`. This is supported by `movapd xmm,xmm`. Application `stoke_debug_circuit` is stimulated with safe xmm-registers `xmm1` and `xmm2`.

$$\begin{array}{l}\texttt{movapd xmm1, xmm2}: \texttt{ymm1} := \langle 255,128\rangle\texttt{ymm1}\smile\langle 127,0\rangle\texttt{ymm2}\\ \texttt{movapd m128, xmm}:\\ \quad \texttt{OP1} := \langle 127,0\rangle(\langle 255,128\rangle\texttt{OP1}\smile\langle 127,0\rangle\texttt{OP2})\end{array}$$

In case the supporting register operand is larger than the supported memory operand, the upper parts of the register are truncated. This means that no further steps are required. Consider the following example for `addsd xmm,m64` which is supported by `addsd xmm,xmm`:

$$\begin{array}{l}\texttt{addsd xmm1, xmm2}: \quad \texttt{ymm1} := \langle 255,128\rangle\texttt{ymm1}\smile\\ \qquad \langle 127,64\rangle\texttt{ymm1}\smile(\langle 63,0\rangle\texttt{ymm1} +^{f} \langle 63,0\rangle\texttt{ymm2})\\ \texttt{addsd xmm, m64}: \quad \texttt{OP1} :=\\ \quad \langle 255,128\rangle\texttt{OP1}\smile\langle 127,64\rangle\texttt{OP1}\smile(\langle 63,0\rangle\texttt{OP1} +^{f} \langle 63,0\rangle\texttt{OP2})\end{array}$$

For some instructions, we do not use learned semantics. Some instructions have no Strata support. For these, we supply manually written semantics. For another 119 IVs (with 8-bit immediate operands), there is no register-only equivalent. For these IVs, Strata learns 256 distinct formulas per variant. Our methodology does not support these "brute-forced" formulas. We drafted manual semantics to support branching instructions such as jump and call, as well as stack operating instructions such as push and pop. The learned semantics of the parity flag are impractical for theorem proving, since they produce very large bvf's. We therefore express the semantics of the parity flag manually.

Figure 3 shows the semantics of the `sub` IV after extraction from Strata and embedding within Isabelle/HOL, for a 32-bit register $r32 = \langle 31,0\rangle r(r64,\sigma)$ and a 32-bit memory location $m32 = r_{mem}(a,4,\sigma)$. The instruction causes 6 state changes: the 64-bit register $r64$ corresponding to register $r32$ is completely overwritten with a bv consisting of 1.) zeroes for the upper 32 bits, and 2.) for the lower 32 bits the lower 32 bits of the result of two's complement subtraction in 33-bit mode. The input values come from reading register $r64$ and taking the lower 32 bits and from reading 4 bytes of memory from address a. The zero flag is obtained by checking whether the result is zero, and the carry- and sign flags check respectively bits 32 and 31. An overflow occurs when the most significant bits (msb) initially were different, whereas the msb's of the result and the initial value at a are equal. Note that we do not show the parity flag update.

$$\begin{array}{lcl}
r64 & := & \underbrace{0}_{32}\smile\langle 31,0\rangle(\underbrace{0}_{1}\smile\neg op2 + \underbrace{1}_{33} + \underbrace{0}_{1}\smile\langle 31,0\rangle(op1))\\
ZF & := & \langle 31,0\rangle(\underbrace{0}_{1}\smile\neg op2 + \underbrace{1}_{33} + \underbrace{0}_{1}\smile\langle 31,0\rangle(op1)) == \underbrace{0}_{32}\\
CF & := & \langle 32,32\rangle(\underbrace{0}_{1}\smile\neg op2 + \underbrace{1}_{33} + \underbrace{0}_{1}\smile\langle 31,0\rangle(op1)) == \underbrace{1}_{32}\\
SF & := & \langle 31,31\rangle(\underbrace{0}_{1}\smile\neg op2 + \underbrace{1}_{33} + \underbrace{0}_{1}\smile\langle 31,0\rangle(op1)) == \underbrace{1}_{32}\\
OF & := & \neg\langle 31,31\rangle(op2) == \underbrace{1}_{1} \longleftrightarrow \langle 31,31\rangle(op1 == \underbrace{1}_{1}) \wedge
\end{array}$$
$$\neg(\neg(\langle 31,31\rangle(op2)) == \underbrace{1}_{1} \longleftrightarrow$$
$$\qquad \langle 31,31\rangle(\underbrace{0}_{1}\smile\neg op2 + \underbrace{1}_{33} + \underbrace{0}_{1}\smile\langle 31,0\rangle(op1)) == \underbrace{1}_{1})$$
$$\text{where } op1 = r(r64,\sigma),\ op2 = r_{mem}(a,4,\sigma)$$

Figure 3. Learned semantics of `sub r32 m32` deeply embedded into Isabelle/HOL.

5 Big-Step Semantics

The machine model provides small-step semantics, i.e., semantics per instruction. Our objective is to find a high-level representation of the big-step semantics of blocks within the binary. Note that our definition of blocks characterizes larger chunks of the code than the traditional notion of "basic blocks", i.e., our notion of blocks also includes if-statements. Our aim is to prove an *equivalence theorem* of the form:

$$f(\sigma_b) = run(\sigma_b)$$

Here, function f should be defined using high-level, i.e., C-like, constructs only. For the main function in the binary, state σ_b will be the initial state. For the functions in other text sections we quantify over any state σ_b with the read-only data sections loaded. This ensures compositionality.

The approach we use is *formal symbolic execution*. The symbolic execution engine needs to tackle two challenges:

1. Since the semantics embedded into Isabelle/HOL are generated by a machine learning algorithm (instead of being hand-written) they generally are not in a form where they can be used for theorem proving directly. Consider, e.g., the semantics in Figure 3. The semantics are expressed in bv operations such as concatenation, bit slicing and logical operations, instead of arithmetic operations such as subtraction and (in)equality.
2. Whenever possible, the semantics of a sequence of one or more instructions needs to be simplified to a high-level operation.

Effectively, formal symbolic execution is implemented by adding a library of formally proven correct rewrite rules to the Isabelle simplifier. Whenever two subgoals are introduced, an if-then-else is introduced in the logic manually. This allows us to consider for each case whether we actually want to introduce an if-then-else, or whether we want to add a precondition that prevents one of the cases from happening.

5.1 Formal Symbolic Execution

The starting point is of the form:

$$run(\sigma_b) = ?f(\sigma_b)$$

Here, $?f$ is a function representing the high-level semantics of the block. Crucially, this function does not have to be defined when running symbolic execution. It is a *schematic variable*. A schematic variable in a lemma basically means that the final lemma as it will be proven and admitted to the Isabelle logic has not been formulated yet. Whenever the current goal has the form $g(\sigma_b) = ?f(\sigma_b)$, the proof can be stopped and the final formulation of the proven equivalence theorem becomes $run(\sigma_b) = g(\sigma_b)$.

Symbolic execution will rewrite and simplify the left hand side of the goal. This will rewrite *run* to a function f representing a high-level equivalent of *run*.

Example 5.1. Consider the following assembly code:

```
push rbp
mov  rbp, 0
pop  rbp
```

This code first decrements the stack pointer, writes the frame pointer `rbp` into memory at location `rsp` − 8, and increments `rip`. Second, it writes the immediate value 0 to `rbp`. Third, it writes the value in the memory back to register `rbp`, then increments the stack pointer, and increments `rip` again. All these actions are executed symbolically. The equivalence theorem becomes:

$$run(\sigma_b) = (\texttt{rip} := \texttt{rip} + 9; \texttt{rsp} - 8 \triangleright \texttt{rbp})(\sigma_b)$$

Function f is represented as two state updates: register `rip` is incremented by 9, and the frame pointer is stored in memory (notation $a \triangleright v$ denotes writing value v into the memory at address a). Note that both registers `rsp` and `rbp` are unchanged, even though they have been changed during execution.

We have developed an Isabelle *proof method* using Eisbach [27]. A proof method is essentially a proof script that can be used to automate certain tasks. Our proof method is called `symbolic_execution`. It can be applied to a goal of the following form:

$$run(\sigma) = f(\sigma_b).$$

If the current goal has this form, method `symbolic_execution` does the following:

1. Check the termination condition on state σ. In case of termination, rewrite $run(\sigma)$ to σ and stop the proof method.
2. Fetch the next instruction based on the current state σ.
3. Check whether Strata semantics are available for that IV. If not check whether there is a manually written one available.
4. Simplify the semantics before applying them to the current state.
5. Apply the simplified semantics to the current state σ, producing a new state σ'.
6. Simplify state σ' to state σ''.

The resulting goal is of the form $run(\sigma'') = f(\sigma_b)$, where σ'' is the result of execution of one instruction on σ. Whenever the goal splits into two subgoals, we either introduce an if-then-else in function f, or add a precondition to exclude one of the two cases.

5.2 Rewrite Rules

We provide some interesting examples of rewrite rules that are applied when doing symbolic execution in Figure 4. For more details and proofs, we refer to the sources online.

Rules 1 to 8 are examples of rules that deal with word arithmetic, logical operations, bit operations such as shifting and concatenation, and bit slicing. The standard Isabelle/HOL library provides a strong library and support to apply SMT solvers such a Z3 [10] and CVC4 [5] to the current subgoal. We have augmented this library, especially to deal with cases of under- and overflow of memory addresses.

Rules 9 and 10 are examples of rewrite rules for *memory*. The read- and write operations must, whenever applicable, behave as a lens [12]. The first rule deals with writing to and reading from the same address a, and the same size s. The second rule deals with the case where these differ and do not overlap. Here, the operator $\bowtie$ takes as input two blocks in the

$$
\begin{aligned}
m < n \Rightarrow \langle m, 0\rangle(\underbracket{a + b}_{n}) &\equiv \langle m, 0\rangle(\underbracket{a}_{n}) + \langle m, 0\rangle(\underbracket{b}_{n}) \\
m < n \wedge m < l \Rightarrow \langle m, 0\rangle(\overbracket{\text{zxtend}(\underbracket{a}_{n})}^{l}) &\equiv \langle m, 0\rangle(\underbracket{a}_{n}) \\
m < n \Rightarrow \langle m, 0\rangle(\underbracket{\neg a}_{n}) &\equiv \neg\langle m, 0\rangle(\underbracket{a}_{n}) \\
\langle m, 0\rangle(\ \underbracket{a}_{m+1}\ ) &\equiv \underbracket{a}_{m+1} \\
\neg a + 1 + b &\equiv b - a \\
\overbracket{\text{zxtend}(\underbracket{\neg a}_{n})}^{n+1} &\equiv \overbracket{\neg(2^n + \text{zxtend}(\underbracket{a}_{n}))}^{n+1} \\
\langle n, n\rangle\ \underbracket{a}_{n+1} &\equiv \underbracket{a \geq 2^n}_{n+1} \\
m < n \Rightarrow \underbracket{\text{zxtend}(\overbracket{a}^{m}) < \text{zxtend}(\overbracket{b}^{m})}_{n} &\equiv \overbracket{a}^{m} < \overbracket{b}^{m} \\
r_{mem}(a, s, w_{mem}(v, a, s, \sigma)) &\equiv v \\
(a, s) \bowtie (a', s') &\Longrightarrow \\
r_{mem}(a, s, w_{mem}(v, a', s', \sigma)) &\equiv r_{mem}(a, s, \sigma) \\
(\underbracket{a}_{n} \wedge 0x7FFF\ldots) &\equiv |a|^f \\
(|\langle 63, 32\rangle\underbracket{a}_{64}\rangle|^f = \langle 31, 0\rangle a = 0) &\equiv a \in \{0^-, 0^+\}
\end{aligned}
$$

Figure 4. Examples of rewrite rules

memory and expresses separation. Without the assumption, the write may overwrite some of the bytes in block (a, s), invalidating the lemma. Rules for read/write with overlap are added, and rules for write/write. These rules are crucial in providing a higher level of abstraction: values that are written into memory are split into bytes and then written into memory in little-endian fashion. Reading then reverses the order and subsequently concatenates them. The rules simplify this behavior to a form where none of this is visible.

As discussed in Section 3.2.1, the semantics of floating point instructions such as addsd are expressed in terms of constrained functions. Based on those constraints, rewrite rules are proven. For concrete operations, e.g., the absolute function, we provide rules that simplify bit-level operations to floating point functions (e.g., Rule 11). Common bit-patterns and operations concerning the floating point values 0^- and 0^+, are rewritten (Rule 12).

In total, the library consists of approximately 330 rewrite lemmas, constituting approximately 10000 lines of Isabelle code. This excludes the case studies and the parsers.

5.3 Pointers

We introduce the dereference operator for reading from memory: $*[a, s]$ means reading s bytes from address a. Since we do not extract typed code, we have to add the size s. Consider the following line of C code (the code is compiled using gcc):

```
unsigned x = argv[1][0] - '0';
```

The return value of the program is stored in register eax. Running symbolic execution produces the following equivalence theorem:

$$run(\sigma_b) = (\texttt{eax} := \underbracket{\text{sxtend}(*[*[\texttt{rsi} + 8, 8], 1]) - 48}_{32}; \ldots)(\sigma_b)$$

The address stored in register rsi (storing argv, the second operand of the main function) is first incremented by eight, to get the address argv[1]. That value is then dereferenced to obtain the value argv[1]. That value is treated as address and dereferenced, producing value argv[1][0]. The value is cast to an unsigned int by sign-extension, after which 48 is subtracted. Note that symbolic execution cannot provide type information: the value 48 is subtracted, and it is not inferred that this value is actually the character '0'. What is inferred is that the first dereferencing operator produces an 8-byte value (in this case an address), and the second dereferencing produces a 1-byte value (in this case a char). Moreover, the final result is in 32-bit mode (in this example an int).

```
int main () {
   int var[] = {10, 100, 200};
   int i, *ptr;
   ptr = var;
   ptr += 2;
   i = *ptr;
   i *= 3;
   return i;
}
```

Figure 5. Pointer arithmetic

Consider the example in Figure 5, with pointer arithmetic. This program will always return the value 600. We purposefully compile this program without optimizations to ensure that the binary actually performs the computations to derive the return value. Symbolic execution rewrites the semantics for the eax register to the constant value 600. However, an additional assumption is required. Since this program dereferences a computed pointer, a StackGuard is introduced by the gcc compiler [8]. This is a security mechanism, trying to detect when a stack smashing attack occurs, i.e., an overflow of the stack. The mechanism copies – at the beginning of the function body – an unknown value (the *canary*) to an address offset by the fs register:

```
  mov rax, qword ptr fs:[0x28]
  mov qword ptr [rbp - 8], rax
```

Before return, the canary is read from the memory into rcx and compared to the value currently stored at the address.

```
xor   rcx, qword ptr fs:[0x28]
```

A stack smash attack is detected when this result is not zero (using a je instruction), i.e., when the canary has been overwritten. In that case, the function will not return normally, but fail. Symbolic execution of instruction je introduces two subgoals. A precondition is added, to exclude the case where the canary is overwritten. That precondition suffices to prove that this program indeed returns 600. The equivalence theorem becomes:

$$(\texttt{rsp}-52, 52) \bowtie (\texttt{fs}+40, 8) \Longrightarrow run(\sigma_b) = (\texttt{eax} := 600; \ldots)(\sigma_b)$$

The precondition ensures memory separation between two blocks of memory. The first block is the stack frame with 52 bytes, and the second is the address fs + 40 with size 8. The theorem needs to assume that the canary does not overlap with the stack frame. Given that assumption, symbolic execution can automatically derive the intended semantics.

5.4 Floating Points

Consider the C code in Figure 6. The example has been provided by the US Air Force Research Laboratory. It computes the current speed of some object. If that speed exceeds a maximum value of 58.1152 m/s, an exception is thrown. The three get_ functions return some unknown value, resp. *speed*, *brake*, and *accel*.

```
const double vehicleMass=1000; //in kg
const double timeStep=.001; //1 ms
int updateDisplaySpeed(void) {
  double currentS=get_current_speed();
  double brakeF=get_current_braking_force();
  double accelF=get_current_accel_force();
  double newS=0.0;
  newS=currentS + ((accelF - brakeF)
            / vehicleMass * timeStep);
  if (newS > 58.1152) { /* exception */ }
  return (newS);
}
```

Figure 6. C code with floating point computations

Running symbolic execution produces two subgoals, due to the if. Instead of extracting an if-statement in the logic, we have added a precondition preventing the exception from happening. The equivalence theorem becomes:

$$\texttt{let } v = (accel -^f brake) \div^f \texttt{0x408F400000000000} *^f \texttt{0xFCA9F1D24D62503F} +^f speed \texttt{ in0xE63FA4DFBE0E4D40} \leq^f v \Longrightarrow run(\sigma_b) = (\texttt{xmm0} := v; \ldots)(\sigma_b)$$

The hexadecimal constants are loaded from the data sections of the binary, and are the floating point constants occurring in the C code. The return value is stored in the xmm0 register.

5.5 Memcpy

Consider the following C code:

```
void swap (void* a0, void* a1) {
  const char temp[9];
  memcpy((void*) temp, a0, 9);
  memcpy(a0, a1, 9);
  memcpy(a1, temp, 9);
}
```

The code swaps 9 bytes at the given addresses. We have compiled the code on Linux with gcc, optimization level 3. Since a constant number of bytes is copied, this will replace the memcpy function calls with inline assembly (we swap 9 bytes, since less bytes results in much simpler code; then 64-bit registers can be used to perform the swap). This means that direct verification of this program not only requires a formal C semantics, but additionally a semantics of assembly, and a semantics to calling assembly from C.

The program writes bytes into memory. Function bytes_of takes as input a word and produces a list of byte-sized chunks, such that the least significant byte comes last. Function rev is the reverse function. The equivalence theorem becomes:

$$run(\sigma_b) = \begin{array}{l}(\texttt{rdi} \triangleright \text{rev}(\text{bytes\_of} *[\texttt{rsi}, 8]); \\ \ \texttt{rdi} + 8 \triangleright \text{rev}(\text{bytes\_of} *[\texttt{rsi} + 8, 1]); \ldots)\end{array} (\sigma_b)$$

After termination of the block, the 9 bytes stored at the address in rdi are the 9 bytes initially stored at the address in rsi in little-endian fashion. It is easy to prove that reading 9 bytes from the address in rdi produces the original 9 bytes at the address in rsi.

6 Testing

We conduct testing of the machine model from within Isabelle/HOL. For each IV, we create Isabelle/HOL test lemmas. These test lemmas formulates that for a certain pre-execution state the formalized semantics compute a correct post-execution state. Each lemma is then proven automatically using the proof method described in the previous section. Testing makes sure that no errors are made during the Chum extraction and deep embedding into Isabelle/HOL. Most importantly, it validates the generalization arguments made in Section 4.2.

A *test case* consists of a pre- and post-execution state. For each IV, 6630 test cases are generated. The pre-execution states are determined from the set Strata used as part of their stochastic search methodology. This is significantly better than just random testing, since they also include contain cases learned by counter-example guided refinement. The post-execution states are computed using dynamically generated binaries executed on live x86-64 hardware (Skylake architecture). These binaries each consist of the instruction under test, and – if applicable – a read/write data segment for memory operands. The binaries are generated from templates. We show the template for instructions with a memory source operand. For each test case, variables preceded by a % are replaced by the appropriate values.

```
MemOperand:
        .quad %mem_value
        .section .text
        .align 64
        .globl main
        .type main, @function
main:
        %mnemonic %register,[MemOperand]
        ret
```

We utilize a custom built Pintool with Pin [26] to instrument these binaries. For each test case, the Pintool injects the pre-execution state into hardware, executes the instruction, and extracts the post-execution state prior to executing the `ret`.

In Isabelle/HOL, each test case produces a test lemma using lemma templates. The following template is for register-only 2-ary IVs:

$$\begin{array}{l}\texttt{lemma Test\_Case} : \forall dst, src :: \texttt{reg} \cdot src \neq dst \implies \\ \quad \text{let } i = \%mnemonic\ dst, src; \\ \quad \sigma_{pre} = \sigma(dst := \%dst,\ src := \%src, flags := \%flags) \\ \quad \text{in} \\ \quad \mathcal{M}(i, \sigma_{pre}) := \sigma(dst := \%dst', src = \%src', flags := \%flags')\end{array}$$

These test lemmas show the step function within the machine model produces the state transformation as described in the test case.

Proving this test lemma effectively achieves *concolic testing*: instead of specifying the complete pre- and post-execution state, only those portions of the state operated on are concretely specified. Concolic testing dramatically increases the amount of concrete states covered by a single test case. Moreover, one test lemma tests all possible combinations of registers / memory addresses.

In total, we tested 886 IVs. The IVs tested cover a wide range of IVs operating on general registers, SIMD registers, memory operands up to 256-bit and immediates. Out of scope for consideration are instructions that contained uninterpreted functions (439), optimized versions of another variant (162), and ternary (124). Additionally, 14 variants were not tested due to the length of the bvf's and their slow execution (`blsi`, `tznt`, and `bt`).

Two IVs failed on testing: `movss xmm, m32` and `movsd xmm, m64`. These two instructions fail to meet the generalization argument from register to memory for a similar reason. Consider the following semantic:

$$\begin{array}{l}\texttt{movsd xmm1, xmm2} : \\ \quad \texttt{ymm1} := \langle 255,128\rangle\texttt{ymm1} \smile (\langle 127,64\rangle\texttt{ymm1} \smile \langle 63,0\rangle\texttt{ymm2})\end{array}$$

Per the generalization argument to memory we would expect the following for `movsd xmm, m64`:

$$\begin{array}{l}\texttt{movsd xmm, m64} : \\ \quad \texttt{OP1} := \langle 255,128\rangle\texttt{OP1} \smile (\langle 127,64\rangle\texttt{OP1} \smile \langle 63,0\rangle\texttt{OP2})\end{array}$$

However, the actual semantics is:

$$\texttt{OP1} := \langle 255,128\rangle\texttt{OP1} \smile (\underbrace{0}_{64} \smile \langle 63,0\rangle\texttt{OP2})$$

In these two variants, while they generalize correctly in terms of reading operand 2, they introduce novel behavior when writing to operand 1, with respect to the supporting IV. We modeled these IVs manually in order to support case studies that used these instructions.

7 Case Study: FDLIBM IEEE754 Remainder Function

Figure 7 shows the source code of the IEEE 754 remainder function for floating points from Sun's FDLIBM library. The C code defines functions `__HI` and `__LO` specifically for a little-endian architecture. These functions are used to obtain the high and low 32 bits of parameters' x (the numerator) and p (the denominator). The code performs a series of checks dealing with division by zero, infinite values, and NaN's. It then uses the modulo function to normalize x to a value less than $2p$. Subsequently, it performs a series of if-then-else's to compute the result. The final line restores the sign bit.

The text section in the binary belonging to the remainder function consists of 157 lines of assembly code. The binary contains 3 data sections totaling 158 bytes of data. The code:

- contains various instructions from the SSE2 instruction set;
- contains 17 conditional jumps; based on the carry-, zero-, and sign-flags;
- reads from memory to access the data sections in the binary;
- uses the `lea` instruction to do pointer arithmetic;
- builds a return value by overwriting only parts of it, i.e., the lower- and higher 32 bits of the 64-bit return value are computed separately.

The right hand side of Figure 7 shows the semantics lifted out of the binary. We show the value that is stored in register `xmm0` after execution of the block, i.e., the return value of the function. The proof is largely automated: the proof consists of repeatedly applying the method `symbolic_execution` until it fails, i.e., until it is no longer able to rewrite the current subgoal. In all those cases, the current subgoal could be discharged with standard Isabelle/HOL tools, such as `simp` and `auto`. No additional lemmas where required, and no interactive theorem proving such as induction, generalization, or quantifier-reasoning. We did introduce a *cut*. The current proof has been split up into two parts, corresponding to the point where the first x' is introduced. A cut can reduce the number of instructions to be symbolically executed. For this

```c
#define __HI(x) *(1+(int*)&x)
#define __LO(x) *(int*)&x
double rem(double x, double p) {
  int hx,hp; unsigned sx,lx,lp;
  double p_half;
  hx = __HI(x);   lx = __LO(x);
  hp = __HI(p);   lp = __LO(p);
  sx = hx & 0x80000000;
  hp &= 0x7fffffff;
  hx &= 0x7fffffff;
  if((hp|lp)==0)
    return (x*p)/(x*p);
  if ((hx>=0x7ff00000) ||
      ((hp>=0x7ff00000) &&
      (((hp-0x7ff00000)|lp)!=0)))
    return (x*p)/(x*p);
  if (hp <= 0x7fdfffff)
    x = mod(x,p + p);
  if (((hx-hp) | (lx-lp))==0)
    return zero * x;
  x = fabs(x);
  p = fabs(p);
  if (hp < 0x00200000) {
    if(x+x>p) {
      x -= p;
      if(x + x >= p) x -= p;
    }
  } else {
    p_half = 0.5*p;
    if (x > p_half) {
      x -= p;
      if(x >= p_half) x -= p;
    }
  }
  __HI(x) ^= sx;
  return x;
}
```

(a) Source code

```
let $x$ = ymm0;
    $p$ = ymm1;
    $x_{lo} = \langle 31, 0\rangle x$;
    $x_{hi} = \langle 63, 32\rangle x$;
    $p_{lo} = \langle 31, 0\rangle p$;
    $p_{hi} = \langle 63, 32\rangle p$;
in
if $p \in \{0^-, 0^+\}$
  $\vee\ |x_{hi}|^f >$ 0x7FEFFFFF
  $\vee\ ((|p_{hi}|^f >$ 0x7FEFFFFF$) \wedge$
     $(|p_{hi}|^f +$ 0x80100000 $\neq 0 \vee p_{lo} \neq 0))$ then
   $(p *^f x) \div^f (p *^f x)$
elseif $x_{lo} = p_{lo} \wedge |x_{hi}|^f = |p_{hi}|^f$ then
  $0^+ *^f x$
else
 let $x'$ = if $|p_{hi}|^f >$ 0x7FDFFFFF then $x$
           else mod$(x, p +^f p)$) in
 let $x'$ =
  if $|p_{hi}|^f >$ 0x1FFFFF then
   if $|p|^f *^f 0.5 \leq^f |x'|^f$ then
    $|x'|^f$
   elseif $|p|^f *^f 0.5 <^f (|x'|^f -^f |p|^f)$ then
    $|x'|^f -^f |p|^f$
   else
    $|x'|^f -^f |p|^f -^f |p|^f$
  else
   if $|p|^f \leq^f |x'|^f + |x'|^f$ then
    $|x'|^f$
   elseif $|p|^f <^f |x'|^f -^f |p|^f +^f (|x'|^f -^f |p|^f)$
    $|x'|^f -^f |p|^f$
   else
    $|x'|^f -^f |p|^f -^f |p|^f$
 in
  xor_sign$(x'$, msb$(x))$
```

(b) Semantics extracted from binary

Figure 7. FDLIBM IEEE754 floating point remainder. Source code is shown instead of the assembly, due to space limitation. Only the binary is used when applying the methodology.

example, the current proof requires symbolic execution of 231 instructions. This is more than the 157 from the text section, since conditional jumps can cause instructions to be executed twice. Without introducing the cut-point, a significantly larger amount of instructions are executed twice (or more). Introducing a cut thus improves verification time.

We have added one function to the Isabelle/HOL logic specifically for this case. Function *xor_sign* takes as input a 64-bit word w and a Boolean b. It XOR's the sign bit of w with b:

$$\text{xor\_sign}(\underbrace{w}_{n}, b) \equiv \text{set\_bit}(n-1, \text{msb}(w) \neq b)$$

We then add rewrite lemmas for this function:

$$\underbrace{a \oplus b \wedge 2^{n-1}}_{n} \quad \equiv \quad \text{xor\_sign}(a, \text{msb}(b)) \qquad (1)$$

$$\text{xor\_sign}(|a|^f, \text{msb}(a)) \quad \equiv \quad a \qquad (2)$$

Rule 1 introduces function xor_sign, when certain bit operations occur. When given an absolute value and the msb of that value, the function has no effect (Rule 2).

The construction of the high-level specification, i.e., the code in Figure 7b, is largely automated as well. There are two types of user interaction required. The first concerns the introduction of if-then-else statements. The second interaction is introducing `let`-constructs. Whenever a value occurs twice, such as the value $\langle 31, 0\rangle x$, it is determined manually whether it makes sense to introduce a `let`. Semantically, this makes no difference. However, without this interaction the extracted semantics can become significantly larger and more unreadable. Using `let` essentially allows to introduce local variables. For sake of presentation, we have matched the variable names to the ones in the original C code.

The control flow structure between the C code and the lifted semantics are different, but similar. Since the library of rewrite rules is sometimes able to rewrite bv operations to arithmetic, the lifted branching conditions can be on a higher level of abstraction than the original C code. For example, the branching condition `(hp | lp) == 0` uses a logical bv operation, whereas the lifted semantics branch on $p \in \{0^-, 0^+\}$. As another example, the C code branches on `((hx - hp) | (lx - lp)) == 0`. The equivalent branching condition in the semantics lifted out of the binary is expressed solely in arithmetic operations: $x_{lo} = p_{lo} \wedge |x_{hi}|^f = |p_{hi}|^f$.

8 Related Work

Figure 8 shows a summary of related work. We consider DiL, x86 machine models and their testing methodologies, and binary verification efforts in general. We compare it to our approach, called Leviathan.

8.1 Decompilation-Into-Logic

Binary verification mandates an underlying mechanism for lifting machine code into logic with associated pre- and post-conditions for correctness. Myreen et al. [29, 30] present such an approach, architecture-agnostic.

Our implementation differs on one aspect: our approach allows decompiling assembly source code as well as machine code. The difference is in the fact that assembly code is often position-independent (symbolized memory references). This requires proofs to quantify over all possible address layouts that an assembler could make. Position-dependent machine code does not require this, as everything is resolved to a specific (virtual) address.

Our methodology, like existing works, is able to scale to large binaries. We successfully decompiled-into-logic the binary of gcc 4.9 with approximately 361K lines machine code in 1.3 hours on a machine with a 6-core Intel i9-8950HK CPU running at between 2.9 and 4.8Ghz. Myreen et al. have applied DiL to 1.3k lines of assembly within 1.5 hours.

8.2 Machine Models

Underneath any binary verification effort is a machine model of the ISA. RISC architectures [2] have smaller instruction sets, and the ARM instruction set is well modeled [13]. All related works that tackle x86 do so by manually coding operational semantics based on semi-formally specified Intel documentation [16]. Generally, support is lacking for more complex x86 instructions added within later processor revisions.

Sarkar et al. introduce a formal x86 machine model in HOL4 [33] supporting 33 IVs. Their model is supported by a DiL framework. The focus of their work was concurrency: proving memory consistency levels in multiprocessor execution. As of the latest release of HOL: Kananaskis-12, this model has been extended to 114 IVs.

Kaufmann and Hunt developed an x86 machine model in ACL2 supporting 21 instructions [18, 19]. Goel et al. extended this work to 407 IVs and used those semantics to verify a modified word-count program that avoids AVX instructions [14, 15]. A major contribution of this work was to formalize system call behavior.

Leroy et al. developed CompCert: a certified compiler which relies on an ISA machine model to perform co-simulation between a pre- and post-compiled program to verify compilation [23–25]. This work initially targeted the PowerPC, but has since been extended to x86 as of CompCert v3.4, with support for 172 IVs.

8.3 Testing Methodologies

Testing methodologies found in related works test using either 1.) likely execution sequences (LE) or 2.) random execution (RE). Testing over likely execution sequences generalizes well to realistic code. It is less likely however to accurately cover undefined behavior or rarely executed code. Random (or fuzz) testing ensures a certain amount of coverage over each variant, and is more likely to find undefined behavior. Purely random testing, however, requires more test cases to provide coverage over likely execution paths, as more cases are utilized exploring potentially undefined or infrequent behavior.

Goel et al. utilized Pintool to verify co-simulation between their model and live hardware, over a benchmark application [15]. This approach tests the model over a likely sequence of execution. Sarkar et al [33], perform random testing on a sample of input/output pairs for each IV.

In our model, we perform RE, LE, and testing derived from an iterative counterexample guided refinement machine learning process (CE). The refinement strategy and associated test cases generated, tease out corner cases in instructions with more exotic/complex behavior [17]. We go further by making concolic versions of all three types of tests, which significantly increases coverage.

x86 Machine Models			
Model / Framework	IVs Supported (Total, Post Pentium-Pro)	Testing Methodology	DiL support
Leviathan	1625, 1210	concolic, RE, LE, CE	yes
Sarkar et al. [33]	114, 0	RE	yes
Goel et al [14, 15, 18, 19]	407, (unknown/no-SIMD)	co-simulation, LE	n/a
CompCert [23–25]	172, 16	unknown	n/a

Binary Verification Efforts			
Verification Effort	Architecture	Source Code Required	Verification Properties
Leviathan	x86	No	Big-Step Equivalence
Translation Validation seL4 [20, 37]	ARMv6	Yes	Functional Correctness
CakeML [22]	ARM/x86	Yes	Verified Compilation
CompCert [23–25]	PowerPC/ARM/x86	Yes	Verified Compilation
Costanzo et al [7]	x86	Yes	Confidentiality
Goel et al [15]	x86	Yes	Functional Correctness

Figure 8. Related x86 machine models and binary verification efforts

8.4 Binary Verification Efforts

One of the most influential current state-of-the-art formal verification efforts has been the work done in seL4 [20, 28]. The seL4 microkernel is an OS with a formal specification and a formal conformance proof that the implementation satisfies the specification. Subsequently, it is proven that the ARM binary conforms to the source code of the implementation. The properties proven include, but are not limited to, no buffer overflows, no null pointer dereferences, no memory leaks, and noninterference. Its binary verification effort is based on – among others – the work of Sewell et al. [37].

Various research targets verification of the compilation process [22–25, 37]. The CompCert project provides an optimizing C99 compiler with such guarantees. This achieved by correspondence proofs between each intermediate representation during the CompCert compilation process. It targets the PowerPC, ARM, RISC-V and x86 ISAs. Constanzo et al. leverage the CompCert machine model to verify information flows within multiple processes [7]. CakeML [22, 39] is a framework for verified compilation of a functional ML-like programming language. CakeML is available for several architectures, including RISC-V and x86-64.

9 Conclusion

To apply formal methods to systems where source code is unavailable, *bottom-up* formal verification is required. Bottom-up formal verification starts with a binary and considers it a black-box. This paper presents a fundament of bottom-up verification: a methodology for embedding a binary into a theorem prover and lifting it to a higher level of abstraction. It is shown that we can take blocks of assembly in x86-64 binaries and systematically derive a formally proven correct high-level representation of the semantics of those blocks.

Our methodology is largely automated. The proofs of conformance between the binary and the high-level semantics are taken care of by developed proof methods and standard off-the-shelf Isabelle/HOL tools. The formulation of the high-level representation is obtained by running formal symbolic execution. Interactively, when non-determinism occurs, a user can decide to introduce if-then-else statements in the high-level representation, or add preconditions to exclude the non-determinism. The exact branching condition (or precondition) is provided by the proof methods; the choice how to resolve the non-determinism is left to the user. Finally, a user can interactively introduce local variables to derive a more succinct high-level representation.

Bottom-up verification minimizes the TCB, as no compilers need to be trusted, nor is a semantical model of the source language needed. Obtaining a trustworthy machine model of an architecture as complex as x86-64 is a challenge. We have leveraged the machine learned semantics of Strata. To gain further trust in the machine model, test lemmas are proven within Isabelle/HOL that demonstrate equivalence between the machine model and instructions run on an actual machine.

We aim to apply this methodology to industrial control systems, which are characterized by relatively simple flow-control but advanced floating point formulas. This requires a strong reasoning engine over low-level models of floating point operations. In the near future, we want to deal with concurrency by combining our model with x86-TSO [35, 36]. Eventually, this will result in a reliable bottom-up verification methodology for concurrent, safety-critical systems.

Acknowledgments

This work is supported in part by ONR under grant N00014-17-1-2297. The authors would like to thank Dr. Peter Lammich for his useful insights while developing our methodology in Isabelle/HOL. The authors gratefully acknowledge the highly insightful feedback of Dr. Lok Yan, US AFRL, and the reviewers, which have significantly improved the paper.

References

[1] [n. d.]. *IEEE Standard for Floating-Point Arithmetic.* https://doi.org/10.1109/IEEESTD.2008.4610935

[2] ARM ARM. 2012. Architecture Reference Manual. *ARMv7-A and ARMv7-R edition* (2012).

[3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86 – A platform for analyzing x86 executables. In *International Conference on Compiler Construction.* Springer, 250–254.

[4] Clemens Ballarin. 2003. Locales and locale expressions in Isabelle/Isar. In *International Workshop on Types for Proofs and Programs.* Springer, 34–50.

[5] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification.* Springer, 171–177.

[6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.

[7] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for C and assembly programs. *ACM SIGPLAN Notices* 51, 6 (2016), 648–664.

[8] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo.*

[9] Jeremy Dawson. 2009. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 55–70.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[11] Chris Eagle. 2011. *The IDA pro book.* No Starch Press.

[12] John Nathan Foster. 2009. *Bidirectional programming languages.* Ph.D. Dissertation. University of Pennsylvania.

[13] Anthony Fox and Magnus O Myreen. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *International Conference on Interactive Theorem Proving.* Springer, 243–258.

[14] Shilpi Goel, Warren A Hunt, and Matt Kaufmann. 2017. Engineering a formal, executable x86 ISA simulator for software verification. In *Provably Correct Systems.* Springer, 173–209.

[15] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. 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 (FMCAD '14).* FMCAD Inc, Austin, TX, Article 18, 8 pages. http://dl.acm.org/citation.cfm?id=2682923.2682944

[16] Part Guide. 2011. Intel® 64 and IA-32 Architectures Software DeveloperâĂŹs Manual. *Volume 3B: System programming Guide, Part 2* (2011).

[17] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[18] Warren Hunt Jr and Matt Kaufmann. 2012. *Towards a Formal Model of the X86 ISA.* Technical Report. University of Texas at Austin Austin United States.

[19] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2013. *Computer-aided reasoning: ACL2 case studies.* Vol. 4. Springer Science & Business Media.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 207–220.

[21] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *International Conference on Interactive Theorem Proving (ITP'18).*

[22] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 179–191.

[23] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[24] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

[25] Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and userâĂŹs manual. INRIA Paris-Rocquencourt* (2012).

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[27] Daniel Matichuk, Toby Murray, and Makarius Wenzel. 2016. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning* 56, 3 (2016), 261–282.

[28] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 415–429.

[29] Magnus O Myreen, Michael JC Gordon, and Konrad Slind. 2012. Decompilation into logic – Improved. In *Formal Methods in Computer-Aided Design (FMCAD), 2012.* IEEE, 78–81.

[30] M. O. Myreen, M. J. C. Gordon, and K. Slind. 2008. Machine-Code Verification for Multiple Architectures – An Application of Decompilation into Logic. In *2008 Formal Methods in Computer-Aided Design.* 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.24

[31] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media.

[32] John Rushby. 1997. Formal methods and their role in the certification of critical systems. In *Safety and reliability of software based systems.* Springer, 1–42.

[33] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 379–391.

[34] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 305–316.

[35] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 22.

[36] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[37] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 471–482.

[38] Sardar Muhammad Sulaman, Alma Orucevic-Alagic, Markus Borg, Krzysztof Wnuk, Martin Höst, and Jose Luis de la Vara. 2014. Development of Safety-Critical Software Systems Using Open Source Software – A Systematic Map. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on.* IEEE, 17–24.

[39] Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 60–73.

[40] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*.

[41] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal Methods: Practice and Experience. *ACM Computing Survey* 41, 4, Article 19 (Oct. 2009), 36 pages. https://doi.org/10.1145/1592434.1592436