A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture

Anonymous Author(s)

Abstract

We present the most complete and thoroughly tested formal semantics of x86-64 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 also demonstrate the applicability 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.

Keywords x86-64, ISA specification, Formal Semantics

1 Introduction

The x86-64 instruction set architecture (ISA) is one of the most complex and widely used ISAs on servers and desktops, and ensuring the correctness of the x86-64 binary code is important. The ability to directly reason about the binary code is desirable, not only because it allows to analyze the binary even when the source code is not available (e.g., legacy code or malware), but also because it avoids the need to trust the correctness of compilers [18, 49].

A formal semantics of x86-64 is required for formal reasoning about binary code, one of the strongest ways to ensure its correctness. An *executable* semantics is especially powerful because it allows direct testing to gain confidence in the definitions of the semantics, and also because it can allow additional tools based on symbolic execution, like deductive verification and symbolic test generation. Completely formalizing the semantics of x86-64, however, is challenging especially due to the complexity and the sheer number of instructions that are informally specified in approximately 3,000-page standard [7].

Existing Semantics for x86-64: To date, to the best of our knowledge, despite several explicit attempts [24–26, 30] and other related systems [10, 28, 29, 33, 34], no *complete* formal semantics of x86-64 exists that can be used for formal reasoning about x86 binary programs. Heule *et al.* [30] presented a formal semantics of x86-64, but it covers only a fragment (~47%) of all instructions; as the authors of [30] 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 3.6 & 4). Goel *et al.* [24–26], on the other hand, specified a formal semantics in the ACL proof assistant [31], allowing to reason about functional correctness, but their semantics covers only a small fragment (~33%) of all user-level instructions. There also have been several attempts [17, 20, 28, 46] 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 7 for a more detailed comparison to existing semantics.

Our Approach: We present the most complete and thoroughly tested formal semantics of user-level x86-64 to date. We employed the $\mathbb{K}$ framework [45] (Section 2.1) 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.* [30]'s semantics (Section 2.2) 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 [7] to obtain the complete semantics. A manual specification may sound 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 [30] and Hasabnis *et al.* [28, 29]) suffer from scalability and/or faithfulness issues (see Section 3.2 and 7 for details), makes the effort worth undertaking.

Like closely related previous work [26, 30], 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 [39, 43], 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

PL'18, January 01–03, 2018, New York, NY, USA
2018.

promising direction toward rigorously reasoning about real-world programs running on modern multiprocessors. We leave it for future work.

Contributions: We describe our primary contributions:

- *Completeness.* 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, except deprecated ones, the AES cryptography extensions, and the concurrency-specific instructions, that is, 3155 instructions covering 774 mnemonics [7] (Section 3.1).
- *Faithfulness.* Being executable, our semantics has been thoroughly tested against 7,000+ test cases using the co-simulation method (Section 4). We found errors in both the x86-64 standard document and other existing semantics including the baseline semantics (Section 4).
- *Generality.* We demonstrate that our semantics can be used for various formal analyses, such as symbolic execution, deductive program verification, and program equivalence checking (Section 5). The $\mathbb{K}$ framework also enables to represent our semantics as SMT theories, which allows others to easily reuse our semantics for their own purposes.

Our formal semantics is publicly available at [11].

1.1 Challenges in Formalizing x86-64

Size and Complexity: The x86-64 ISA has a vast 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 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 quite 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 3.6, 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) 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 the undefined case.

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 [10] updates the flag with 0, whereas Radare [17] 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 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).

2 Preliminaries

Here we provide background on the $\mathbb{K}$ framework and the Strata project [30] (used for our baseline semantics).

2.1 K Framework

$\mathbb{K}$ [45] 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 both in terms of applicability of the semantics and in terms of engineering the semantics itself.

We refer the reader to [41, 45] for details. In a nutshell, in $\mathbb{K}$, a language syntax is given using conventional Backus-Naur

[1]These numbers are obtained by parsing the official manual "Volume 2: Instruction Set Reference" and cross checked with projects [14, 44] investing similar efforts.

Form (BNF). A language semantics is given as a parametric transition system, specifically a set of reduction rules over configurations. A configuration is an algebraic representation of the program code and state. Intuitively, it is a tuple whose elements (called cells) are labeled and possibly nested. Each cell 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.

2.2 Strata Project

Strata [30] 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 [44] 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.

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 Formalization of x86-64 Semantics

This section presents how we get the complete semantics of all the user-level instructions. Section 3.1 details the scope of our work. Section 3.6 mentions how we leverage the information available in Strata, our baseline semantics. Section 3.3 explains how we formalize our model in $\mathbb{K}$.

3.1 Scope of the Work

We support all but a few non-deprecated user-level instructions, amounting to 3155 total variants of the Haswell x86-64 ISA (representing 774 out of 996 unique mnemonics). The entire implementation took 8 man-months (not including extensive time spent on projects related to binary decompilation that gave the lead author relevant experience and strong familiarity with the x86-64 architecture and documentation). Below is a summary of the instruction categories that we support:

- **General-Purpose Instructions:** These implement data-movement, arithmetic, logic, control-flow, string operations (including repeated- and fast- string operations).
- **Streaming SIMD Extensions (SSE) Instructions & 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 (or AVX) & subsequent extensions (Fused-Multiply-Add (or 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: (1) System-level instructions, which are related to the operating system, protection levels, I/O, cache lines, and other supervisor instructions; (2) x87 & MMX instructions, consisting of legacy floating-point and vector operations, respectively, which are now superseded by SSE; (3) Concurrency-related operations, including atomic operations and fences; and (4) Cryptography instructions, which support cryptographic processing specified by Advanced Encryption Standard (AES).

3.2 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 for about 60% of the instructions in scope from the formal semantics in Strata, 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 4. First, we use the $\mathbb{K}$ interpreter to execute the semantics of each instruction for 7000+ 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 torture tests. 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 4.

3.3 Program Configuration

$$\Big\langle \langle \mathsf{K} \rangle_{\mathsf{k}} \;\; \langle \mathsf{ID}_{\mathsf{regname}} \mapsto \mathsf{Value} \rangle_{\mathsf{regstate}} \;\; \langle \mathsf{Address} \mapsto \mathsf{Value} \rangle_{\mathsf{memstate}} \Big\rangle_{\top}$$

Figure 1. 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 1. The cells are represented using angle brackets. The outer $\top$ cell contains the cells used during program evaluation. The inner k cell contains a list of computations to be executed. Below we describe the two other inner cells.

Register State The regstate cell contains a map from registers or flag names to values. 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 [22, 34]. The memstate cell is a map from 64-bit addresses to bytes, which specifies the byte-addressable memory, 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[2] of the memory into sections like code, data and stack. The following is an example snapshot of a memory state, holding a 4-bytes integer value 65535:

$$\left\langle \begin{array}{lll} 4 & \mapsto & \mathrm{byte}(0,\;\; 32'65535) \\ 5 & \mapsto & \mathrm{byte}(1,\;\; 32'65535) \\ 6 & \mapsto & \mathrm{byte}(2,\;\; 32'65535) \\ 7 & \mapsto & \mathrm{byte}(3,\;\; 32'65535) \end{array} \right\rangle_{\mathsf{memstate}}$$

This says that the memory address 4 stores the 0th byte of the bit-vector 32'65535[3], the address 5 stores the 1st byte, and so on. While memory read, requested bytes are aggregated according to the size of the memory access.

3.4 Semantics of Execution Environment

We now give the reader a flavor of our semantics, by discussing a few of the roughly 5, 200 rules[4] 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 semantics of execution of an x86-64 program involves initializing the memory with program instructions and then fetching the instructions from the memory one at a time and executing it. The instruction to be executed next is pointed to by the instruction pointer register %rip.

[2]These abstractions are useful for executing x86-64 programs.

[3]All the values and address are represented as bit-vectors which are depicted in this paper as $W'V$, and interpreted as a bit-vector of size W and value V.

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

The following rule is 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. We use ":T" to represent the type of a term used in a rule. For example, 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 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>
```

The symbol ↷ is used to separate the computations in the k cell. 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, then match L in memstate to find the mapped instruction. The 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 and updates the value of %rip. The execution will be terminated when there is no instruction stored in the memory at the address pointed to by %rip.[5]

3.5 Semantics of Individual Instructions

Here we explain how we define the semantics of an instruction in 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.

[5]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.

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) ⇒ BV_r ...</k>
  <regstate>... R ↦ BV_r ...</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, depicted as getRegisterVal(%rbx), and regstate contains a register with name "RBX". This rule resolves the register lookup to the mapped bit-vector value BV$_r$.

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 ⇒ ( 64'Offset + BV_r):Address
...</k>
  <regstate> ... R ↦ BV_r ... </regstate>
```

The term to the left of ⇒ shows the memory addressing expression at the top of k cell, which gets reduced to a memory address. The effective memory address for the memory operand used in the running example is (64'-4 + BV$_{rsp}$) which is then used to do memory read access. The rules associated with memory read is responsible to read a memory value of requested number of bits (64-bits for the current example) starting from the effective memory address.

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 following rule realizes the destination write operation, where memVal$_{64}$ and BV$_r$ represents the 64-bit data values evaluated using the respective rules for register and memory operands (mentioned above).

```
rule <k>
  exec(andnq memVal_64, BV_r, R:R64) ⇒ .
...</k>
  <regstate>
    "R"   ↦ _ ⇒ (~BV_r & MemVal_64)
    // sf and zf are updated based on the result.
    "SF"  ↦ (~BV_r & MemVal_64)[63:63]
    "ZF"  ↦ (~BV_r & MemVal_64) = 64'0 ? 1'1:1'0
    // of and cf are cleared.
    "OF"  ↦ 1'0
    "CF" ↦ 1'0
    // af and pf are undefined.
    "AF" ↦ undef
    "PF" ↦ undef
  ...</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 $n > i \geq j \geq 0$. 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. The reader is encouraged to take a look at the instruction semantics (made available as supplemental material [11]).

3.6 Constructing the x86-64 Semantics

Systematic Translation of Strata Rules to $\mathbb{K}$ As mentioned in the introduction, we leverage the Strata [30] 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 (~500 LOC).

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[6]. 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. Below are their pseudo-code semantics:

```
// Register Variant: movsd %xmm1 , %xmm0
S1. XMM0[63:0] ⟵ XMM1[63:0]
S2. XMM0[127:64] (Unmodified)

// Memory Variant: movsd (%rax) , %xmm0
S1. XMM0[63:0] ⟵ MEM_ADDR[63:0]
S2. XMM0[127:64] ⟵ 0
```

As seen, only the memory instruction clears the higher 64 bits of the destination register, which cannot be inferred from the register instruction behavior that does not touch the 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 (= 118×256)[7] formulae 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 some simplification rules 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.[8]

4 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.

4.1 Co-Simulations against Hardware

Empowered by the fact that we can directly execute the semantics using the $\mathbb{K}$ framework, we validated our model by

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

[7]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.

[8]Refer to our semantics [11], specifically instruction-semanatics/shrxl_r32_r32_r32.k, for details.

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. We performed all experiments on an Intel Xeon E5-2640 machine with 20 physical cores, running at 2.4 GHz. 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 [6] 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 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:

- In section 2.2, 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.
- While testing instructions implementing floating point operations, we found that many of the test inputs are representing a NaN or Infinity and it does not makes sense to test with many instances of these. We did our best effort by manually generating more than 100 unique floating point values by consulting the IEEE floating point arithmetic standard [1].
- We used the (~100) test-inputs offered by Remill [10].
- We manually implemented a regression test-suite worth of around 200 test-inputs which we accumulated over the course of the project.

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[9] 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 (e.g. `vfmadd132pd`).

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 suggest that it should depend on the lower 128 bits. Similar inconsistencies are found in instructions with mnemonics `vpsllvd`, `vpsllvq`, `vpsravd`.
- We found misleading typos related to instructions with opcodes `vpsravw`, `vpsravd`, `vpsravq`, `packsswb`.

All these items were reported and acknowledged by Intel as issues in the manual [11]

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

[9]According to the standard IEEE-754-2008 [1] (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.

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 [4].
- The simplification rule sub_double(A, B) == 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 [3] 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, like malloc, fprintf, most of whose semantics are modeled in $\mathbb{K}$. 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[10].

4.2 Comparing with Stoke

Stoke [44] 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:

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.)
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 through a pull request [11].

5 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 demonstrate 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. Each application can be leveraged into a standalone tool, with its own user interface, case studies and even reference manual, 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.

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

5.1 Validating Processor Hardware and Documentation

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 post-silicon validation, an additional challenge is the difficulty of debugging and diagnosing observed erroneous behaviors. For all these reasons, verification is estimated to use 70% of the resources and time, while design takes only 30% [23].

A fully executable formal ISA-level specification such as the one developed here can improve the state of practice in verification in two significant ways.

First, it can provide a reliable specification of 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. This is especially valuable in post-silicon validation, where observability within the chip is very limited, and functional validation is a key goal. This method can also help post-silicon failure diagnosis by identifying buggy input/output pairs and pinpointing specific erroneous output state bits.

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 [16, 35, 36], we are unaware of similar tools at the ISA level. The most significant advantage of such symbolic execution is the ability to detect corner case or hard to detect bugs [36]. (This is analogous to finding security vulnerabilities due to corner-case software bugs, illustrated in Section 5.3, 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 better design perspective 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 4. 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. However, given an existing 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, and much more sophisticated versions that synthesize illustrative examples and even explanatory text automatically could be possible soon, given recent advances in concolic test generation, program synthesis, and natural language processing.

```
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 2. sum-to-n program

5.2 Program Verification

The $\mathbb{K}$ framework provides a language-parametric, reachability logic theorem prover [40, 48]. 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 2. It takes N as input and returns the sum from 1 to N. The functional correctness can be essentially described as: $\%\text{rax} = \sum_1^N n = N(N+1)/2$. 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.

Figure 3(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 will have the expected return value. The

```
<regstate>...
    "RDI" ↦ 64'N
    "RBP" ↦ 64'56
    "RIP" ↦ (64'0 => 64'-1)
    "RAX" ↦ (64'_ => 64'(N * (N + 1)) / 2)
...</regstate>
<memstate>...
  // -8(%rbp): n
  48 ↦ (byte(0,_) => byte(0, 32'0))
  49 ↦ (byte(0,_) => byte(1, 32'0))
  50 ↦ (byte(0,_) => byte(2, 32'0))
  51 ↦ (byte(0,_) => byte(3, 32'0))
  // -4(%rbp): s
  52 ↦ (byte(0,_) => byte(0, (32'(N * (N + 1)) / 2)))
  53 ↦ (byte(0,_) => byte(1, (32'(N * (N + 1)) / 2)))
  54 ↦ (byte(0,_) => byte(2, (32'(N * (N + 1)) / 2)))
  55 ↦ (byte(0,_) => byte(3, (32'(N * (N + 1)) / 2)))
...</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, 32'0))
  49 ↦ (byte(1, A) => byte(1, 32'0))
  50 ↦ (byte(2, A) => byte(2, 32'0))
  51 ↦ (byte(3, A) => byte(3, 32'0))
  // -4(%rbp): s
  52 ↦ (byte(0, B) => byte(0, 32'(B + A * (A + 1) / 2)))
  53 ↦ (byte(1, B) => byte(1, 32'(B + A * (A + 1) / 2)))
  54 ↦ (byte(2, B) => byte(2, 32'(B + A * (A + 1) / 2)))
  55 ↦ (byte(3, B) => byte(3, 32'(B + A * (A + 1) / 2)))
...</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. Specification of sum-to-n program

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. The `requires` clause specifies the condition of N that prevents the arithmetic overflow. Figure 3(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 to verify the `sum-to-n` assembly code satisfies the functional correctness specification.

5.3 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

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

(a) C source code

```
  # Address %ebp - 8  contains 64-bit value 'a'
  # Adress  %ebp - 16 contains 64-bit value 'b'
  # Let a[32:0]: lower 32 bits of 'a'
  #     b[32:0]: lower 32 bits of 'b'

  movl -8(%ebp), %edx    # a[32:0] moved to %edx
  movl -16(%ebp), %eax   # b[32:0] moved to %eax
  addl %edx, %eax        # r = a[32:0] + b[32:0]
  movl $0, %edx          # check if (32'0 ∘ r) < a
  cmpl -8(%ebp), %eax    #  ∘: denotes concatenation
  movl %edx, %eax
  sbbl -4(%ebp), %eax
  jnc L2                 # true branch: "error state"
  ...                    # set *of to 1
  ...                    # set %eax to r
  jmp L3
L2:                      # else branch: "safe state"
  ...                    # set %eax to r
L3:
  ret
```

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

Figure 4. A security vulnerability in the HiStar kernel

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 [50] kernel, as shown in Figure 4(a), in which the KLEE [21] 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 the sizes of a and b are still 64-bit (`uint64_t`).[11] A suggested fix [21] 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 that, in the assembly code as shown in Figure 4(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:

$$\mathsf{a} + \mathsf{b} \geq 2^{32} \wedge (\mathsf{a} + \mathsf{b} \mod 2^{32}) \geq \mathsf{a}$$

[11]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.

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

Figure 5. popcnt program

where $0 \le a < 2^{64}$ and $0 \le b < 2^{64}$. 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.

5.4 Translation Validation of Optimizations

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 5 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,[12] 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 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). 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.

6 Limitations

Our limitations include the following missing features of the x86-64.

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

Project Name	Complete Support of x86-64 User-Level Instructions (in scope)	Executable Semantics	Support for Full-Fledged Formal Reasoning
Strata [30]	✗	◒	✗
Goel et. al. [26]	✗	✓	✓
CompCert [33]	✗	✓	✓
Remill [10]	✗	✓	✗
TSL [34]	✗	✓	◒
Our Semantics	✓	✓	✓

✓: Yes ✗: No ◒ : Partially True

Table 1. Projects hosting formal semantics of the x86-64 ISA.

- *Floating Point Operations*: Our testing shows that we have FP precision issues with instructions implementing the fused-multiply-add operation. We plan to fix it by including the FMA capabilities of GNU MPFR library [13] in 𝕂's library support [9].
- *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.
- *Concurrency*: Like the closest previous work [26, 30], we do not model concurrent semantics or the relaxed memory model. Other previous work has defined a formal, executable semantics for the x86 memory model [39, 43], and we leave it future work to merge that with our semantics. The design of our model is parameterized by the memory model and hence amenable to plug-in another one.

7 Related Work

There have been many projects that host a formal semantics of x86-64 either as their main contribution or as part of their infrastructure. Table 1 summarizes such previous work and compares it to our formal semantics.[13] We do the comparison in three directions that reflect the primary contributions of our work: the completeness of the definition in terms of supported user-level instructions, the faithfulness of the definition in terms of whether it is executable and hence can be evaluated with real code execution, and the generality of the definition in terms of its applicability to formal reasoning analyses. Next, we discuss in more detail each of the related works.

Strata [30] uses program synthesis to generate the instruction semantics, as SMT bit-vector formulas, by learning their input/output behavior through execution on an actual processor. Strata's semantics definition for x86-64 is incomplete as it does not include semantics of about 40% of the user-level instructions in scope that are not straightforward to automatically learn by the presented algorithm for reasons mainly due to limitations of underlying synthesis engine. Even then,

[13]Here we focus on comparing with other *direct* semantics, since a complete *direct* semantics is our goal and required for our purpose. We will discuss other *indirect* semantics later in this section.

the result that the formal semantics of 60% of the target x86-64 ISA can be learned automatically is impressive, and we leverage this result in our work as described in section 3. The specifications are executable only for non floating-point (FP) instructions. The FP operations are represented in the SMT formulas of the definition as uninterpreted functions. Finally the specifications are given as SMT formulas but have not been demonstrated to be usable in a formal analysis setting out-of-the-box.

Goel *et al.* [26] use the ACL2 theorem prover [31] to model the x86-64 ISA and has support for 33% of all user level instructions [15]. They have specifications of a selection of user-level instructions, some system-level instructions, paging, and segmentation. This list is far from a complete semantic definition of x86-64, however 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 in its use for simulations. In our work we do not leverage this definition, since we found a more complete definition such as Strata to be a better starting point towards completeness.

The CompCert verified compiler [33] includes semantics definitions for all intermediate and target languages used within the compiler, including a definition for x86 assembly. The definition is specified in Coq [12], and has been used in a formal setting for proving the correctness of the CompCert's compilation step to assembly, as well as outside CompCert, e.g., in proofs relating to the certified concurrent OS kernel CeriKOS [27]. 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.

Remill [42] is a static binary translator from x86-64 to LLVM IR [32]. The translator contains specifications for x86-64 instructions in the form of equivalent C programs to assist the translation. This specification is neither complete nor formal and cannot be easily used in formal analysis.

TSL [34] 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 for the generation of various tools, including a machine code synthesizer [47]. This definition, to the best of our knowledge, have not been used for formal specification proofs, i.e., to prove whether a given x86 program meets its specification.

Our semantics, like all of the aforementioned works, uses a sequential consistency memory model, omitting weaker memory models. Existing efforts to specify weaker memory models for x86-64 such as Owens et al. [39] and Sarkar et al. [43], however, suffer from their limited support for instruction semantics (i.e., they consider only a small subset of 32-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.

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 [2] using the VEX IR of Valgrind [38], the QEMU [19] emulator using the TCG IR, the software fault isolation tool RockSalt [37] using its own RTL DSL, the disassembler and binary analyzer Radare2 [17] using the ESIL IR [5], and the binary analysis tool BAP [20] using the BIL IR. We refer these semantics as *indirect* as they give the semantics of the x86-64 binary via the translation to their IR, as opposed to the *direct* semantics such as ours. Note that the direct semantics has its own value. 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.* [28, 29] also present an indirect semantics of x86-64, but in contract to other indirect semantics, they use machine learning [29] and symbolic execution [28] 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 [28], 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.

8 Conclusion and Future Work

We have presented the most complete formal semantics about x86-64 user-level instructions to date, which has been thoroughly tested and demonstrated to be applicable for different formal analyses such as symbolic execution, deductive verification, and translation validation. Moreover, the $\mathbb{K}$ framework enables us to represent a semantics as SMT theories, which other projects can leverage for their own purposes.

As mentioned in Section 3.2, we found our ideas of extending Strata do not scale because we need information about the instruction, whose semantics we want to learn, in order to constraint the search space. This lesson intrigued us to a new idea, where the information needed to reduce the search space can be automatically extracted from the manual. This information does not need to be precise, and we believe that such a rough information can be automatically extracted from the manual using text processing. We plan to explore this idea while defining the semantics of unspecified and/or new instructions.

References

[1] 2008. IEEE Std 754-2008 - IEEE Standard for Floating-Point Arithmetic. https://standards.ieee.org/findstds/standard/754-2008.html. https://doi.org/10.1109/IEEESTD.2008.4610935

[2] 2018. Angr: A powerful and user-friendly binary analysis platform! http://angr.io/. Last accessed.

[3] 2018. C Language Testsuites: C-torture version 8.1.0. https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html. Last accessed.

[4] 2018. Eric Schkufza. Personal communication.

[5] 2018. Evaluable Strings Intermediate Language. https://radare.gitbooks.io/radare2book/content/disassembling/esil.html. Last accessed.

[6] 2018. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. Last accessed.

[7] 2018. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm. Published on October 12, 2016, updated May 18, 2018.

[8] 2018. Matthew Fernandez. Personal communication.

[9] 2018. MPFR Java Bindings. https://github.com/kframework/mpfr-java. Last accessed.

[10] 2018. Remill: Library for lifting of x86, amd64, and aarch64 machine code to LLVM bitcode. https://github.com/trailofbits/remill. Last accessed.

[11] 2018. Supplemental Materials Submitted Along with Paper. Link to repository removed for double blind review. Last accessed.

[12] 2018. The Coq Proof Assistant. https://coq.inria.fr/. Last accessed.

[13] 2018. The GNU MPFR Library. https://www.mpfr.org/. Last accessed.

[14] 2018. x86 and amd64 Instruction Reference (UnOfficial). http://www.felixcloutier.com/x86/. Last accessed.

[15] 2018. 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. Last accessed.

[16] A. Ahmed, F. Farahmandi, and P. Mishra. 2018. Directed test generation using concolic testing on RTL models. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1538–1543. https://doi.org/10.23919/DATE.2018.8342260

[17] Sergi Alvarez. 2018. Radare2. https://rada.re/r/. Last accessed.

[18] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[19] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[20] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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 (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[22] Chucky Ellison and Grigore Roşu. 2012. 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, 533–544. https://doi.org/10.1145/2103656.2103719

[23] Harry D. Foster. 2015. Trends in Functional Verification: A 2014 Industry Study. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, 48:1–48:6. https://doi.org/10.1145/2744769.2744921

[24] Shilpi Goel and Warren A. Hunt. 2014. Automated Code Proofs on a Formal Model of the X86. In *Revised Selected Papers of the 5th International Conference on Verified Software: Theories, Tools, Experiments - Volume 8164 (VSTTE 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 222–241. https://doi.org/10.1007/978-3-642-54108-7_12

[25] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. 2017. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. Springer International Publishing, Cham, 173–209. https://doi.org/10.1007/978-3-319-48628-4_8

[26] 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

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

[28] Niranjan Hasabnis and R. Sekar. 2016. Extracting Instruction Semantics via Symbolic Execution of Code Generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 301–313. https://doi.org/10.1145/2950290.2950335

[29] Niranjan Hasabnis and R. Sekar. 2016. 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 (ASPLOS '16)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2872362.2872380

[30] 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

[31] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA.

[32] Chris Lattner and Vikram Adve. 2004. 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.

[33] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[34] Junghee Lim and Thomas Reps. 2013. TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 4 (April 2013), 59 pages. https://doi.org/10.1145/2450136.2450139

[35] L. Liu and S. Vasudevan. 2011. Efficient validation input generation in RTL by hybridized source code analysis. In *2011 Design, Automation Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763253

[36] Lingyi Liu and Shobha Vasudevan. 2014. Scaling Input Stimulus Generation Through Hybrid Static and Dynamic Analysis of RTL. *ACM Trans. Des. Autom. Electron. Syst.* 20, 1, Article 4 (Nov. 2014), 33 pages. https://doi.org/10.1145/2676549

[37] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. *PLDI: Programming Languages Design and Implementation* (2012), 395–404. https://doi.org/10.1145/2254064.2254111

[38] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 44 – 66. https://doi.org/10.1016/S1571-0661(04)81042-9 RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[39] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: X86-TSO. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[40] Grigore Roşu and Andrei Ştefănescu. 2012. Checking Reachability using Matching Logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 555–574. https://doi.org/citation.cfm?doid=2384616.2384656

[41] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[42] Andrew Ruef and Artem Dinaburg. 2014. Static Translation of X86 Instruction Semantics to LLVM with McSema. *REcon* (2014). https://github.com/trailofbits/mcsema

[43] 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 *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 379–391. https://doi.org/10.1145/1480881.1480929

[44] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[45] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. [n. d.]. *The K Primer (version 3.2)*. Technical Report.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

[47] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of machine code from semantics. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015* (2015), 596–607. https://doi.org/10.1145/2737924.2737960

[48] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 74–91. https://doi.org/10.1145/2983990.2984027

[49] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210

[50] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=1267308.1267327