Stratified Synthesis: Automatically Learning the x86-64 Instruction Set

Stefan Heule
Stanford University, USA
sheule@cs.stanford.edu

Eric Schkufza
VMware, USA
eschkufza@vmware.com

Rahul Sharma
Stanford University, USA
sharmar@cs.stanford.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

Abstract

The x86-64 ISA sits at the bottom of the software stack of most desktop and server software. Because of its importance, many software analysis and verification tools depend, either explicitly or implicitly, on correct modeling of the semantics of x86-64 instructions. However, formal semantics for the x86-64 ISA are difficult to obtain and often written manually through great effort. We describe an automatically synthesized formal semantics of the input/output behavior for a large fraction of the x86-64 Haswell ISA's many thousands of instruction variants. The key to our results is *stratified synthesis*, where we use a set of instructions whose semantics are known to synthesize the semantics of additional instructions whose semantics are unknown. As the set of formally described instructions increases, the synthesis vocabulary expands, making it possible to synthesize the semantics of increasingly complex instructions.

Using this technique we automatically synthesized formal semantics for 1,795 instruction variants of the x86-64 Haswell ISA. We evaluate the learned semantics against manually written semantics (where available) and find that they are formally equivalent with the exception of 50 instructions, where the manually written semantics contain an error. We further find the learned formulas to be largely as precise as manually written ones and of similar size.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM.

PLDI '16 June 13–17, 2016, Santa Barbara, CA, USA
Copyright © 2016 ACM 978-1-4503-4261-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2908080.2908121

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques

Keywords ISA specification, program synthesis, x86-64

1. Introduction

The various x86 instruction sets have been ubiquitous for decades, and currently the Haswell family of processors is widely used for server and desktop machines. Because of the importance of x86, many software analysis tools have been built that reason about programs written in x86 directly (e.g., [2, 9, 24]) or that reason about programs in high-level languages that are then translated to x86 (e.g., [17]). All such tools require a precise formal semantics of the x86 ISA. One might assume, then, that a formal semantics of the current 64-bit x86 ISA (known as x86-64) would be readily available, but one would be wrong. In fact, the only published description of the x86-64 ISA is the Intel manual [14] with over 3,800 pages written in an ad-hoc combination of English and pseudo-code. There are two primary barriers to producing a formal specification of x86-64. First, the instruction set is huge: x86-64 contains 981 unique mnemonics and a total of 3,684 instruction variants. Second, many of the instructions have complex semantics. For example, the rotate instruction implicitly truncates its integer argument, modifies an implicit register operand, and depending on its input either changes the value of multiple condition registers or leaves them undefined. Many instructions have comparable edge cases and variations, and the behaviors are often inconsistent with other, related instructions.

For those who need a formal semantics of x86-64, a common though necessarily inefficient approach is to generate formal specifications by hand on demand: whenever progress requires the semantics of an instruction i, someone writes a formula for i. Many software research groups have taken this approach, and ours is no exception. However, our personal

experience is that it is nearly impossible to write and maintain correct specifications solely by hand—x86-64 is just too big and complicated. As a result, we gradually built an elaborate infrastructure to support developing a formal semantics, including templates to describe the semantics of families of closely related instructions and testing tools to compare the formal descriptions with the behavior of the actual hardware. Wherever we added automation, we found additional bugs in the human written specifications. Eventually we came to ask whether we could eliminate the problematic human effort entirely and generate a useful formal description of most of x86-64 from a small "proto specification".

The most successful previous attempt to automatically generate a formal semantics for x86-64 is a template-based method for explaining input-output examples that are derived from hardware executions. This approach has been used to generate formulas for 534 instruction variants [12]. These variants cover the core of 32-bit x86, but do not include any of the more recent extensions of x86-64 (e.g., vector instructions). The templates are written by hand and must satisfy certain structural restrictions. It seems unlikely that these requirements are expressive enough to cover the remainder of the instruction set.

In this paper we demonstrate a new technique that can automatically synthesize a formal semantics for a large fraction of the x86-64 ISA. Our approach uses program synthesis to learn bit-vector formulas that can be consumed by SMT solvers such as Z3 [32] or CVC4 [6]. We begin by choosing a small set of x86-64 instructions that we call the *base set*. Our approach expects a formal semantics for every instruction in the base set as input and then generates formulas for the remaining instructions automatically. The effort required to produce the formulas for the base set is small—they can either be written by hand or generated using some other approach (including [12]).

At a high level, our approach works as follows. We execute an instruction i for which the formal semantics is not known yet on a set of test inputs T to obtain an initial description of its behavior. We then search for a program p that matches the behavior of i on the tests T, where p only uses instructions drawn from the base set S. However, there is little reason to have confidence that one program that happens to match an instruction's behavior on a set of test cases actually captures the behavior of that instruction for all possible inputs, so we perform the search multiple times to find a set of programs P that match the behavior of i on T and use only instructions from S. Given two programs $p, p' \in P$, we test whether $p \equiv p'$ using an SMT solver and the formulas from the base set. If the two programs are semantically distinct (meaning the agreement on T is coincidental), we know that one or both programs are not a correct description of i. We use the model produced by the SMT solver to obtain an input t that distinguishes p and p', add t to the set of tests T, and start over. We repeat this process until we are unable to generate another program not in P that agrees with i on T (which includes all counterexamples discovered along the way). When we are done, we choose one $\hat{p} \in P$ and return the formula for $\hat{p}$ as the semantics of i.

Even given that we search for multiple programs until no further inconsistencies can be discovered, there is no guarantee that the program $\hat{p}$ correctly implements i. This uncertainty is unavoidable: we do not start with a formal specification of i's behavior, and so the best we can possibly do is to produce a semantics that is consistent with whatever information is available. However, we have carefully compared our automatically generated semantics to previously developed formulas for x86-64 and found a number of discrepancies. In every case, the automatically generated semantics was correct and the hand-written semantics was wrong. Furthermore, as a result of this effort we discovered a number of new bugs and inconsistencies in the Intel manual (see Section 5).

Because we want to automate the production of the formal semantics as much as possible, we must minimize the size of the base set of instructions that is the input to the process. However, many of the instructions that we consider can only be characterized by programs that are so long that no currently available program synthesis technique can produce them if they are written using only instructions in the initial base set. To address this problem, we introduce the idea of *stratified synthesis*: whenever we learn a program p for an instruction i, we add the formula for p to the base set as the semantics of i. Thus, as we learn the semantics of simpler instructions, our vocabulary for expressing the semantics of more complex instructions expands. We have implemented the approach in a tool called STRATA and we empirically observe that the use of stratified synthesis is both effective and necessary—there are many instructions that are learned only after other, intermediate, instructions have been learned.

This paper makes the following contributions:

- We introduce a stratified program synthesis technique that enables us to learn complex programs in small steps and show how it can be used to synthesize a formal semantics for the x86-64 ISA.
- We contribute SMT formulas for a subset of the x86-64 ISA that is significantly larger than previous automatically synthesized semantics [1].
- We evaluate these formulas for correctness and usability; in doing so, we identify several important errors in pre-existing formalizations.

2. Modeling x86-64

In this paper, we focus on the functional aspects of the x86-64 instruction set. That is, we search for bit-vector formulas that give a precise description for the input-output behavior of instructions on registers and memory. The details of the x86-64 memory model, such as how memory behaves under concurrency or alignment requirements are important but

[1] Available at https://stefanheule.com/strata/.

orthogonal to that goal and are not inferred by our approach. Nonetheless, we consider the majority Haswell instruction set, with extensions such as AVX and AVX2.

2.1 Modeling the CPU State

We model the CPU state as bit-vectors that correspond to registers and an array that represents a byte-addressable memory. We restrict our register model to the registers described below. The x86-64 architecture has additional special purpose registers and flags used by system-level instructions, which we do not attempt to model or learn (see Section 2.2).

- **General Purpose Registers:** The 16 64-bit registers: `rax`, `rcx`, ..., `r15`; the lower 32, 16, and 8 bits of those registers: `eax`, `ax` and `al`, etc.; and the legacy registers that name bits 8 through 15: `ah`, `ch`, `dh`, `bh`.
- **Vector Registers:** The 16 256-bit registers: `ymm0`, ..., `ymm15`; and the lower 128 bits of those registers: `xmm0`, ..., `xmm15`.
- **Status Flags:** Five bits from the `rflags` register. These store partial results from many arithmetic operations and are used by conditional jump instructions: `cf` (carry), `pf` (parity), `zf` (zero), `sf` (sign), and `of` (overflow).

2.2 Instructions in Scope

We exclude a few classes of instructions because they are very difficult to model, rarely used, or both. Of the entire 3,684 instruction variants that make up the x86-64 Haswell ISA, these exclusions leave us with 2,918 variants whose specifications we can potentially infer. The excluded instructions are:

- **Systems-level** (302 variants): These instructions are rarely used by application-level code and would require a more detailed model of the operating system, protection levels, and other low-level details. Examples are `hlt` (stopping execution), `syscall` (system call) and `invpcid` (invalidate entries in the TLB).
- **Cryptography** (35 variants): These instructions support AES encryption. An example is `aeskeygenassist` (AES round key generation).
- **x87** (155 variants): These are floating-point instructions introduced in 1980 and deprecated by SSE (1999). The newer instructions are faster, more versatile, and vectorized.
- **MMX instructions** (177 variants): These are vector instructions introduced in 1997 and deprecated by SSE (1999). These instructions suffer from limitations such as the inability to interleave integer and floating-point operations.
- **String instructions** (97 variants): Currently unsupported by the synthesizer we use.

2.3 Dataflow Information

Our approach to synthesizing formulas requires information about which locations an instruction reads from and writes to with respect to both registers and memory. In most cases, these sets are given by an instruction's operands. However, some instructions implicitly read or write additional locations. For example, `mulq rcx` multiplies `rcx` (the explicit operand) by `rax` (implicit operand) and stores the result in `rax` and `rdx` (both also implicit). Certain x86-64 instructions may also place undefined values in register locations. Because these values are free to vary between implementations ([12] showed that different CPUs actually do exhibit different behavior), we require that these locations be specified as well, so as not to overfit to the CPU that we run our experiments on. We neither attempt to learn the output relation for undefined values that are placed in output registers, nor allow instructions to read from an undefined location. Although this information is already provided by the x86-64 specification in sufficient detail, we note that it would have been possible to automatically infer the majority of it as well.

3. Approach

In this section we describe our approach, starting with a simple example that shows the main steps, followed by an explanation of the details. The full algorithm in pseudocode is shown in Algorithm 1 and Algorithm 2.

3.1 Overview

We start with a *base set* of instructions, for which we already have formal specifications. This set is meant to be small but should cover all of the unique functionality in the instruction set. For all remaining instructions, we automatically infer a formula for an instruction i by learning a small x86-64 program, consisting only of instructions whose semantics are already known, that behaves identically to i. For instance, consider the instruction variant `decb` (the decrement instruction that operates on an 8 bit register). Since the only operand of `decb` is a register, we can instantiate the instruction (InstantiateInstr in Algorithm 1) to obtain a one instruction program by choosing an input register; e.g., `decb bl`. We call this the *target instruction* t, and attempt to synthesize a loop-free program p_0 that behaves just like t on a number of test cases (more on test case generation in Section 3.3) using a stochastic search. For the stochastic search Synthesize we use STOKE [26] for which we give a description in Section 4.

For instance, for the decrement instruction we might learn this program:

```
1  xorb al, al # al = 0 and clears cf
2  setae ah    # set ah if cf is 0
3  subb ah, bl # bl = bl - ah
```

Algorithm 1 Main algorithm

```
Input: base set baseset, a set of instruction/formula pairs,
    and a set worklist of instructions
Result: a set of instruction/formula pairs
 1: procedure STRATA(worklist, baseset)
 2:   tests ← GENERATETESTCASES()
 3:   while |worklist| > 0 do
 4:     instr ← CHOOSE(worklist)
 5:     worklist ← worklist − {instr}
 6:     t ← STRATAONE(instr, baseset, tests)
 7:     if t is failure then
          # we couldn't learn a program for instr yet
 8:       worklist ← worklist ∪ {instr}
 9:     else
10:       baseset ← baseset ∪ {⟨instr, t.formula⟩}
11:   result ← baseset
12:   for (instr, formula) ← baseset do
13:     result ← result ∪ GENERALIZE(instr, formula)
14:   return baseset

    # learn a formula for instr
15: procedure STRATAONE(instr, baseset, tests)
16:   prog_instr ← INSTANTIATEINSTR(instr)
17:   t = SYNTHESIZE(prog_instr, baseset, tests)
18:   if t is timeout then return ⟨failure⟩
      # the set of equivalence classes of learned programs
19:   eqclasses ← {{t.prog}}
20:   while true do
21:     t = SYNTHESIZE(prog_instr, baseset, tests)
22:     if t is timeout then
23:       break
24:     else
25:       t ← CLASSIFY(t.prog, eqclasses, tests)
26:       if t is failure then return ⟨failure⟩
27:     if |eqclasses| > threshold then
28:       break
29:   bestclass ← CHOOSECLASS(eqclasses)
30:   bestprog ← CHOOSEPROG(bestclass)
31:   formula ← BUILDFORMULA(instr, bestprog)
32:   return ⟨success, formula⟩
```

Algorithm 2 Helper functions

```
    # classify prog into the equivalence classes
 1: procedure CLASSIFY(prog, eqclasses, tests)
 2:   eqs ← ∅
 3:   for class ← eqclasses do
 4:     t ← SMTSOLVER(prog, CHOOSEPROG(class))
 5:     if t is equivalent then
 6:       eqs ← eqs ∪ {class}
 7:     else if t is counterexample then
 8:       tests ← tests{t.counterexample}
 9:       remove all programs p in eqclasses for which
            RUN(p) ≠ RUN(prog_instr)
10:       if |eqclasses| = 0 then return ⟨failure⟩
11:   if |eqs| = 0 then
12:     eqclasses ← eqclasses ∪ {prog}
13:   else
14:     eqclasses ← merge all classes in eqs
15:     add prog to merged class
16:   return ⟨success⟩

    # generalize a learned formula for an instruction to a set
    of formulas
17: procedure GENERALIZE(instr, formula)
18:   result ← ∅
19:   for instr′ ← ALLINSTRUCTIONS() do
20:     if instr′ has same mnemonic as instr then
21:       continue
22:     candidate ← NONE
23:     if operands of instr′ have same size then
24:       candidate ← rename operands in formula
25:     if operands of instr′ have smaller size then
26:       candidate ← rename and sign-extend operands
27:     if operands of instr′ have larger size then
28:       candidate ← rename and select operands
29:     if candidate ≠ NONE then
30:       test candidate with random constant operands
              and on random test cases
31:       add to result if all tests pass
32:   return result
```

The comments indicate the main steps of the program, but note that the instructions not only decrement the `bl` register, but also sets a number of status flags appropriately.

The search for a program has the freedom to overwrite registers that are not written by the target instruction; the resulting program p_0 must only agree with t on the locations that t may write. For example, the program above overwrites `al`, `ah` as well as `cf`, even though the decrement instruction does not affect these locations. Requiring that the synthesized program agree with the target instruction only on the target's write set effectively gives the search temporary scratch locations to use.

Once we have learned a program p, we can convert it to a formula by symbolic execution of p since we have formulas for all the instructions in p. However, the program p may overfit to the test cases, and there might be inputs for which p and t disagree. We run the search again to find a potentially different program p' that also behaves just like t on all test cases. Now, we use an SMT solver to check $p \equiv p'$.

If $p \not\equiv p'$, we get a counterexample, that is, an input for which p and p' behave differently. We can then remove the incorrect program (or possibly both programs) by comparing

their behavior on this new input against t. We also add the new test case to our set of test cases and search again. Otherwise, if $p \equiv p'$, we add p' to the set of programs we know. We repeat this process until we can either not find any more programs, or until we have enough programs according to a threshold. We discuss how to deal with solver timeouts and spurious counterexamples in Section 3.7.

This process increases our confidence that the learned programs in fact agree with t on all inputs, and allows us to find tricky corner-case inputs. High confidence is the most we can hope to achieve because, again, our purpose is to infer specifications for which there is in general no ground truth. In Section 5 we evaluate the correctness of the learned semantics for those instructions where we happen to have hand-written formulas.

Once this processes finishes, we are left with a collection of programs from which we can choose one (details in Section 3.8) to use as the semantics of t. At this point, we can add this instruction variant to the base set, and use it in further searches for other instructions. This creates a *stratified* search, where simple instructions are learned first, followed by slightly more complex instructions that use the simpler instructions learned previously, and so on, until finally quite complex instructions are learned.

3.2 Base Set

The base set consists of 51 instruction variants that cover the fundamental operations of the x86-64 instruction set:

- **Integer addition** (4 instruction variants).
- **Bitwise operations**, including bitwise or and exclusive or, shifts (both arithmetic and logical) as well as population count (6 instruction variants).
- **Data movement** from one register to another, as well as variants to sign-extend values (7 instruction variants).
- **Conditional move** (1 instruction variant).
- **Conversion operations** between integers and floating-point values for both 32- and 64-bit data types (8 instruction variants).
- **Floating point operations** including addition, subtraction minimum, maximum, division, approximate reciprocal and square root, fused multiply and add/subtract (which is not the same as a multiply followed by an addition due to the higher internal precision of the fused operation), for both single and double precision floating-point values where available, all vectorized (24 instruction variants).
- **Clear registers**. We include `vzeroall`, which clears all vector registers. This is included for technical reasons only: STOKE requires the initial program (conceptually the empty program) to define all output locations, and this instruction allows this easily for the vector registers (1 instruction variant).

Precisely modeling floating-point instructions over bit-vectors in a way that performs well with SMT solvers is challenging [10]. For this reason, we model the floating-point instructions in the base set as uninterpreted functions. This encoding is extremely imprecise, but still provides a useful semantics for many instructions. For example, it is sufficient for inferring that vectorized floating-point addition consists of several floating-point additions over different parts of the vector arguments. The main limitation of using uninterpreted functions is that it complicates equivalence checking. Given two programs that potentially represent the semantics of an unmodeled instruction, the appearance of uninterpreted functions substantially reduces the likelihood that we will be able to show those two programs to be equivalent.

In addition to these instructions, we found that certain data movements are not well supported by any instruction in the x86-64 instruction set. For instance, there is no instruction to set or clear the overflow flag `of` (there is an instruction that can set any of the other four flags), and in fact several instructions are needed just to set this single flag. Similarly, updating, say, the upper 32 bits of a 64-bit general purpose register in a non-destructive manner is surprisingly difficult and again requires several instructions. Many instructions include setting specific flags or parts of registers as part of their functionality, and the fact that the x86-64 instruction set is missing some of these primitives makes implementing such instructions in x86-64 more difficult than necessary. For this reason, we augment the base set with *pseudo instructions* that provide this missing functionality. Of course, like other base set instructions, we provide formulas for the pseudo instructions.

We implement the pseudo instructions as procedures (that could be inlined). Unlike regular instructions, pseudo instructions cannot be parameterized by operands, since assembly-level procedure calls take arguments in predefined locations according to a calling convention. We address this issue by having a template for every pseudo instruction that can be instantiated with several specific register operands. For instance, we might have a template to set a flag that is instantiated separately for all six status flags. We add the following pseudo instruction templates:

- **Split and combine registers**. Move the value of a $2n$-bit register into two n-bit registers (upper and lower half), as well as the inverse operation (2 templates, 108 instantiations).
- **More splitting and combining**. For the 128-bit registers, we also allow them to be split into four 32-bit registers, and vice versa (2 templates, 18 instantiations).
- **Moving a single byte**. Move the value in a 1-byte register to a specific byte in an n-byte register, and vice versa (2 templates, 152 instantiations).
- **Move status flag** to a register and back. Move the bit stored in a status flag to the least significant bit of a register

by zero-extending its value, and move the least significant bit of a general purpose register to a specific status flag (2 templates, 22 instantiations).

- **Set and clear status flags** (2 templates, 12 instantiations).
- **Set `sf`, `zf` and `pf` according to result**. The sign, zero and parity flags have relatively consistent meaning across many instructions, and this instruction sets them according to the value in a given general purpose register (the sign flag is the most significant bit, the zero flag is 1 if and only if the value is zero, and the parity flag indicates if the least significant byte has an even or odd number of set bits in the 8 least significant bits). (1 template, 4 instantiations).

This makes for a total of 11 pseudo instructions with 316 instantiations. All of these have trivial semantics, as they are mostly concerned with data movement.

3.3 Test cases

A test case is a CPU state, i.e., values for all registers, flags and memory. When beginning a synthesis task we initially generate 1024 random test cases. Additionally, we also pick heuristically interesting bit patterns such as 0, -1 (all bits set) and populate different register combinations with these values. Similarly, we pick some special floating-point values, such as NaN (not a number), infinity, or the smallest non-zero value. With 22 heuristically interesting values we generate an additional 5556 test cases for a total of 6580. In Algorithm 1, this step is called GENERATETESTCASES.

3.4 Generalize Learned Programs

So far we have a method for learning a formula for an instruction variant that uses a specific set of registers. We need to generalize these formulas to other registers, and to instructions that take constants (so-called immediates) and memory locations as operands.

3.4.1 Generalization to Other Registers

If we have learned a formula for a particular set of register operands, then we can generalize this to other registers by simple renaming. For instance, after learning a formula for `decb bl`, we can create a formula for `decb r8` by computing the formula for `decb bl`. This determines the value for all registers that are written to, including `bl`. For all such outputs, we rename all occurrences of `bl` to `r8`. In our example, we might have learned the formula

$$\begin{aligned} \texttt{bl} &\leftarrow \texttt{bl} - 1_8 \\ \texttt{zf} &\leftarrow (\texttt{bl} - 1_8) = 0_8 \end{aligned}$$

(only showing the zero status flag for brevity; the two updates are done in parallel, so that `bl` on the right-hand side refers to the previous value in both cases). Here, $-$ is subtraction on bit-vectors, $=$ equality on bit-vectors, and c_w is the constant bit-vector of width w with value c. This can easily be renamed to obtain the fomula for `decb r8`:

$$\begin{aligned} \texttt{r8} &\leftarrow \texttt{r8} - 1_8 \\ \texttt{zf} &\leftarrow (\texttt{r8} - 1_8) = 0_8 \end{aligned}$$

Some instructions read or write implicit locations as well as take explicit operands. For the renaming to be unambiguous, we should learn a formula for an instruction where the implicit and explicit operands aren't referring to the same register. This can be done by ensuring that INSTANTIATEINSTR selects the explicit operands to be distinct from the any implicit ones an instruction might have.

This procedure makes two assumptions: (1) that using different registers as operands cannot make the instruction behave differently, and (2) the renaming is unambiguous (after taking care of implicit operands appropriately).

We can validate assumption (1) by running the instruction with different operands and validating that it behaves as we expect. If it does not, we can learn a separate formula for the operands that behave differently. Interestingly, we have found exactly one case where an instruction has different behavior depending on the register arguments, namely `xchgl eax, eax`. Normally, `xchgl` exchanges two 32-bit registers and clears the upper 32 bits of both corresponding 64-bit registers. However, `xchgl eax, eax` leaves everything unchanged[2].

Assumption (2) is actually not true, and the renaming might be ambiguous. For instance, the instruction `xaddq rax, rcx` exchanges the two registers and stores the sum in `rcx`. That is, the formula for this instruction is (ignoring status flags for simplicity):

$$\begin{aligned} \texttt{rax} &\leftarrow \texttt{rcx} \\ \texttt{rcx} &\leftarrow \texttt{rcx} + \texttt{rax} \end{aligned}$$

where $+$ is the bit-vector addition on 64-bit values. Now, if we want to get the formula for `xaddq rdx, rdx` then we have two possible values for updating the register `rdx`, as both locations that `xaddq` writes to are identical here. Fortunately, this situation only applies to instructions where two or more locations are written that could possibly alias (considering both explicit operands as well as implicit locations), and there are only five mnemonics that write two or more locations: `cmpxchg`, `xchg`, `mulxl`, `vinsertps`, and `xaddw`. We manually decide the correct value for these cases by consulting the Intel manual as well as testing the instruction on sample inputs. For the first two it does not matter as both possible values are the same, and for the remaining three it is easy to determine the correct value. For instance, the `xaddq` instruction writes the sum, and so `xaddq rdx, rdx` would store

$$\texttt{rdx} + \texttt{rdx}$$

in `rdx`. This generalization is done in BUILDFORMULA in Algorithm 1.

[2] To be precise, there are several possible encodings for the instruction

3.4.2 Generalizing to Memory Operands

The vast majority of instruction variants that can take a memory operand have a corresponding variant that takes only register operands. We can validate that the two instructions behave identically (except for the operand location) on random test cases, and then use the formula we have learned for the register variant.

For several floating-point operations, the memory location used has a different size. For instance, consider a floating-point addition on two 32-bit values where one of the values is stored in memory: `addss xmm0, (rax)`. The corresponding register-only variant operates on two 128-bit registers (because all `xmm` registers where floating-point values are stored are 128 bits). We can still generalize to these instructions, but only considering the lowest 32 bits from the register variant. Again, we validate this step by testing on random inputs.

While in principle this generalization could also be done in BUILDFORMULA, we do this as a post-processing step after we have learned all register-only formulas, namely in GENERALIZE.

3.4.3 Generalizing to Immediate Operands

Similar to memory operands, many instructions with immediate operands have a corresponding instruction that takes only registers. Again we can validate the hypothesis that they behave identically (other than where the inputs come from) and use the same formula. Some instructions do not have a directly corresponding instruction, but first require the constant to be extended to a higher bit width. We might hypothesize that the constant needs to be either sign-extended or zero-extended, but otherwise can use a formula already learned. We test this hypothesis on random inputs when we generalize instructions with immediate operands in GENERALIZE. We find that sign-extending the constants is the correct decision for all such instructions (even ones where one might expect a zero-extension, such as bit-wise instructions).

However, other instructions with immediate operands do not have a corresponding register-only instruction. Unfortunately, we cannot easily apply the same trick we used for registers and learn a formula directly for such variants. The problem is we would need to convert the instruction into a program in INSTANTIATEINSTR and therefore instantiate all operands. This means that we cannot vary the value of the immediate, and thus can only learn a formula for a particular constant. However, for many cases this turns out to be enough, as we can just learn a separate formula for every possible value of the constant. If the constant only has 8 bits, then brute force enumeration of all 256 possible values is feasible. This approach works well, as many instructions with 8 bit constants actually are a family of instructions and the constant controls which member of the family is desired. For instance, the `pshufd` instruction rearranges the four 32-bit values in a 128-bit register according to control bits in an 8-bit immediate argument. Not only is it possible to learn 256 different formulas for such instructions it is also considerably simpler to only learn a formula for a single control constant rather than one that works for all possible constants.

3.5 Preventing Formula Blowup

Due to the stratified search, where formulas for entire programs are reused in the semantics of newly learned instructions, our formulas have the potential to grow in size and the formulas that are learned later in the process might become very large. We found that a small number of simplifications reduce the size of the synthesized formulas dramatically and cause our learned formulas to be manageable in size. We preform two types of simplifications:

- Constant propagation.
- Move bit selection over bit-wise operations and across concatenation. For instance, for a 32 bit variable `eax`, we can simplify $(\texttt{eax} \circ \texttt{eax})[31:0]$ to `eax` (where $\circ$ is the concatenation operator for bit-vectors). This transformation can sometimes directly simplify the formula, or enable further simplifications in subexpressions.

3.6 Finding Precise Formulas

We strongly prefer precise formulas that do not involve uninterpreted functions. To ensure we learn a precise formula whenever possible, we first attempt to synthesize any formula. If the formula is imprecise (i.e., involves an uninterpreted function), then we additionally perform another search where we restrict the set of available instructions to only ones that have a precise formula. If the search succeeds, we categorize the program as usual and keep learning more precise programs until no more can be found or a threshold is reached. If no program using only this limited set of instructions can be found, then we accept the imprecise formula. For brevity this additional step is not shown in Algorithm 1.

3.7 Unknown Solver Answers

An SMT solver (SMTSOLVER in the pseudo code) does not always answer ”equivalent” or ”counterexample” when asked whether two formulas are equivalent. It is possible for the solver to time out, or to report that the two programs might not be equivalent, without giving a valid counterexample. The latter can happen with imprecise formulas that make use of uninterpreted functions. In that case, the counterexample provided by the SMT solver might be incorrect: the two programs in question may actually behave identically on that input.

Since we do not have precise formulas for most floating-point operations, we must deal with the fact that we cannot always prove equivalence or inequivalence of programs, and we cannot always obtain useful counterexamples. Instead of

---

`xchgl eax, eax`, and only the two variants that hard-code one of the operands to be `eax` are affected. These essentially encode to `nop`.

learning a set of programs for which we know that all programs are equivalent, we instead learn a set of equivalence classes $c_1, \ldots, c_n$ (called *eqclasses* in STRATAONE of Algorithm 1). All programs in a given class are formally equivalent (as proven by the SMT solver). Now, if we learn a new program p, we can try to prove it equivalent to the programs in all of equivalence classes and either add it to an existing class, merge classes if it happens that p is equivalent to programs in currently distinct equivalence classes, or create a new class for just p (done in CLASSIFY in Algorithm 2).

Thus, at the end of our search, we have a set of equivalence classes to choose from. If the formulas are precise, then there will typically be only a single class (unless timeouts are encountered, e.g., due to non-linear arithmetic).

3.8 Choosing a Program

Once we can no longer find new programs (or we have decided we have found sufficiently many programs), we must pick one program as the representative we use to create a formula. If there is only a single equivalence class, then all programs in it are equivalent and in theory it doesn't matter which program we pick. However, some formulas might be more desirable than others, e.g., if they are simpler or use no non-linear arithmetic. For this reason, we heuristically rank all formulas (that correspond to the synthesized programs) in order of the following criteria.

- Number of uninterpreted functions
- Number of non-linear arithmetic operations
- Number of nodes in the AST of the formula

This means we prefer precise formulas (ones that do not contain uninterpreted functions), formulas that perform well in SMT queries (no non-linear arithmetic that is prone to timeouts) and are simple (small formulas). This step is done in CHOOSEPROG in Algorithm 1.

Since we have no way to further distinguish between equivalence classes, we consider all classes that are above a threshold size (to avoid unreproducible outcomes) and pick the best one according to the ranking heuristic (CHOOSECLASS above).

4. Program Synthesis using STOKE

We use STOKE for the program synthesis step. Although we us it as a black box we provide a high-level description of how STOKE works here for completeness.

STOKE [26] is a stochastic search tool that synthesizes (or optimizes) x86-64 code from examples. Given a set of test cases $H = \{(h_1, o_1), \ldots, (h_n, o_n)\}$ consisting of pairs of inputs/outputs, and a subset χ of x86-64 instructions, STOKE synthesizes a program p that uses these instructions and agrees with the test cases, i.e., $\forall j . p(h_j) = o_j$. To synthesize a formula corresponding to an instruction i, we set χ to the instructions in the base set for which the formulas are already known. The examples are generated by executing a program containing only the instruction i on input CPU states (Section 3.3) and recording the output states.

STOKE starts from an empty program p and makes repeated randomly selected transformations to p. The transformations include inserting a randomly chosen instruction, deleting a randomly chosen instruction, swapping two randomly chosen instructions, etc. The transformations applied by STOKE are guided by a *cost function* defined on the test cases. The cost function penalizes programs that are "further" from being equivalent to i on the tests. For instance, a candidate cost function is the sum of Hamming distances between $p(i_j)$ and o_j. STOKE searches for programs that minimize cost by transforming the current program p to a new program p'. If the cost of p' is lower than p, then p' becomes the current program and the process is repeated again. If instead the cost of p' is greater than the cost of p, then with some probability—exponentially decaying with the difference of cost between p and p'—STOKE still updates the current program to p'. STOKE terminates when it finds a program that satisfies all the test cases in H.

Due to its stochastic nature and the large size of the x86-64 ISA, multiple runs of STOKE usually lead to distinct synthesized programs. This property is useful for synthesizing the multiple programs required by our approach. Nonetheless, STOKE allows for user-defined cost functions and we use this facility to avoid finding a program we have already learned before again. To this end, we add a term to the regular cost function that is 1 if the current program has been learned already, and 0 otherwise.

5. Experiments

In this section we evaluate our implementation STRATA according to the following research questions.

(RQ1) What number of instructions can STRATA automatically synthesize a formula for?
(RQ2) Are the synthesized formulas correct?
(RQ3) How large are the synthesized formulas compared manually written ones?
(RQ4) How precise and usable are the synthesized formulas compared to the hand-written ones?
(RQ5) Is the simplification step effective?
(RQ6) Is the stratification of the synthesis necessary?

We also give some details on the overall synthesis process, such as how long it takes and what fraction of the SMT solver calls returned unsatisfiable, a counterexample or timed out.

5.1 Synthesis Results

Table 1 summarizes the number of formulas that our approach can learn automatically. Starting from 51 base instructions and 11 pseudo instruction templates, both with formulas written by hand, we first learn 692 instruction variants that only use registers as operands. These can then be generalized to other instruction variants that use memory or immediate

Base set	51
Pseudo instruction templates	11
Register-only variants learned	692
Generalized (same-sized operands)	796
Generalized (extending operands)	81
Generalized (shrinking operands)	107
8-bit constant instructions learned	119.42
Learned formulas in total	**1795.42**

Table 1. Overall synthesis results. The table lists the number of base instructions, and how many formulas can be automatically learned from these, broken up into different categories.

operands. More precisely, we can generalize to another 796 instruction variants that use operands of the same size as the register-only variants. 81 variants require an operand to be sign-extended and another 107 variants are instructions that use smaller operands (a 32- or 64-bit memory location instead of a 128-bit `xmm` register). Finally, we also learned formulas for instructions with an 8-bit constant that do not have a corresponding register-only instruction. We attempted to learn a separate formula for all 256 different possible values for the constant. In some cases, we learned a formula only for some of the possible constants, which is why we count every formula as 1/256 of an instruction variant. We learn 119.42 instruction variants in this way. In total, STRATA learned 1,795.42 formulas automatically, or 61.5% of the instructions in scope.

5.2 Limitations

We manually inspected the instructions for which STRATA was unable to learn a formula. They can be roughly categorized as follows, though there might be some overlap between categories:

- **Missing base set instructions.** Some instructions perform floating-point operations that do not appear in the base set. The same is true of several vectorized integer instructions. For these instructions, there is no non-vectorized instruction that performs the corresponding primitive operation (e.g., saturated addition). A possible solution would be to add pseudo instructions that perform these operations in a non-vectorized fashion.
- **Program synthesis limits.** Some instructions are learnable in principle, though the length of the shortest program required to do so is beyond the reach of STOKE's program synthesis engine.
- **Cost function.** There are several instructions for which STOKE's cost function does not provide sufficient guidance for search to proceed effectively. For instance, for some instructions it is difficult to produce a program that correctly sets the overflow flag. STOKE's cost function cannot provide much feedback for just this single bit until a mostly correct program is found.

5.3 Correctness

STOKE already contains formulas for a subset of the x86-64 instruction set. These are written against the same model of the CPU state used in this paper, and so we can easily compare the STOKE formulas against the ones learned by STRATA by asking an SMT solver if the formulas are equivalent.

We have hand-written formulas from STOKE for 1,431.91 instructions variants (or 79.75%) of the formulas learned by STRATA (none of these were instruction variants with a constant). For 1,377.91 instruction variants, we could prove equivalence automatically. We manually inspected the 54 instruction variants where the SMT solver was not able to automatically prove the equivalence between the manually written formula and the one synthesized by STRATA. For 4 instructions, the formulas are in fact equivalent if some additional axioms about the uninterpreted functions are added: Because we encode all floating point instructions as uninterpreted functions, the SMT solver does not know that there is a well-defined relationship between some of these. For instance, a fused multiply-add with an additive operand that is zero is equivalent to a regular multiply. Adding such axioms allows us to prove the equivalence for these 4 instructions. Alternatively, if an encoding for floating point operations would be used that the SMT solver can reason about, then no axiom would be necessary.

In the remaining 50 cases, there were semantic differences between the hand-written and synthesized formulas. We inspected each case manually by consulting the Intel manual as well as running example inputs on real hardware to determine which is correct. In all cases, the formulas learned by STRATA were correct and the manually written ones were not:

- The AVX extension adds many variants of existing instructions with three operands; two source operands and one destination. In 32 cases, the manually written formulas incorrectly left the upper 64 bits of the destination unchanged instead of copying the bits from the first source operand.
- For 10 vectorized conversion operations (from double to single precision floating point values or from doubles to 32-bit integers), the upper 64 bits of the result are supposed to be 0, but the hand-written formulas mistakenly left these unchanged.
- There are 8 variants of a combined addition/subtraction operation (e.g., `addsubps`). The hand-written formulas erroneously swap the position of the addition and subtraction operations.

In summary, we found 50 instruction variants where the hand-written formulas contained errors, and 4 cases where the formulas required some relatively simple axioms about the uninterpreted functions used to be proven equivalent. All other automatically learned formulas are provably equivalent to the hand-written ones.

Errors in Intel documentation The Intel manual [14] only contains informal pseudo-code and an English description of the semantics, so there is no automatic way of comparing the learned formulas with the manual. However, over the course of this project we found several inconsistencies when manually comparing the two. This is unsurprising given the over 3,800 pages of intricate technical content, the vast majority of which is not machine-checked. Examples of these errors include:

- The upper 128 bits of the output register of **`pshufb xmm0, xmm1`** are kept unmodified on Intel hardware, but the pseudo-code in the manual incorrectly says they are cleared.
- The *Op/En* column for the **`vextracti128`** instruction should be MRI, but it was RMI. This causes a wrong operand ordering when assembling this instruction.
- One possible encoding for **`xchgl eax, eax`** is `0x90` according to the manual, and its semantics are to clear the upper 32 bits of **`rax`**. However, `0x90` is also the encoding of **`nop`**, which does nothing according to the manual (and hardware).
- The textual description of `roundpd` claims to produce a single-precision floating point value, when it produces a double-precision floating point value.

We have reported all of these items and they were confirmed by Intel as errors in the manual [8].

5.4 Formula Size

Large formulas can cause problems for SMT solvers and perform less well, especially if formulas are further combined (e.g., to prove equivalence between two x86-64 programs). For this reason, we answer (RQ3) by comparing the number of nodes in the AST of the formula representation in the automatically learned formulas and the hand-written formulas. For every instruction variant where we have both a learned and hand-written formula, we compute

$$\frac{\text{number of nodes in learned formula}}{\text{number of nodes in hand-written formula}}$$

We plot a histogram of this distribution in logarithmic space (such that increases and decreases in size are shown symmetrically) in Figure 1. The center of the figure consists of formulas that have the same size, with instructions where STRATA's formulas are smaller to the left and instructions where the hand-written formulas are smaller to the right. The median of this distribution is one (i.e., formulas are equally large). The smallest formula has size 0 (for the **`nop`** instruction), and the largest formula has 211 and 196 nodes for STRATA and the hand-written formulas, respectively.

5.5 Formula Precision

Two further important aspects of the quality of formulas is the number of uninterpreted function applications (a measure of preciseness) and the number of non-linear arithmetic operations (which tend to perform poorly in queries), as stated by

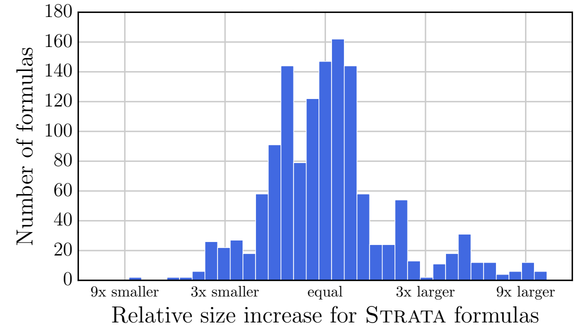

Figure 1. Distribution of the relative increase in size of the formulas learned by STRATA compared to the hand-written formulas. Negative numbers indicate that the learned formulas were smaller.

(RQ4). We found that the automatically synthesized formulas have the same number of non-linear operations, except for 9 instruction variants, where they contain fewer such operations (the hand-written formulas for some rotate instructions contain a modulo operation, whereas the automatically learned ones do not). In all of these cases the non-linear operation is a modulo by a constant, and is likely not a big problem for SMT solvers (unlike non-linear operations by variables).

The number of uninterpreted functions (UIF) is the same for almost all instruction variants, with the exception of 4 formulas, where the automatically learned formulas contain redundant UIFs: Instead of using a single UIF, the learned formula uses 2 UIFs that combined perform the same operation as the single UIF. In all cases a simplification step that knows about floating point operations should be able to remove these redundant UIFs.

In summary, the automatically learned formulas are comparable in size, and with the exception of 4 instruction variants are equally precise and contain at most as many non-linear arithmetic operations compared to the hand-written formulas.

5.6 Simplification

To address (RQ5), we measure the size of the learned formulas before and after simplification and compute the relative decrease in size (as before when comparing with handwritten formulas). At the median of this distribution, simplified formulas are 8.45 times smaller, and in the extreme they are smaller by up to a factor of 1,583. A small number of formulas are larger after simplification (due to using a distributive law), but at most by a factor of 1.33. The full distribution is shown in Figure 2.

5.7 Stratification

To answer (RQ6), we can investigate if instructions could have been learned sooner. To this end, we define the *stratum* of a formula that corresponds to an instruction variant i induc-

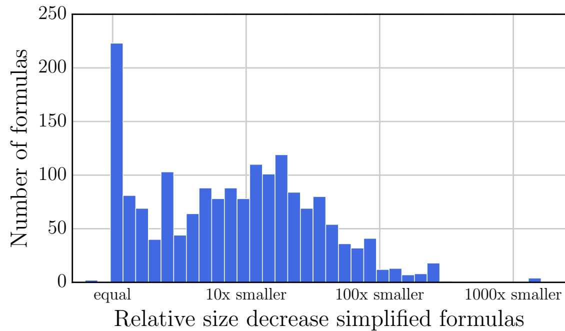

Figure 2. Distribution of the relative decrease in size of the formulas learned by STRATA due to the simplification step.

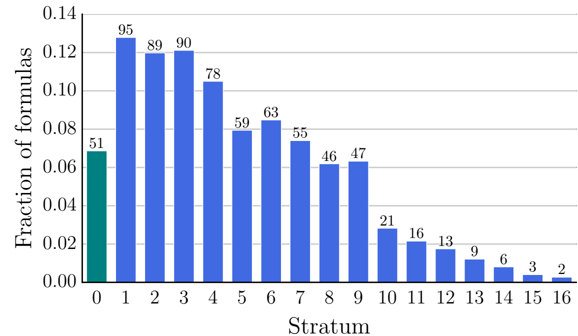

Figure 3. Distribution of the strata for all register-only instruction variants. Formulas at stratum 0 are from the base set, and all others are learned formulas.

tively as follows. Let $M(i)$ represent the set of instructions that the synthesized program for i uses, and let B be the set of instructions in the base set. Then,

$$\text{stratum}(i) = \begin{cases} 0 & \text{if } i \in B \\ 1 + \max\limits_{i' \in M(i)} \text{stratum}(i') & \text{otherwise} \end{cases}$$

Intuitively, this captures the earliest point at which a particular formula could have been learned with the given program. We show the distribution of strata in Figure 3, which shows that stratified synthesis is in fact necessary for our approach to learning the semantics of x86-64: only 13.73% of instructions are learned at stratum 1, the 50th percentile of all learned instructions is reached only at stratum 4, and the 90th percentile is not reached until stratum 10.

In STRATA, formulas are added to the base set as soon as a specification has been learned (by learning multiple programs). So when an instruction i at stratum n is added to the base set and another instruction i' makes use of i during the search for its specification, we say i' was learned at stratum (at least) $n + 1$. However, there might be another (equivalent) specification where the instruction i was not used, and thus the stratum for i' then might be different (and lower). Therefore, to definitively answer (RQ6), we run STRATA where new instructions are not added to the base set (for the same amount of time, on the same machine as the main experiment). This effectively turns off stratification and determines the maximum number of instructions this approach can learn at stratum 1. We find that for the register-only variants, we can learn only 426 formulas without stratification, compared to the total 692 formulas learned with stratification (an increase of 62.4%). In the experiment with stratification, only 95 formulas were learned at stratum 1, showing that indeed, some formulas at higher strata could have been learned at lower strata if we had given more time to the program synthesis at those lower strata. However, overall stratification was essential to learning.

5.8 Experimental Details

We performed all experiments on an Intel Xeon E5-2697 machine with 28 physical cores, running at 2.6 GHz. For engineering reasons, we split the problem into learning register-only variants and learning formulas for instructions with an 8-bit constant. Finally, we generalized these formulas to other similar instruction variants with memory or immediate operands.

Our algorithm is embarrassingly parallel and we run it on all available cores. We let the register-only experiment run for 268.86 hours, and spent 159.12 hours for the 8-bit constant experiment. Total runtime therefore is 427.98 hours, or 11983.37 CPU core hours. Figure 4 shows the progress during the learning of register-only instructions, both for STRATA as presented and STRATA with stratification turned off. STRATA is able to quickly learn formulas for the simple instructions, with progress tapering off as more difficult instructions are attempted.

During the process of learning formulas, we invoked an SMT solver to decide what equivalence class a program belongs to (procedure CLASSIFY in Algorithm 2). Out of the 7,075 such decisions STRATA had to make when learning register-only variants, we found an equivalent program in 6,669 cases (94.26%) and created a new equivalence class in 356 cases (5.03%) as the solver could not prove equivalence with any existing class. In only 3 out of these 356 cases a solver timeout was the cause, and in all other cases the solver did not give a valid counterexample.

We also found a (valid) counterexample in 50 cases. This highlights the importance of learning many programs, as the first one we learn can actually be wrong. It also shows that for the most part the testcases work well.

5.9 Implementation

We make STRATA, our implementation of stratified synthesis available online, together with the results of the experiments described in this paper including all learned formulas.

https://stefanheule.com/strata/

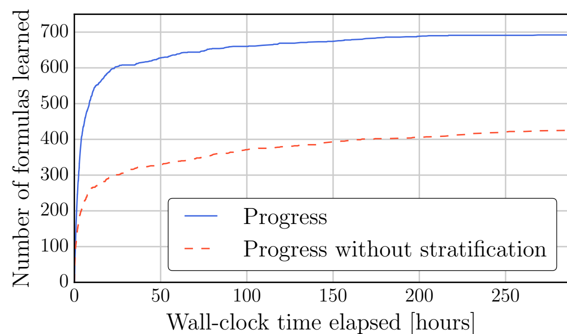

Figure 4. Progress over time as measured by the number of (register-only) formulas learned.

6. Related Work

There has been a proliferation of program synthesis techniques in recent years. These include techniques based on SAT solvers [28], SMT solvers [13, 15], exhaustive enumeration [4, 11, 20, 30], and stochastic search [19, 26, 27]. In this work, we demonstrate that stratification significantly enhances the performance of STOKE, a stochastic synthesizer of x86-64 code. We believe that other approaches to program synthesis can also benefit from the modularity introduced by stratification, which we leave as future work. Our focus in the current work is to obtain a formal semantics for x86-64 instructions.

The work closest to ours is by Godefroid and Taly [12]. Using 6 manually provided templates and SMT-based synthesis, they are able to synthesize formulas for 534 32-bit x86 arithmetic (no floating-point) instructions. For tractability, the templates are required to have "smart inputs", which is a set of inputs such that if two instantiations of the template agree on the smart inputs then the instantiations are guaranteed to be equivalent for all inputs. They show that in the absence of smart inputs, the SMT based synthesizer runs out of memory while synthesizing formulas for 32-bit shifts [12]. In contrast, we do not require templates, smart or otherwise. We bootstrap a small base set of formulas to obtain formulas for a larger set of instructions. While the base set of formulas that we use were written manually, many of these could be synthesized by extending the approach of [12] to x86-64.

The HOIST system automatically synthesizes abstract semantics (for abstract domains such as intervals, bit-wise domain, etc.) by exhaustively sampling 8-bit instructions of embedded processors [23]. In subsequent work, templates were used to scale this process to 32-bit instructions [22]. In contrast, we synthesize bit-precise concrete semantics for x86-64 instructions that operate on registers which can have up to 256 bits. These concrete semantics can be used to obtain abstract semantics (for any abstract domain) using [25].

Several research groups have implemented, with considerable effort, their own formal specification for the x86 instruction set. Our work automates this task and generates a more complete specification than the existing ones. For instance, CompCert includes a partial formal specification for 32-bit x86. The paper [30] uses formulas for 179 32-bit x86 opcodes described in [18]. The STOKE tool itself contains manually written formulas for a subset of x86-64. Other platforms for x86 include BAP [7], BitBlaze [29], Codesurfer/x86 [3], McVeto [31], Jakstab [16], among many others. We note that Intel does not appear to have a formal model that fully defines CPU behavior, not even internally [1, Section 4.1].

The idea of comparing two implementations by translating them to SAT or SMT solvers has been used in many contexts, e.g., deviation detection to find bugs [21] or to determine the soundness of optimizations [5]. In our work, we use it to find corner-case inputs that are not part of our testcases.

7. Conclusions

We have presented STRATA, a tool for automatically synthesizing the semantics of x86-64 instructions. Using stratified synthesis, in which we use the semantics of instructions we have learned to bootstrap the process of learning the semantics of additional instructions, we have learned specifications for 1,795 x86-64 instruction variants starting from a base set of just 58 instructions and 11 pseudo instruction templates. Our results are the most complete formal description of the x86-64 instruction reported to date.

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government. This work was also supported by NSF grants CCF-1409813 and CCF-1160904, as well as a Microsoft Fellowship.

References

[1] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 311–327, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815420. URL http://doi.acm.org/10.1145/2815400.2815420.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Compiler Construction*, pages 250–254. Springer, 2005.

[3] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 250–254, 2005.

[4] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 394–403, 2006.

[5] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168906. URL http://doi.acm.org/10.1145/1168857.1168906.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, 2011.

[8] M. Charney. Personal communication, February 2016.

[9] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, volume 31, pages 88–95, 2005.

[10] E. Darulova and V. Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248, 2014.

[11] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[12] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254116. URL http://doi.acm.org/10.1145/2254064.2254116.

[13] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[14] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, Revision 325462-057US, December 2015. URL http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[16] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 423–427, 2008.

[17] X. Leroy. The CompCert C Verified Compiler, 2012.

[18] J. Lim and T. W. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4, 2013.

[19] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 208–217, 2015.

[20] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

[21] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification*, 2011. doi: 10.1007/978-3-642-22110-1_55. URL http://dx.doi.org/10.1007/978-3-642-22110-1_55.

[22] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*, pages 34–43, 2006.

[23] J. Regehr and A. Reid. HOIST: a system for automatically deriving static analyzers for embedded systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 133–143, 2004.

[24] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Compiler Construction*, pages 16–35. Springer, 2008.

[25] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 252–266, 2004.

[26] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9, 2014.

[28] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Pro-*

gramming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, pages 281–294, 2005.

[29] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, pages 1–25, 2008.

[30] V. Srinivasan and T. W. Reps. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 596–607, 2015.

[31] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed proof generation for machine code. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 288–305, 2010.

[32] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.