Master Thesis
Computer Science

Radboud University

libLISA: Learning Instruction Set Architectures from scratch

Author:
Jos Craaijo
s4481674
`j.craaijo@outlook.com`

First supervisor:
Freek Verbeek
`freek@vt.edu`

First assessor:
Bernard van Gastel
`B.vanGastel@cs.ru.nl`

Second assessor:
Freek Wiedijk
`freek@cs.ru.nl`

June 22, 2021

Abstract

CPU instruction semantics are often written manually in natural language. Many software analysis tools and compilers depend on these semantics. We present an approach to learning the semantics of instructions with minimal human input. Our approach only needs a description of CPU state and a way to observe the result of executing an instruction from a certain CPU state. We introduce a novel algorithm to analyze instructions to extract data flow information and generalize instructions into *encodings*. We then use existing program synthesis techniques to synthesize semantics for each encoding. We implement this approach in libLISA, and show that it can learn the semantics of some non-privileged x86-64 instructions that operate on general-purpose registers and flags.

Contents

1 Introduction **2**

2 Related work **4**
2.1 Instruction enumeration 4
2.2 Instruction set semantics 6

3 Approach **8**
3.1 Instruction semantics 9
3.2 Enumeration 10
3.2.1 Encoding Analysis 10
3.2.2 Filters 11
3.2.3 Instruction prefixes 11
3.3 Synthesizing Semantics 12
3.4 Implementation 14
3.4.1 Observations 15
3.4.2 Results 15

4 Encoding analysis **17**
4.1 Fuzzing 18
4.2 Inferring dataflows 18
4.2.1 Memory accesses 19
4.2.2 Normal dataflows 21
4.3 Inferring encodings 21
4.3.1 Comparing dataflows 22
4.3.2 Changes in dataflows 23
4.3.3 Changes in output values 27
4.4 Encoding parts 27
4.5 Instantiation 29

5 Evaluation **30**
5.1 Enumeration completeness 30
5.2 Semantics correctness 32
5.3 Coverage 34
5.4 Undocumented instructions 34

6 Conclusion **36**
6.1 Discussion 36
6.2 Future work 38

Chapter 1

Introduction

Modern CPU instruction set architectures (*ISAs*) have complex semantics. Each CPU implements its own version of an ISA, with unique implementation details and semantics. Manufacturers provide semantics for their CPUs in huge reference manuals. For example, Intel's x86 reference manual [3] specifies semantics using pseudocode and natural language in more than 5,000 pages.

Tools like compilers, disassemblers, decompilers and binary program analysis frameworks like Valgrind [14] or Pin [15] rely on specifications of these semantics. All these tools perform some kind of translation from or to instruction semantics. Compilers translate higher-level code to sequences of instructions with the same semantics. Disassemblers, decompilers and binary program analysis frameworks translate instruction semantics back to a higher-level representation or interpretation.

For many of these tools, custom, hand-written specifications are created, which is an error-prone process. Writing these semantics can sometimes be guesswork, as reference manuals contain errors. For example, the developer of the Intel XED (dis)assembler library stated: "I've found enough errors in the docs that, when I can, I usually check the hardware directly or have someone look at the RTL for a couple of designs." [1]

Besides being error-prone, specifying semantics is also incredibly time-consuming. For example, one attempt at formally specifying the full, non-deprecated, x86-64 user space instruction set took 8 man-months [4]. This makes it infeasible to scale this approach to multiple implementations. It is important to be aware of differences between implementations, even if both implementations conform to the official specification. For example, Sandsifter [6] identified an issue where Intel processors interpreted an instruction's length differently compared to AMD processors and most disassemblers and emulators (like QEMU). Such a difference makes it possible to hide malicious code from analysis tools.

Because of these two issues, it is important to have *trustworthy* and machine-readable semantics. Trustworthy semantics should be correct, complete and easily verifiable. When semantics are easily verifiable, we no longer need to trust authoritative sources like reference manuals. Semantics can only be easily verifiable if verification can be automated. From this, it follows that semantics must therefore also be machine-readable. A side-effect of having machine-readable semantics, is that compilers and analysis tools can also re-use these same semantics.

There is little directly related work. Related work either focuses on enumeration of instructions, or synthesis of semantics, but not both. Existing manual and computer-generated specifications rely on disassembler libraries, which contain hand-written code

[1] `https://github.com/intelxed/xed/issues/56#issuecomment-312376836`

to decode byte sequences. Only the semantics of the decoded instructions are specified. Disassembler libraries are not trustworthy, since we cannot easily verify that they function correctly. The reliance on disassembler libraries also means that existing specifications are also hard to scale. Generating semantics for a new architecture requires manually writing a disassembler library for that new architecture, which is a time-consuming process.

In this thesis, we introduce an algorithm to learn trustworthy semantics of instruction sets, without prior knowledge of instruction syntax or behavior. Additionally, we build an implementation of our algorithm, libLISA, which can automatically learn semantics for the x86-64 ISA. We limit the scope of our implementation to instructions that operate on general-purpose registers and flags in Linux x86-64 user space.

Compared to existing solutions, our approach requires less information. It only needs to know how to execute an instruction and how to observe CPU state, but it does not need any knowledge on how instructions are structured.

We run libLISA on a Ryzen 3900X, and enumerate around 36.13% of the instruction space in 322 hours, finding 25373 instruction variants. Our enumeration correctly identifies most instructions, finding 99.20% of all instructions within the enumerated range. We learn the correct semantics of 4297 instruction variants, which represents around 4.65% of the instruction space. Additionally, we identify a group of undocumented instructions by comparing the output of libLISA to the disassembler library XED.

All results in this thesis, including the tool and its output are publicly available at: `https://github.com/Jos635/libLISA`.

Our approach still has limitations. Specifically for x86-64, we are unable to handle the millions of possible prefixes an instruction might have. We solve this by restricting prefixes to a fixed order. Additionally, we use a synthesis approach that does not support floating point operations, vector operations or memory reads or writes of more than 64 bits.

In Chapter 2, we discuss related work. In Chapter 3, we give an overview of the approach we have taken. In Chapter 4, we explain one part of our approach, encoding analysis, in more detail. In Chapter 5, we evaluate the correctness and completeness of our approach. Finally, we conclude in Chapter 6.

Chapter 2

Related work

Our work combines two areas: instruction enumeration, and instruction set semantics. As there is no related work combining these areas, we will discuss work for each area separately.

2.1 Instruction enumeration

Small instruction sets can be enumerated effectively by considering all possible byte strings in lexicographical order. In variable-length instruction sets like x86-64, there are so many instructions that it becomes infeasible to enumerate them all. Research into instruction enumeration focuses on developing methods to skip less "interesting" instructions.

Work in this area started with the Sandsifter tool [6], a CPU fuzzer for finding undocumented instructions. Sandsifter introduces the concept of *tunneling*. When tunneling, you extend a byte string one byte at a time as long as you observe that the byte string is too short for an instruction. Having "tunneled in" to the full instruction, we start iterating through possible instructions in lexicographical order. In order to not get "stuck" in a 16-byte deep hole, we start taking progressively bigger iteration steps as long as the instructions look similar. We determine whether two instructions look similar by comparing their length, and potentially some other properties (like whether the instruction performs a memory access). Steps are commonly increased by factors of 2^8, aligning the changed bits in the enumerated instructions with byte boundaries. This approach, which we will call *length-based enumeration*, is illustrated in Figure 2.1.

By doing length-based enumeration, Sandsifter is able to enumerate a large part of x86-64 instruction space with very little prior knowledge. Because it needs only simple observations on instruction length and basic instruction behavior, Sandsifter enumerates instructions relatively quickly while still remaining accurate. Sandsifter identified many undocumented instructions both on Intel and AMD CPUs. It also identified a flaw in disassembler libraries, a flaw in the Azure hypervisor and a "halt and catch fire" instruction on an undisclosed CPU [6].

Unpublished work by Mahoney and McDonald [13] uses a different approach. Rather than starting to skip instructions at some arbitrarily chosen point, they instead use a disassembler to determine which instructions are less interesting to enumerate, and can be skipped. Whereas a normal disassembler might output a list of operands for the instruction, their modified disassembler instead outputs which bits were used to determine the operands. This information can then be used to skip similar instructions, i.e., instructions that only differ in which operands are being used. The idea is that once we have seen one instruction, for example `add r1, r2`, we can probably expect the instruction to function

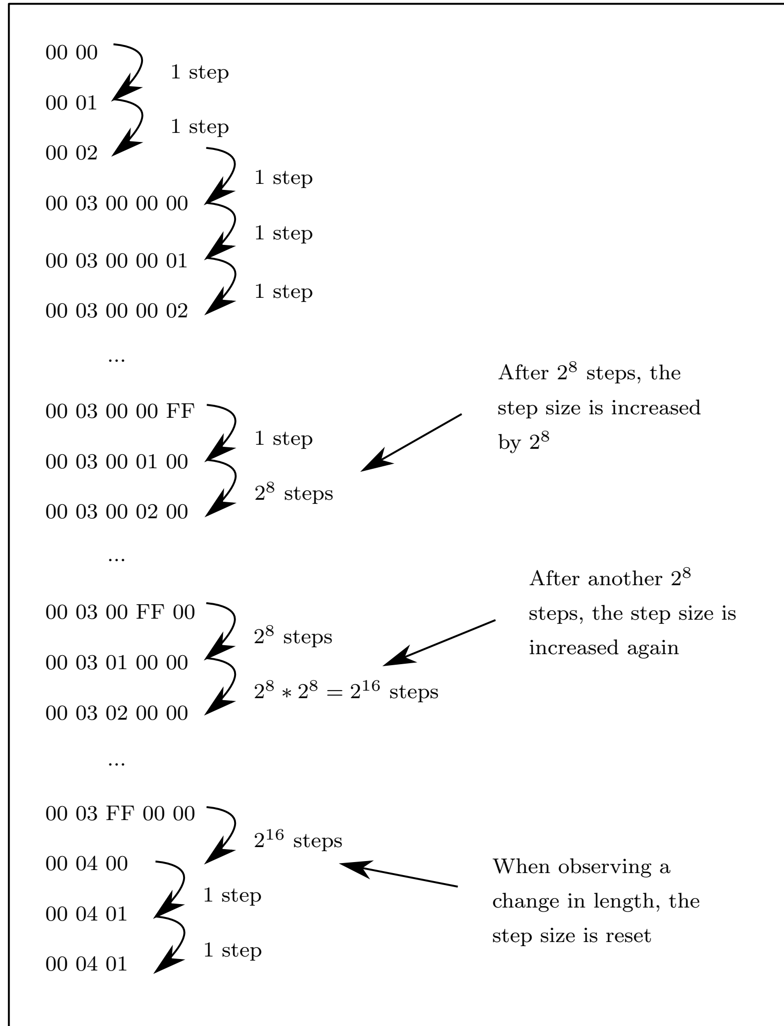

Figure 2.1: Length based enumeration increases the step size by some factor (here 2^8) if instructions remain similar. Once an instruction seems to be different, step size is reset back to 1.

the same if we alter an operand, for example, `add r1, r3` would not be interesting if we have already seen `add r1, r2`. We call this approach *encoding-based enumeration*.

Both techniques were only recently combined, in UISFuzz [12]. UISFuzz is a CPU fuzzing tool used for finding undocumented instructions. By default, encoding-based enumeration is used. Whenever the disassembler does not have information in its database about an instruction, or when instruction decoding fails, UISFuzz falls back to length-based enumeration. The subsequent iScanU [5] adapts this approach to work for fixed-length instruction sets like RISC-V or ARMv8.

UISFuzz and iScanU are more performant than Sandsifter. The extra knowledge from the disassembler library allows UISFuzz to enumerate the entire instruction space six times faster than Sandsifter [12]. In theory, an encoding-based approach would also be more accurate, since it can make more informed decisions about which instructions are interesting, and which ones are not.

All approaches we discussed in this section are aimed at finding undocumented instructions. We are interested in learning the semantics of the instructions without using

manually generated knowledge. This means that we cannot use a disassembler library. Encoding-based approaches are therefore not possible. Length-based enumeration is less accurate than encoding-based enumeration. In certain cases, length-based enumeration might skip over valid instructions. For example, if an instruction `00 F0 13` is valid, but all other instructions between `00 00 00` and `00 FF FF` are invalid, it will not be found. Tunneling will enumerate `00 00 00`, `00 00 01`, `00 00 02`, ..., `00 00 FF`, then `00 01 00`, `00 02 00`, `00 03 00`...`00 FF 00`. It will not consider the instruction `00 F0 13`, because `00 F0 00` is not a valid instruction, so it continues to tunnel.

Length-based enumeration is also not possible on fixed-length instruction sets, since it relies on differences in instruction lengths. And finally, length-based enumeration enumerates many more instructions than encoding-based enumeration. For example, given an instruction `mov r1, #0x00000000` (where the second operand is an immediate value encoded in the instruction byte sequence), we would enumerate at least 1024 variations of the form `mov r1, #0x00000001`, `mov r1, #0x00000002`, ..., `mov r1, #0x000000FF`, `mov r1, #0x00000100`, `mov r1, #0x00000200`, corresponding to progressively increasing step sizes. This significantly increases the amount of instructions for which we will need to synthesize semantics.

2.2 Instruction set semantics

Official instruction set manuals, like Intel's x86-64 reference manual [3] do not fit the definition of trustworthy, as we saw in Section 1. The semantics are described in a mix of natural language and pseudocode, and have often been found to contain errors.

Over the years, several attempts [8, 9] have been made to manually produce formal specifications for the x86-64 instruction set. Most recently, Dasgupta et al. [4] were the first to provide a (manually generated) complete formal semantics of the x86-64 user-level instruction set. It covers 3155 instruction variants, corresponding to 774 different *mnemonics*. The semantics were thoroughly validated by running co-simulations against real hardware, and by manually comparing the semantics to earlier work. The semantics replicate the official x86-64 specifications, and include special symbols for *undefined* results (results that may be different on each implementation).

Manually writing semantics is a time-consuming process. For example, it took Dasgupta et al. 8 man-months to produce the semantics [4]. Various attempts have been made to automatically learn semantics. Godefroid and Taly [7] proposed using program synthesis techniques to automatically synthesize some semantics. By constructing "templates" for common CPU operations, they were able to synthesize semantics for some x86 instructions. The templates must still be specified manually, but templates are much more generic than instructions. From 6 templates, semantics for 534 instruction variants could be generated automatically.

Later work [10] uses a more advanced program synthesis technique, *stratified synthesis*. With stratified synthesis, a *base set* of instruction semantics is needed. The set should cover all unique functionality in the instruction set. Semantics for new instructions are generated by synthesizing small programs consisting of only instructions in the base set using STOKE [17]. Semantics for around 1,800 instruction variants were learned from a base set of just 51 instructions. When comparing with earlier (hand-written) work, 50 cases with semantic differences were found. In all cases, the learned semantics were correct and the hand-written specification was wrong.

All approaches we discussed in this section rely on a manually generated specification of instruction syntax, operands and reading/writing behavior (usually an assembler/dis-

assembler library). This allows details like memory accesses and operand types to be abstracted away. To date, to the best of our knowledge, there exists no system that can synthesize semantics without relying on such a manual specification.

Chapter 3

Approach

Our goal is to build an algorithm that can learn the semantics of all "valid" instructions of a CPU. Semantics available in, for example, reference manuals and disassembler libraries are often manually specified. Manually specifying semantics is a time-consuming and error-prone process, so we do not want to rely on manually generated information.

Definition 1. We will use the term *instruction* to refer to a single bit sequence. The length of such a bit sequence is usually a multiple of 8.

A *mnemonic* describes a set of instructions. For example, in x86-64 the `ADD`[1] mnemonic describes all instructions that perform an addition. What is usually referred to as "the `ADD` instruction" actually consists of 22 different opcodes, and billions of different bit sequences. The mapping of instructions to mnemonics is described in reference manuals. Since we do not want to rely on reference manuals, we will not use the concept of mnemonics in this thesis.

Similarly, the "validity" of an instruction is often also determined by reference manuals. We instead use the term *decodablility*.

Definition 2. An instruction is decodable if the CPU, when presented with the instruction, does not raise an undefined instruction exception (for example the "#UD" exception for x86, or the "UNDEF" exception for ARM).

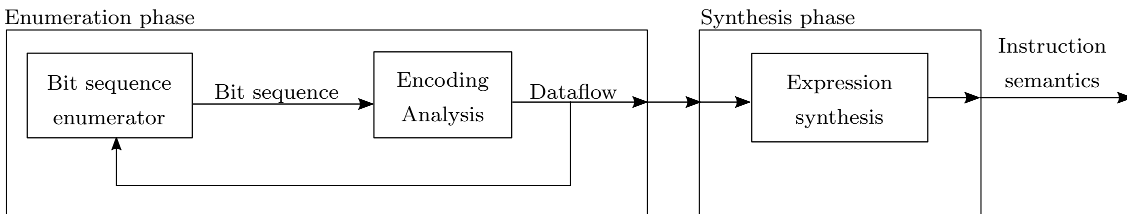

Figure 3.1: Learning an instruction set can be divided into two phases: enumeration and synthesis.

In this thesis we describe an algorithm for learning an instruction set from scratch. As illustrated in Figure 3.1, we divide the algorithm into an *enumeration* and a *synthesis* phase. The goal of the enumeration phase is to determine all decodable bit sequences. Some analysis is used to determine which bit sequence to consider next. In prior work,

[1] Some assemblers do use different mnemonics for different operand kinds, for example `ADDb` for an addition of bytes. Even in that case, the mnemonic still groups instructions with different operands together. Operands might for example be registers, memory or immediate values.

a disassembler libraries or instruction length analysis has been used. We use a more extensive *encoding analysis*, based on fuzzing. We explain our encoding analysis in more detail in Chapter 4.

During the synthesis phase, the semantics of each decodable bit sequence are determined using program synthesis techniques. We use the output of the encoding analysis as a "template" for the semantics we will generate, to speed up this phase.

3.1 Instruction semantics

When executing an instruction on a CPU, that instruction can modify *storage locations*. Storage locations can be *registers* or *memory locations*. We consider CPU *flags* to be special registers which can only be 0 or 1. Some instructions also produce side-effects such as outputting a value on a processor pin, triggering an interrupt or changing processor modes. We consider such side-effects out of scope.

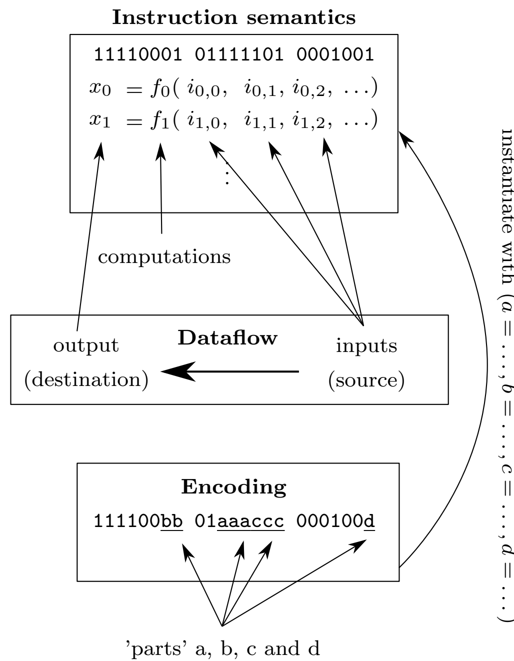

Figure 3.2: The semantics of a CPU instruction.

To describe the *semantics* of an instruction, we describe how each storage location is updated. This gives us a set of assignments of the shape $\{x_0 = f_0(i_{0,1}, i_{0,2}, \ldots), x_1 = f_1(i_{1,1}, i_{1,2}, \ldots), \ldots\}$ as illustrated in Figure 3.2. We will refer to $x_0, x_1, \ldots$ as the *outputs* of the instruction, and to $a_1, a_2, \ldots$ as the *inputs* for x_0. We call this a *dataflow* from $i_{0,1}, i_{0,2}, \ldots$ to x_0. To determine the output values of $x_0, x_1, \ldots$, the CPU performs *computations* $f_0, f_1, \ldots$. When leaving out the computations, we get a partial specification of semantics, consisting only of inputs and outputs which we call *dataflows*.

Modern CPUs can often access billions of distinct memory locations. Adding a separate assignment for each memory location is therefore unfeasible. Instead, we identify memory locations by the order in which they are accessed. For each memory location an instruction accesses, we add an additional assignment which computes the address of the storage location.

Example 3. *Consider an instruction `02 04 8a` that computes the addition of a register r_1 and a value stored at memory address $r_2 + r_3 * 4$, and stores this result in r_1. This instruction is similar to `sum = sum + array[index]` in C code.*

We describe its execution with the following assignments:

1. $Address(m_1) = r_2 + r_3 * 4$
2. $r_{IP} = r_{IP} + 3$
3. $r_1 = r_1 + m_1$

For simplicity we omit the storage locations that do not change. Note in particular, that we are also describing how the instruction pointer r_{IP} is updated afterwards (in 2). While it might be trivial for this specific example, it is needed for instructions such as jumps or function calls.

Many CPUs include support for accessing different parts of the same register separately. For example, the 8-byte (64-bit) x86-64 register `rax` can be accessed via `eax` (lower 4 bytes[2]), `ax` (lower 2 bytes), `ah` (one-but-lowest byte) and `al` (lowest byte). For this reason, we consider each operand to have a *size*. An operand size is a range of byte indices that are read or written when the operand is accessed. Each destination and source might have a different operand size, but the same source or destination may not occur multiple times with different operand sizes.

We can group similar instructions together in *encodings*. An encoding can describe a group of instructions that only differ by inputs, outputs or immediate values. These differences must be caused by certain *parts* of the instruction bit sequence. To get semantics out of an encoding, we can *instantiate* it with values for each of the parts. For example, to instantiate the encoding `111100bb 01aaaccc 000100d`, we must provide values for `aaa`, `bb`, `ccc` and `d`. Instantiating this encoding with $a = 111, b = 01, c = 101, d = 1$ would give us the instruction semantics shown in Figure 3.2.

3.2 Enumeration

As discussed in Chapter 2, the two main approaches to instruction enumeration or length-based and encoding-based enumeration. Length-based enumeration is requires little information, but is less accurate and does not work for fixed-size instruction sets. We prefer to use encoding-based enumeration, as it is more accurate. However, it requires knowledge about the structure of an instruction, often in the form of a disassembler library. Our goal is to use as little manually generated information as possible, so we cannot use a disassembler library. Instead, we introduce *encoding analysis* as a replacement. Encoding analysis can provide the information we need for encoding-based enumeration in many cases. When encoding analysis fails, we fall back to length-based enumeration.

3.2.1 Encoding Analysis

Our encoding analysis can learn the encoding and partial semantics (dataflows) of groups of instructions, without using any manually provided information. Encoding analysis produces semantics similar to those described in Section 3.1, but without any computations.

[2]When writing to register `eax`, the upper 4 bytes of `rax` are set to 0, while writing to `ax`, `al` or `ah` will not do so.

During enumeration, we are only interested in the classification of bits into parts. The partial semantics are of no use to us during enumeration, but form the basis of the semantics that we will learn during synthesis. Once we have identified an encoding, we can skip over all but one instruction in the encoding, as we already know how they behave.

Example 4. *We have identified an encoding* `00000100 aaaaaaaa`*, which performs an addition* $r_0 = r_0 + a$*. The part* `aaaaaaaa` *is the immediate value that is added to* r_0*. For example,* `00000100 00000010` *performs the addition* $r_0 = r_0 + 2$*.*

Given the semantics of, for example, `00000100 00000010`*, it becomes trivial to determine the semantics of any other instruction in this encoding. To determine the semantics of* `00000100 00000001`*, we can simply replace the* 2 *by a* 1*, giving* $r_0 = r_0 + 1$*.*

To enumerate as efficiently as possible, we want to enumerate either the instruction `00000100 00000010` *or* `00000100 00000001`*, but not both. Once we have identified the encoding for one of these instructions, we can skip the other 255 similar instructions.*

3.2.2 Filters

We aim to enumerate just one instruction for each encoding. Once we have enumerated an instruction belonging to an encoding, we generate a *filter* to skip over all other instructions also belonging to that encoding.

Example 5. *Assume we are currently considering an instruction* `0000 0110 1101 0011`*. Encoding analysis infers an instruction encoding* `0000 0110 1bb1 0aaa`*, where* `aaa` *and* `bb` *are two parts.*

We can now filter all instructions matching `0000 0110 1bb1 0aaa`*, i.e., all instructions matching the filter* `0000 0110 1__1 0___`*, where '_' may be either a 0 or a 1. The next instruction we enumerate will therefore be* `0000 0110 1101 1000`*.*

If encoding analysis fails, we cannot continue enumerating instructions one-by-one until we eventually get to an instruction for which encoding analysis works again. It might take billions (e.g., the equivalent of a 32-bit immediate value) of instructions before encoding analysis is able to analyze an instruction again.

To mitigate this issue, we adapt the tunneling approach from Sandsifter [6]. Whenever encoding analysis fails and we carry a bit past a byte boundary, we increase the step size by which we enumerate the instructions by a factor of 2^8.

Example 6. *Assume we are currently enumerating instruction* `0E 12 64`*, and that encoding analysis fails for this instruction. If we are not filtering any instructions, we will continue enumerating normally to* `0E 12 FF`*. On the next increment, the* `FF` *will overflow and we carry a 1 to* `12`*. At this point, we start taking steps of* 2^8 *instructions at a time. So after* `0E 12 FF` *follows* `0E 13 00`*, then* `0E 14 00`*,* `0E 15 00`*, etc.*

Whenever we encounter any kind of difference, we reset the step size back to 1. This difference might be a change in instruction length, an instruction that gets filtered or successful encoding analysis.

If we still have not encountered an instruction that we could analyze successfully by the time we reach `0E FF 00`*, we once again increase the step size by another* 2^8 *(giving a step size of* $2^8 * 2^8 = 2^{16}$*. After* `0F 00 00` *follows* `10 00 00`*, then* `11 00 00`*, etc.*

3.2.3 Instruction prefixes

Unfortunately all these steps still do not allow us to efficiently enumerate the entire x86-64 instruction space. This is caused by *instruction prefixes.* Instruction prefixes are

sequences of one to three bytes that are placed in front of the instruction. The x86-64 instruction set allows an instruction to be prefixed multiple times, even if prefixes are conflicting. The only constraint is that the total instruction length must be at most 15 bytes. As a consequence, there are many ways to encode the same instruction. For example, there are at least 78 billion ways to encode the NOP (no operation) instruction (considering only the case where the NOP instruction consists of 14 bytes, each filled with one of 6 segment override prefixes, followed by the NOP opcode byte `0x90`).

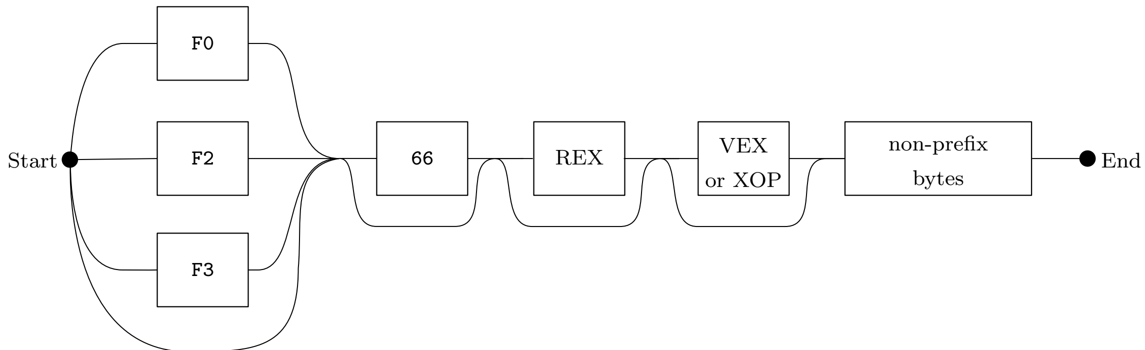

Figure 3.3: We permit 6 different prefixes: one prefix from group 1 (`F0`, `F2` or `F3` – which can be mandatory for some instructions), followed by a group 3 operand size prefix `66`, then a REX prefix and finally a VEX or XOP prefix. All prefixes are optional, and may be skipped. Note that `F2`, `F3` or `66` prefixes may not be placed in front of a VEX prefix. We expect the CPU to give us an invalid instruction exception in those cases.

We believe it might be possible to identify prefixes automatically, but consider this out of scope. To make enumeration possible, we simply enforce a fixed order of prefixes. We also do not allow any of the segment override prefixes, as 4 of them have no effect in long mode[3] (long mode is the name of the 64-bit mode of x86-64 CPUs) and the other two are harder to use from Linux user space. We also exclude the rarely-used address size prefix (`67`), which limits all address calculations to 32 bits. The prefixes we permit are illustrated in Figure 3.3.

3.3 Synthesizing Semantics

To synthesize instruction semantics, we use the dataflows we learned during enumeration as a starting point. Each dataflow consists of sources and a destination. To turn a dataflow into semantics, we need to synthesize a computation. As described in Section 3.1, a computation is a function f that takes the values of the sources as inputs, and returns the new value of the destination as output. We will use an *enumerative* program synthesis technique [2] to synthesize an expression for each computation. We employ the *divide-and-conquer* approach proposed by Alur, Radhakrishna and Udupa [1]: we separately enumerate expressions that are correct for a subset of all possible inputs, and predicates to distinguish the subsets. We then combine these using *decision trees* to form the full computation.

Enumerative program synthesis does not use theorem proving or SMT solvers to intelligently synthesize a program from some specification. Instead, it simply enumerates all possible programs that fit some grammar, in order to find a program that produces the same result as the real computation.

[3]This only applies to the common case where compatibility mode is not being used. We consider only the non-compatibility long mode in scope.

We use a non-recursive grammar to simplify enumeration. Our syntax describes a linear program, which performs a number of steps in sequence. The steps must occur in a specific order. Inputs must be preprocessed first, then constant values may occur, and after that bitshifts, divisions, bitwise operations (in disjunctive normal form, DNF) and arithmetic operations (sums of products) may occur in any order. We also include a shorthand parity operation that computes the XOR of the lower 8 bits of a value, since this is commonly needed for x86 flags. The result of each step is stored in a fresh variable, giving a static single assignment (SSA) program.

The semantics of the grammar are as follows. All values are whole, signed, 128-bit numbers. Preprocessing an input consists of interpreting it as a little endian or big endian number, and optionally sign-extending it. This "lifts" the bitvector input into a 128-bit number. Steps in a program are executed one-by-one. The result of the last operation is returned as the result of the program. A tree is an if-then-else statement that uses the result of a program (a "condition") to decide between returning the result of one out of two subtrees. If the condition is zero, the then-branch of the if statement is returned, otherwise, the else-branch is returned.

Given a computation, we first synthesize a set of programs. Each program in the set will describe some part of the output space of the computation. Once we have synthesized such a set, we start learning a set of *predicates*. Predicates are programs, where we interpret any non-zero return value as true, and a return value of zero as false. We combine the predicates and programs using the decision tree learning algorithm ID3 [16].

Sign	$:=$ *Signed* $\mid$ *Unsigned*
Endianness	$:=$ *Little* $\mid$ *Big*
Input	$:=$ *(I,* Sign*,* Endianness*)*
Const	$:= N$
DNF	$:= (V$ **and** V **and** $\ldots)$ **or** $(V$ **and** V **and** $\ldots)$ **or** $\ldots$
Arith	$:= (V * V * \ldots) + (V * V * \ldots) + \ldots$
Operation	$:= V \ll V \mid V \gg V \mid V/V \mid$ *Parity(V)* $\mid$ DNF $\mid$ Arith
Program	$:= (V =$ Input*;*$)*$ $(V =$ Const*;*$)*$ $(V =$ Operation*;*$)*$
Tree	$:=$ **if** [Program] $=$ *0* **then** Tree **else** Tree **endif** $\mid$ [Program]

Figure 3.4: The grammar that is used for program synthesis. Programs are synthesized, while Trees are learned with a decision tree learning algorithm. I represents an input, N represents any natural number, and V represents a variable. If a variable is being assigned, it must always be a fresh variable.

During synthesis, we further restrict how variables are used in operations. We limit the number of different variables that may appear in a single operation to 4. Additionally, we do not allow the same variable to appear multiple times in a product or a conjunction.

The grammar is designed such that it can efficiently enumerate most of the x86-64 general-purpose register semantics. In practice, the main limitation is the speed at which we can generate and verify new programs. Above 4 or 5 operations, the number of possible programs grows so quickly that it becomes infeasible to enumerate them.

Example 7. *Consider the instruction* `02 04 8a`*, introduced in Example 3, that computes the addition of a register r_1 and a value stored at memory address $r_2 + r_3 * 4$, and stores this result in r_1.*

We described its execution with the following assignments:

1. *Address*$(m_1) = r_2 + r_3 * 4$

2. $r_{IP} = r_{IP} + 3$

3. $r_1 = r_1 + m_1$

The computations for these assignments are:

1. $f_1(x, y) = x + y * 4$

2. $f_2(x) = x + 3$

3. $f_3(x, y) = x + y$

Using our grammar, the first function would be described as:

```
[
    v1 = (i0, Unsigned, Big);
    v2 = (i1, Unsigned, Big);
    v3 = 4;
    v4 = v1 + v2 * v3;
]
```

The second function as:

```
[
    v1 = (i0, Unsigned, Big);
    v3 = 3;
    v4 = v1 + v2;
]
```

And the third as (assuming memory is stored little endian):

```
[
    v1 = (i0, Unsigned, Big);
    v2 = (i1, Unsigned, Little);
    v4 = v1 + v2;
]
```

3.4 Implementation

LibLISA is written in Rust. The project is split into six *crates* (Rust terminology for a package/library):

1. `liblisa-core` contains generic definitions of CPU state, ISAs, encodings, dataflows and other core components of libLISA.

2. `liblisa-enc` contains all code for encoding analysis.

3. `liblisa-synth` contains all code for the synthesis of computations.

4. `liblisa` contains some high-level code for processing encodings as well as code to handle long-running enumeration, synthesis, or validation sessions that can be interrupted at any time.

5. `liblisa-x64` contains all code related to the x64 architecture. It defines registers, flags and CPU state. It also provides implementations for observing the execution of instructions.

6. `liblisa-x64-kmod` contains the kernel module and a wrapper library (see Section 3.4.1).

7. `lisacli` contains code for the command-line (CLI) binaries that can be used to invoke libLISA.

3.4.1 Observations

To observe the execution of instructions, we use the Linux `ptrace` API. Before starting the observation process, we create some pages of shared memory. We then start the observation process. The shared memory pages will be readable and writable from both the observation process and the parent process, making it possible to efficiently read and write memory from and to the observation process.

To make an observation, we preform the following steps:

1. Map memory pages as needed

2. Write the input data to the right location on the memory page

3. Set the registers and flags using `PTRACE_SETREGSET`

4. Execute a single instruction using `PTRACE_SINGLESTEP`

5. Read the resulting CPU state with `PTRACE_GETREGSET` and by reading from the mapped shared memory pages

In order to map the right memory pages in the simple implementation, we must execute the `mmap` syscall from the observation process. To do this, we generate new code on the fly that maps the pages that we want to map, and then write this to the shared memory page mapped as the executable page. Then, using `PTRACE_SETREGSET` and `PTRACE_CONTINUE` we execute the generated code.

`PTRACE_SINGLESTEP` internally uses the x86-64 *trap flag*, which triggers an interrupt after executing a single instruction. This allows us to observe the results after executing just a single instruction, regardless of what that instruction does to the instruction pointer. CPU state is saved automatically by the Linux kernel upon a context switch. The `PTRACE_GETREGSET` syscall simply copies part of this stored state into memory we can access.

If the execution of the instruction fails and triggers a CPU fault, Linux will generate a signal. The `ptrace` API can intercept this signal. For some faults, like a page fault, we can also see the address that was being accessed.

We have two different implementations for making observations. A simple implementation, that just uses the `ptrace` API, and an implementation that requires a kernel module. The kernel module combines all `ptrace` invocations that the simple implementation does into a single `ioctl` call. It also permits us to directly map and unmap memory of the child process. This eliminates two context switches and around a dozen switches between kernel space and user space.

3.4.2 Results

Results are stored by serializing the structures to JSON via the `serde_json` crate. We provide a file containing all encodings that have been enumerated, as well as a file containing the encodings with correctly synthesized computations. The former are stored

as a `Vec<Encoding<X64Arch, BasicComputation>>`, while the latter are stored as a `Vec<Encoding<X64Arch, DecisionTreeComputation>>`.

The easiest way to use the results is to load them using the `serde_json` crate. This can be done as follows:

```
let file = File::open("file.json")?;
let enumerated_encodings: Vec<Encoding<X64Arch, BasicComputation>> =
    serde_json::from_reader(file)?;
```

The specific format of the JSON output depends on the architecture and the kind of computations used. Although we do not provide a specification of the format, it should be possible to export a JSON schema using a crate like `schemars`. This allows the results to easily be used in other programming languages.

Chapter 4

Encoding analysis

In this chapter, we describe our encoding analysis introduced in Section 3.2.1 in more detail. Our encoding analysis serves as a replacement for disassembler libraries. It can learn the encoding and partial semantics of groups of instructions without using any manually provided information. Our goal is to infer encodings as described in Section 3.1, but without computations. Figure 4.1 gives a high-level overview of the encoding analysis.

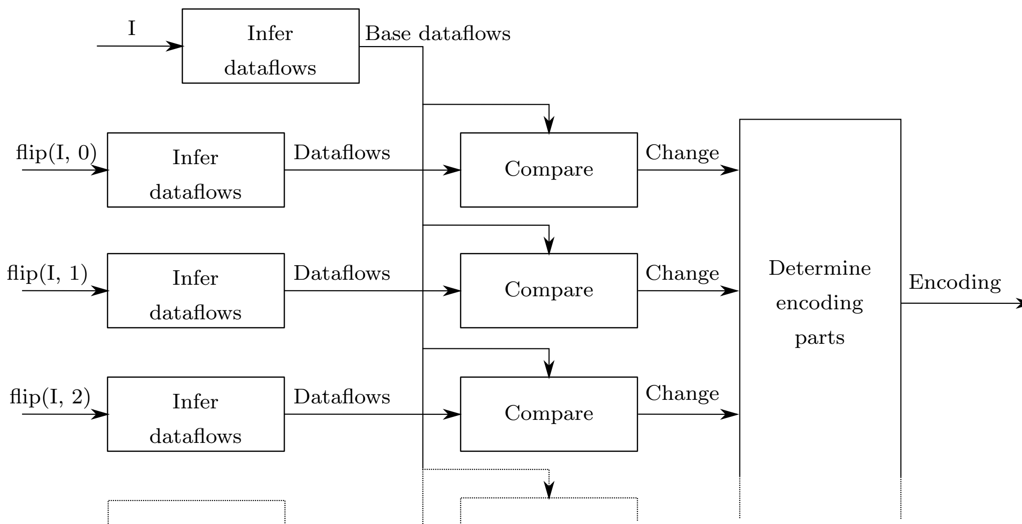

Figure 4.1: Given some instruction I, encoding analysis compares dataflows of I with variants where a single bit has been flipped. $\text{flip}(I, n)$ represents flipping the nth bit in I. The differences are used to identify encoding parts.

We make one simplification specifically and only for encoding analysis. During encoding analysis we assume that if one flag is updated, all flags have been updated. Since at this point we have not identified any immediate values, it would be infeasible to fuzz some output flags correctly. For example, fuzzing the zero flag on a 64-bit comparison with an immediate value (i.e., finding a value for x such that $x = c$ holds given an unknown 64-bit constant value for c) would be infeasible. However, to detect the overflow flag we only need to find an x such that $x + c \geq 2^{64}$, which is much easier. Since both flags are often updated at the same time, we can often correctly determine that ‘some’ flags are updated, but not which ones.

4.1 Fuzzing

We can see a dataflow as a set of *properties*. A property describes behavior of the instruction in terms of the values in storage locations after execution of the instruction, sometimes in relation to the values before execution or after execution of a different instruction. For example, a dataflow from sources $s_1, s_2, .., s_n$ to destination d consists of $n + 1$ properties. One property that describes the fact that d is a destination, and n properties that describe that each of the sources s_i is a source for destination d.

We do not have a specification of CPU instruction behavior, nor are we able to view the hardware design of the CPU. Therefore, we cannot use tools like theorem provers or SMT solvers, which rely on such a specification, to prove properties. The only way prove a property of an instruction would be to exhaustively enumerate the entire *input space* consisting of every possible combination of values for every storage location. This is infeasible for modern CPUs because of the size of the input space.

Since proving properties is impossible, we instead use *fuzzing*. To verify a property, a *fuzzer* enumerates a randomly chosen subset of the input space, trying to find a *counterexample*. If the fuzzer finds a counterexample, we conclude that the property does not hold. If the fuzzer is not able to find a counterexample within a reasonable amount of time, we will assume that the property holds.

When *fuzzing an instruction* (executing a single instruction with many randomly generated values for storage locations), we usually specify properties in such a way that finding a counterexample proves that something exists. This ensures that if the fuzzer runs out of time before finding a counterexample, we do not assume existence of something that may not exist.

4.2 Inferring dataflows

As we saw in Section 3.1, we can model instructions as a set of assignments. We call such an assignment $x_0 = f_0(i_{0,1}, i_{0,2}, \ldots)$ a *dataflow* from *sources* $i_{0,1}, i_{0,2}, \ldots$ to *destination* x_1. For encoding analysis, we will not concern ourselves with learning the computation f_0.

Definition 8. Let I be a valid instruction. The dataflows of this instruction are a tuple of the form $F = \langle M, S, D, \delta, \sigma \rangle$, where:

1. M is an ordered list of memory accesses $[m_1, m_2, \ldots]$;
2. S is a set of sources, as introduced in Section 3.1;
3. D is a set of destinations, as introduced in Section 3.1;
4. δ is an injective function $D \mapsto \{S\}$ that provides for each destination its sources;
5. σ is an injective function that assigns an operand size $l..h$ to each destination (D) or source of a destination ($\langle D, S \rangle$), where $l..h$ indicates that byte indices l up to and including h are read or written.

Example 9. *Consider once again the instruction* `02 04 8a` *that computes the addition of a register r_1 and a value stored at memory address $r_2 + r_3 * 4$, and stores this result in r_1. In Example 3 we saw that we can describe its execution with three assignments:*

1. *Address*$(m_1) = r_2 + r_3 * 4$

2. $r_{IP} = r_{IP} + 3$
3. $r_1 = r_1 + m_1$

The dataflow $F = \langle M, S, D, \delta, \sigma \rangle$ of this instruction is then:

1. $M = [m_1]$
2. $S = [m_1, r_1, r_2, r_3, r_{IP}]$
3. $D = [Address(m_1), r_{IP}, r_1]$
4. $\delta(Address(m_1)) = \{r_2, r_3\}$
5. $\delta(r_{IP}) = \{r_{IP}\}$
6. $\delta(r_1) = \{r_1, m_1\}$
7. $\sigma(x) = 0..7$ *for any* x

Note how the constants (multiplication by 4 in the memory address, and addition of 3 to r_{IP}) have disappeared. Since the constants are part of the computations and are not inputs, we do not include them.

We will express memory accesses, destinations and which sources are used for which destinations as one or more properties. We can then use fuzzing to determine whether these properties hold. In practice, we need to determine dataflows in a particular order, illustrated in Figure 4.2. First, we must determine partial dataflows, consisting only of memory accesses and the dataflows that determine the addresses of these memory accesses. After determining the memory accesses, we can determine other destinations and the sources for each of those destinations, producing one dataflow per destination. The memory accesses together with the dataflows for the other destinations constitute the "complete" dataflows.

Determining the memory accesses has to be the first step in determining the dataflows. When we want to fuzz an instruction, we cannot simply generate some random values for the CPU state and execute that directly on the CPU. If the instruction accesses memory, our randomized states will almost always cause memory faults, as the chance of randomly mapping the right memory location corresponding to the memory location that the instruction will access is very unlikely.

If we do know the memory accesses that an instruction is going to perform, we can easily generate random states that always have the right memory locations mapped. To do this, we execute the instruction, observe the address for which the fault occurred, then map memory at exactly that address, and repeat this process for each of the memory accesses.

4.2.1 Memory accesses

To determine all the memory accesses that an instruction performs, we start by assuming the instruction performs no memory accesses. That is, we assume that the dataflows $F = \langle M, S, D, \delta, \sigma \rangle$ for some instruction I are all empty, i.e., $M = \{\}, S = \{\}, D = \{\}$.

Given this assumption, we now try to find a counterexample to the property "M contains all memory accesses performed by I" by fuzzing. In other words, we try find values for the storage locations, such that the CPU will raise a page fault when executing

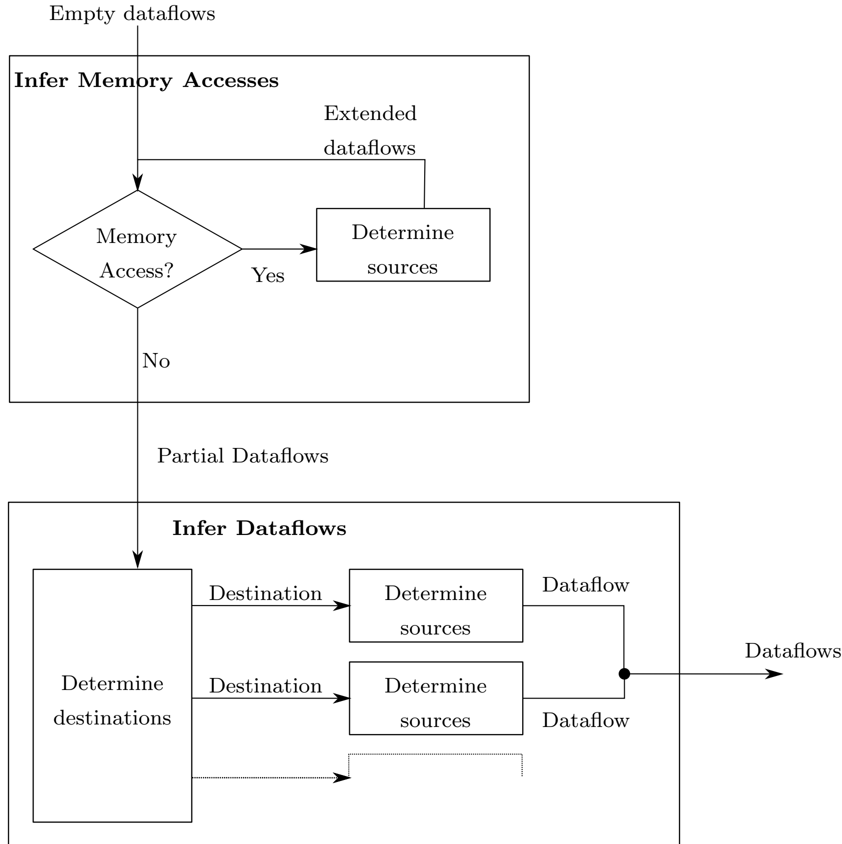

Figure 4.2: In order to infer dataflows, we first iteratively infer memory accesses. Next, we determine the remaining destinations and infer sources for each of these.

the instruction. If we are able to find this, our memory accesses are incomplete. We determine the sources for the address of this memory fault using the approach described in Section 4.2.2, and extend M, S, D and δ with this new information. We repeat this process until fuzzing concludes that M contains all memory accesses.

Remark. *We can make the process of finding the inputs of a memory access more efficient. Memory addresses are often of the form* $i_0 + i_1 * c + i_2 + \ldots$*, i.e., the sum of all inputs where one input is scaled by a constant factor. We can greatly improve performance by directly trying to find such an expression for every memory access. Finding such an expression directly gives us all the sources used to compute the memory address, which is faster than determining the sources using fuzzing as described in Section 4.2.2.*

Normally, we can generate a valid CPU state for an instruction by iteratively observing the memory access error, and then mapping memory at the observed address. This is relatively slow, because making an observation involves multiple context switches (see Section 3.4.1). If we have an expression for the memory address, we can skip this process and map the memory at the right address without making any observations.

In libLISA, our implementation, we have seen this "fast path" be up to 50x faster than the normal path. Additionally, there is less randomness involved in this process, so the fast path is more accurate in some cases. By falling back to the normal approach if the fast path fails, we are still able to analyze instructions that do not conform to the restricted form we described here.

4.2.2 Normal dataflows

To find all destinations D, we once again use fuzzing. To figure out if a certain storage location d is a destination, we are looking for a counterexample to the property "d is *not* modified while executing the instruction". If we find such a counterexample, d must be a destination.

Given a destination d, we use fuzzing to determine its sources. More specifically, for each storage location s we are looking to find a counterexample to the property "storage location s is *not* a source for destination d". To find such a counterexample, we must find two input states that differ only for s, which after execution gives us two output states where d does not have the same value. All s for which we are able to find such a counterexample are sources for destination d.

After identifying the sources and destinations, we fuzz the instruction to determine the operand sizes. We are interested in finding the smallest range of bytes in the storage location that includes all bytes that get modified. For a destination d, we are trying to find a range that includes all bytes for which we cannot find a counterexample to the property "the nth byte of storage location d can be modified by instruction I". For a source s for destination d, we aim to find a range that includes all bytes for which we cannot find a counterexample to "when modifying the nth byte of storage location s, the storage location d might change after execution of instruction I".

4.3 Inferring encodings

An instruction bit sequence consists of several parts, such as the opcode (which indicates which operation is being applied), the instruction operands, prefixes, and in some cases even unused bits. We aim to derive a classification of individual bits based on which part they belong to. Specifically, we are interested in bits that belong to instruction parts that constitute a register, operand size indicator, immediate value or immediate address. Registers, immediates and operands are often encoded in specific bits. We see this with x86-64, but also with other architectures, for example ARM, ARM64 and RISC-V.

Definition 10. Let $[b_0, b_1, \ldots, b_{n-1}]$ be an instruction of length n, which is decodable.

An *encoding* is a tuple $E = \langle P, \varepsilon, \gamma \rangle$, where:

1. $P = [p_0, p_1, p_2, \ldots]$ is a list of instruction parts that constitute some kind of value;

2. ε is a surjective function $\mathbb{N} \mapsto P$ mapping a bit index $i \leq n$ to a part;

3. γ is a function $(P \mapsto \mathbb{N}) \mapsto F$ that takes a mapping of parts to values, and produces the dataflows F corresponding to this mapping.

Example 11. *Consider the instruction `02 04 8a` that computes the addition of a register r_1 and a value stored at memory address $r_2 + r_3 * 4$, and stores this result in r_1.*

In binary, the instruction would be `00000010 00000100 10001010`. Suppose bits 11 and 12 (numbering the first bit 0) indicate which of the four registers r_1, r_2, r_3, r_4 are used to store the result of the addition, such that `00` indicates r_1, `01` indicates r_2, etc.

One valid encoding of this function would be $E = \langle P, \varepsilon, \gamma \rangle$, where:

1. $P = [p_1]$

2. $\varepsilon(11) = \varepsilon(12) = p_1$

3. $\gamma(\{p_1 \mapsto 0\})$ *is the dataflow shown in Example 9,* $\gamma(\{p_1 \mapsto 1\})$ *is the dataflow shown in Example 9 but with destination* r_1 *replaced with* r_2*, etc.*

Less formally, we can write this encoding as `00000010 000aa100 10001010`*. Based on the information we provided in this example, this encoding contains all possible parts. Usually, instruction encodings will consist of multiple parts. For example, we would expect to find parts not only for the destination register* r_1*, but also for the two source registers* r_2 *and* r_3*. Note however, that there is no guarantee that these parts must exist.*

There are multiple valid encodings. Another encoding would be $E = \langle P, \varepsilon, \gamma \rangle$*, where:*

1. $P = []$
2. *the domain of* ε *is empty*
3. $\gamma(\{\})$ *is the dataflow shown in Example 9*

When there are multiple encodings possible, we are aiming to find an encoding that assigns parts to as many bits as possible, as this speeds up enumeration and synthesis.

4.3.1 Comparing dataflows

To identify bits that belong to a part, we use *changes*. Since bits often serve a single purpose (e.g., a bit usually either determines a register or an immediate value, but not both), we can observe a single change when we flip a single bit. This occurs often enough that we will ignore cases where multiple changes occur at the same time.

We can identify changes to sources, destinations and operand sizes by looking at the dataflows. Changes to immediate values can be identified by fuzzing the original instruction and the instruction with a single bit flipped. We will discuss the former in Section 4.3.2 and the latter in Section 4.3.3.

Definition 12. Let $F_1 = \langle S_1, D_1, M_1, \delta_1, \sigma_1 \rangle$ and $F_2 = \langle S_2, D_2, M_2, \delta_2, \sigma_2 \rangle$ be dataflows. A *change* between F_1 and F_2 is one of:

1. None
2. RegisterDifference(locations, $r_{\text{from}}, r_{\text{to}}$) where r_{from} and r_{to} are elements in $S \cup D$ and locations are change locations
3. SizeDifference(locations, $a_{\text{from}}, a_{\text{to}}$) where a_{from} and a_{to} are ranges of byte indices of the form $l..h$ with $l \leq h$, and locations are change locations
4. ImmDifference(locations)
5. MemoryError(m) where $m \in M_1 \cup M_2$
6. Multiple

Changes can be combined. If the changes are equal except for location, a combined change is formed by taking the union of both sets of locations. Otherwise, the combined change is Multiple. Combining the changes in an empty set gives None.

Changes occur at a specific *change location*. We do not want to assign changes that occur at different locations to the same part in an encoding, as they are not identical. Additionally, we want all changes of the same type in an encoding to be *independent*. That is, none of the change locations may overlap. This allows us to instantiate encodings efficiently.

Definition 13. Let $F_1 = \langle S_1, D_1, M_1, \delta_1, \sigma_1\rangle$ and $F_2 = \langle S_2, D_2, M_2, \delta_2, \sigma_2\rangle$ be dataflows.

A *change location* for a change between F_1 and F_2 is either a destination $d \in D_1$ or a tuple $\langle d, s\rangle$ where $d \in D_1$ and $s \in S_1$. A single destination indicates that the output was changed. For example, when a destination d changes into d', it occurs "at d". A tuple $\langle d, s\rangle$ indicates that the source s of the output d was changed.

We are interested in comparing the changes between some base instruction, and a modified version of that instruction. The base instruction I_1 will provide the "from" value of a change, while the modified instruction I_2 will provide the "to" value. A change of None means that both instructions behave identically. A change of RegisterDifference or SizeDifference means that a single register or operand size (which may be used in multiple dataflows) changed into another register or operand size. An ImmDifference indicates that there is a difference in the values in destinations after execution of the instruction.

The last two changes, MemoryError and Multiple, are special cases. During the computation of changes we might encounter a memory error. For example, an instruction might attempt to access memory that we cannot physically map on the CPU. If we encounter such a memory error for memory access m, we return a MemoryError(m) instead. Multiple is used for all cases where there is more than one change, or where changes occur that we cannot represent with one of the other changes.

Example 14. *In Example 11, we showed how a part could be constructed if we* somehow *knew that bits 11 and 12 determined the register of one of the destinations. We can know this by looking at the changes. In the case of Example 11, we would find the changes RegisterDifference*($\{r_1\}, r_1, r_2$) *and RegisterDifference*($\{r_1\}, r_1, r_3$) *for bits 11 and 12. These changes can be in the same part, because the locations and* r_{from} *are equal.*

We will discuss the translation from changes into parts in Section 4.4.

Definition 15. We define changes(I_1, I_2) to be the set of changes observed when comparing the instruction I_1 with a modified variant I_2.

4.3.2 Changes in dataflows

Let $F_1 = \langle S_1, D_1, M_1, \delta_1, \sigma_1\rangle$ for I_1 and $F_2 = \langle S_2, D_2, M_2, \delta_2, \sigma_2\rangle$ for I_2. Additionally, let $d_1 \in D_1$ and $d_2 \in D_2$ such that either $d_1 = d_2$, or $d_1 \notin D_2 \wedge d_2 \notin D_1$. This pairs dataflows from the original instruction to their possible counterparts for the modified instruction.

For destinations that occur in both D_1 and D_2, $d_1 = d_2$ must hold. For destinations that occur in either D_1 or D_2, but not both, all combinations are possible. If a single destination changed, there will only be one possibility where $d_1 \neq d_2$. If more than one destination changed, the bit does not belong to a part, because bits in a part must have a single purpose. In that case, we just want the final result to be Multiple. This happens implicitly, since we will produce multiple conflicting changes if there is more than one possibility where $d_1 \neq d_2$, .

First, we consider changes to registers. We might either find a change in destination registers (*reg.d*), or in source registers of a certain destination (*reg.s*). If, when comparing

the dataflows of I_1 to I_2, we see exactly one source being removed and one source being added, this suggests that the source that was removed has changed into the source that was added. The same applies when one destination is removed and one other destination is added.

$$\text{reg.d}\frac{d_1 \neq d_2 \qquad \text{is_reg}(d_1) \qquad \text{is_reg}(d_2)}{\text{RegisterDifference}([d_1], d_1, d_2) \in \text{changes}(I_1, I_2)}$$

$$\text{reg.s}\frac{\begin{array}{ccc} r_{\text{from}} \in \delta_1(d_1) & r_{\text{from}} \notin \delta_2(d_2) & \text{is_reg}(r_{\text{from}}) \\ r_{\text{to}} \notin \delta_1(d_1) & r_{\text{to}} \in \delta_2(d_2) & \text{is_reg}(r_{\text{to}}) \end{array}}{\text{RegisterDifference}([\langle d_1, r_{\text{from}} \rangle], r_{\text{from}}, r_{\text{to}}) \in \text{changes}(I_1, I_2)}$$

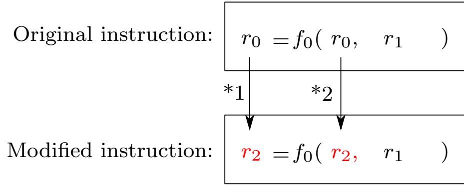

Figure 4.3: Two changes are depicted. *1 represents a *register change* from r_0 to r_2 in change location "output r_0", and *2 represents a register change from r_0 to r_2 in change location "input r_0 of output r_0". These two changes combine, because both are register changes with the same original and new register.

Example 16. *Consider some instruction I_1 that performs an addition $r_0 = r_0 + r_1$. This is a dataflow from sources r_0, r_1 to destination r_0. Let I_2 be a modified version of I_1 such that r_0 changes into r_2, i.e., I_2 performs an addition $r_2 = r_2 + r_1$. This is illustrated in Figure 4.3.*

Both rules reg.d *and* reg.s *apply. First, we can find one combination of d_1 and d_2 where rule* reg.d *applies (this also happens to be the only possible combination, since $|D_1| = |D_2| = 1$):*

$$\textit{reg.d}\frac{r_0 \neq r_2 \qquad \textit{is_reg}(r_0) \qquad \textit{is_reg}(r_2)}{\textit{RegisterDifference}([r_0], r_0, r_2) \in \textit{changes}(I_1, I_2)}$$

Next, we can also find one instance where rule reg.s *applies:*

$$\textit{reg.s}\frac{\begin{array}{ccc} r_0 \in \{r_0, r_1\} = \delta_1(r_0) & r_0 \notin \{r_2, r_1\} = \delta_2(r_2) & \textit{is_reg}(r_0) \\ r_2 \notin \{r_0, r_1\} = \delta_1(r_0) & r_2 \in \{r_2, r_1\} = \delta_2(r_2) & \textit{is_reg}(r_2) \end{array}}{\textit{RegisterDifference}([\langle r_0, r_0 \rangle], r_0, r_2) \in \textit{changes}(I_1, I_2)}$$

Since both of these differences are of the same type and share the same original and new register, we can combine them into: $\textit{RegisterDifference}([r_0, \langle r_0, r_0 \rangle], r_0, r_2)$.

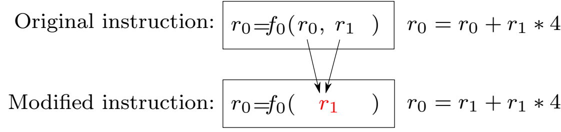

Figure 4.4: Register r_0 is 'folded' onto r_1 in the dataflows, because we cannot observe that the same register is being used twice in the computation.

Another case to consider is what happens when a register is changed into a register that is already present in the sources. For example, if $r_0 = f_0(r_0, r_1) = r_0 + r_1 * 4$ were to change into $r_0 = f'_0(r_1) = r_1 + r_1 * 4$ as illustrated in Figure 4.4. In this case, we will see the sources change from $\{r_0, r_1\}$ to just $\{r_1\}$. We will only see that a source (r_0 in this example) was removed, but we will not see another source being added.

Just by looking at the dataflows, we cannot determine if such a change is occurring, and which register would be r_{to} of a RegisterDifference. We use fuzzing to try and find a register for which this is the case. For example, for Figure 4.4 we would try to find a counterexample to $f_0(r_1, r_1) = f'_0(r_1)$. Let R be all possible registers for which we could not find a counterexample. If and only if R contains exactly one element, we can conclude that a RegisterDifference exists (*reg.fold*). In any other case, we cannot be sure of what is happening, and we consider the change Multiple (*reg.nfold*).

$$\text{reg.fold}\ \frac{\begin{array}{c}\text{from} \in \delta_1(d_1) \qquad \text{from} \notin \delta_2(d_2) \\ |\delta_2(d_1) - \delta_1(d_1)| = 0 \qquad \text{to} \in R \qquad |R| = 1\end{array}}{\text{RegisterDifference}([\langle d, \text{from}\rangle], \text{from}, \text{to}) \in \text{changes}(I_1, I_2)}$$

$$\text{reg.nfold}\ \frac{\begin{array}{c}\text{from} \in \delta_1(d_1) \qquad \text{from} \notin \delta_2(d_2) \\ |\delta_2(d_1) - \delta_1(d_1)| = 0 \qquad |R| \neq 1\end{array}}{\text{Multiple} \in \text{changes}(I_1, I_2)}$$

Finally, we introduce two rules to cover differences in operand sizes for sources (*size.s*) and destinations (*size.d*). These work analogous to *reg.d* and *reg.s*, except that they also work for non-register storage locations.

$$\text{size.s}\ \frac{\text{s} \in \delta_1(d_1) \cup \delta_2(d_2) \qquad \sigma_2(\langle d_1, s\rangle) \notin \sigma_1(\langle d_2, s\rangle)}{\text{SizeDifference}([\langle d, s\rangle], \sigma_1(s), \sigma_2(s)) \in \text{changes}(I_1, I_2)}$$

$$\text{size.d}\ \frac{\sigma_2(d_2) \in \sigma_1(d_1)}{\text{SizeDifference}([d_1], \sigma_1(d_1), \sigma_2(d_2) \in \text{changes}(I_1, I_2)}$$

We cannot represent changes between registers and memory locations or between two memory locations. We would therefore like all of these cases to produce the change Multiple. We introduce two rules to invalidate differences that involve any source or destination that is not a register. The rule *nonreg.s* covers differences in sources, and the

rule *nonreg.d* covers differences in destinations. These rules are necessary because the rules *reg.d* and *reg.s* do not apply if the change is not between two registers. We use the symmetric difference operator Δ for brevity.

$$\text{nonreg.s}\frac{\text{x} \in \delta_1(d_1) \bigtriangleup \delta_2(d_2) \qquad \neg\text{is_reg(x)}}{\text{Multiple} \in \text{changes}(I_1, I_2)}$$

$$\text{nonreg.d}\frac{\text{d} \in D_1 \bigtriangleup D_2 \qquad \neg\text{is_reg}(d)}{\text{Multiple} \in \text{changes}(I_1, I_2)}$$

Example 17. *Consider once again the instruction I_1 = `02 04 8a` that computes the addition of a register r_1 and a value stored at memory address $r_2 + r_3 * 4$, and stores this result in r_1, introduced in Example 3.*

We showed that its execution can be described with the following assignments:

1. $Address(m_1) = r_2 + r_3 * 4$
2. $r_{IP} = r_{IP} + 3$
3. $r_1 = r_1 + m_1$

Imagine a certain bit flip to instruction I_2 would give an instruction that behaved as follows:

1. $r_{IP} = r_{IP} + 3$
2. $r_2 = r_1 + r_2$

This variant performs no memory access, and stores its results in r_2 instead of r_1. When determining $changes(I_1, I_2)$, we consider all of the following possibilities for d_1 and d_2:

1. $d_1 = d_2 = r_{IP}$
2. $d_1 = Address(m_1), d_2 = r_2$
3. $d_1 = r_1, d_2 = r_2$

For the first possibility, we are not able to apply rule nonreg.s*, since $\delta_1(r_{IP}) \bigtriangleup \delta_2(r_{IP}) = \emptyset$. For the second possibility, we also cannot apply* nonreg.s*, since $\delta_1(Address(m_1)) \bigtriangleup \delta_2(r_2)$ contains only registers. Finally, for the third possibility we are able to apply* nonreg.s*:*

$$nonreg.s\frac{m_1 \in \delta_1(r_1) \bigtriangleup \delta_2(r_2) \qquad \neg is_reg(m_1)}{Multiple \in changes(I_1, I_2)}$$

The rule nonreg.d *also applies. Since $D_1 \bigtriangleup D_2 = \{Address(m_1), r_1, r_2\}$, we can apply the rule as follows:*

$$\text{nonreg.d}\frac{\mathit{Address}(m_1) \in D_1 \triangle D_2 \qquad \neg \mathit{is_reg}(\mathit{Address}(m_1))}{\mathit{Multiple} \in \mathit{changes}(I_1, I_2)}$$

Note that from *nonreg.d* it also follows that if memory accesses M_1 and M_2 are not identical, Multiple $\in$ changes(I_1, I_2).

4.3.3 Changes in output values

Since we do not include constant values in dataflows, we cannot observe a change in such a constant value by comparing only the dataflows. A constant value in the specification of an instruction is usually an immediate value. If an immediate value changes, we can observe a difference in the destinations after execution of the instruction. More specifically, given an original instruction I_1 and a modified instruction I_2 we fuzz both instructions to find changes in immediate values. For every destination d, we try to find a counterexample such that the value of d after execution of I_1 is not the same as the value of d after execution of I_2. If we are able to find such a case, then ImmDifference$([d]) \in$ changes(I_1, I_2).

In practice, we also need to take into account all previous changes that we have found. For example, if we know an input for some output changes from r_1 to r_2, we need to adapt the values during fuzzing to make sure we do not get false positives.

4.4 Encoding parts

Finally, we can build an encoding by looking at the change produced by flipping each bit in the instruction.

Definition 18. Given some instruction I from which we are going to build an encoding, let change$(n) = \bigvee$ changes$(I, \text{flip}(I, n))$, i.e., the combination of all changes observed when flipping bit n of I. The $\bigvee$ operator combines all changes together into a single change, as described in Definition 12.

We are purposefully using an approach that is not guaranteed to always be correct. For example, if fuzzing cannot find a counterexample before a certain deadline, we will assume no counterexample exists. This is necessary to make enumeration and synthesis feasible. It may also cause some noise in the changes, which we want to filter out. This means that it might not be desirable to include some inferred changes in the final encoding.

We classify bits into *parts* based on the changes, and (non-)equivalence between changes. Two changes are equivalent if they are either identical except for r_{to} or a_{to}, or if one change is a MemoryError(m) and the other change is an ImmDifference(locations) such that $m \in$ locations.

For register changes and operand size changes, we do not expect the bits corresponding to the changes to be consecutive. All bits belonging to equivalent changes may become a part. For immediate values, we are specifically looking for sequences of consecutive equivalent changes.

Definition 19. Given some instruction $I = [b_0, b_1, \ldots, b_n]$, let c be a change found by flipping some bit in I. If c is either a RegisterDifference or a SizeDifference, then a part may be formed with all bit indices n for which change(n) is equivalent to c.

If c is an Imm, then a sequence of k bits with indices $S = \{m, m+1, m+2, \ldots, m+k-1\}$ may become a part, if the sequence adheres to the following three rules:

1. Given two indices $i \in S, j \in S$, change(i) must be equivalent to change(j);
2. No more than 1/4th of all changes in S may be a MemoryError;
3. The sequence must be at least 2 bits long, i.e., $|S| \geq 2$.

We do not expect to classify all bits into a part, nor do we expect to classify all bits that have a change other than Multiple or None to be in a part. Instead, we aim include as many bits as possible in the parts. Each bit that we are able to add to a part, doubles the number of instructions that we can skip during enumeration. We require all parts to be *independent* from any other part. In order for parts to be independent, two different parts may not contain changes of the same type with (partially) overlapping locations. Because of this restriction, not all parts that *may* be formed, *can* be formed. Usually, most parts will be independent. For parts that are dependent, we choose the part that consists of the most bits.

Example 20. *Consider an instruction* `0010 0101`. *We identify the following changes for each of the bits (left-to-right) by comparing the original instruction with a modified instruction where we flipped that bit:*

1. *None*
2. *None*
3. *ImmDifference*($[d_1]$)
4. *Multiple*
5. *RegisterDifference*($[\langle d_1, s_1\rangle, d_1]$)
6. *RegisterDifference*($[d_1]$)
7. *RegisterDifference*($[\langle d_1, s_1\rangle, d_1]$)
8. *RegisterDifference*($[\langle d_1, s_1\rangle, d_1]$)

We will not be able to form a part with bit 3 (counted from the left, starting at 1). In order for the immediate change to be a part, there would need to be at least two consecutive equivalent changes. We also cannot form a part with the change for bit 6. While there is no requirement on the number of consecutive equivalent changes for registers, the locations conflict with those of the changes in bits 5, 7 and 8. Leaving out this bit allows us to include three other bits, which allows us to classify more bits into parts. Finally, bits 5, 7 and 8 can form a part. Note that, had we not already excluded bit 3 because of sequence length, the locations of bits 5, 7 and 8 would not conflict with those of bit 3, because the changes are of a different type (RegisterDifference versus Imm).

The final classification ends up consisting of just one part:

1. $P = \{p_1\}$
2. $\varepsilon = \{5 \mapsto p_1, 7 \mapsto p_1, 8 \mapsto p_1\}$

Less formally, we could write this as `0010 a1aa`.

As an optimization, None changes could be translated into *"don't care" bits.* "Don't care" bits are bits that do not belong to a part, but may be 0 or 1 without affecting the semantics of an instruction. For example, it would allow us to classify the bits in the instruction in Example 20 as `__10 a1aa`, where '`_`' indicates a "don't care" bit.

4.5 Instantiation

For γ, the function that instantiates an encoding into dataflows for a specific instruction, we can look at *part-wise* changes to update the dataflows we already inferred, rather than inferring new dataflows for each instantiation. When instantiating, we consider each part separately. Given some part p, we determine the change between the original instruction, and the instruction with the bits belonging to part p replaced with the newly assigned value. Once we have done this for every part, we can easily construct the new dataflow by applying the changes to the original dataflow one-by-one. We required that parts are independent, i.e., they not contain overlapping locations. Because of this, we will not have any overlapping changes when performing this procedure.

Example 21. *Consider an instruction `00000010 00000100 10bbbaaa`. For the case where $\underline{\texttt{aaa}} = \texttt{010}, \underline{\texttt{bbb}} = \texttt{001}$ (i.e., the instruction we saw in Example 3) we know that there are three dataflows (written as "destination $\Leftarrow$ sources" for brevity):*

1. $Address(m_1) \Leftarrow r_2, r_3$
2. $r_{IP} \Leftarrow r_{IP}$
3. $r_1 \Leftarrow r_1, m_1$

We would like to instantiate this instruction for $\underline{\texttt{aaa}} = \texttt{110}, \underline{\texttt{bbb}} = \texttt{000}$. For this, we are going to look at the part-wise change compared to our base instruction, for a and b separately.

First, we compute the change for `a`. Comparing our base instruction with the instantiation $\underline{\texttt{aaa}} = \texttt{110}, \underline{\texttt{bbb}} = \texttt{001}$, gives $RegisterDifference(r_2, r_6, \{\langle Address(m_1), r_2\rangle\})$. Next, we compute the change for `b`. Comparing our base instruction with the instantiation $\underline{\texttt{aaa}} = \texttt{010}, \underline{\texttt{bbb}} = \texttt{000}$, gives $RegisterDifference(r_3, r_0, \{\langle Address(m_1), r_3\rangle\})$.

We can now construct the full dataflows for the instruction. We replace r_2 in destination $Address(m_1)$ with r_6, and replace r_3 in destination $Address(m_1)$ with r_0. Because parts are independent, we can make these changes without worrying about overlap. The final dataflows for the instantiated instruction, `00000010 00000100 10000110` are:

1. $Address(m_1) \Leftarrow r_6, r_0$
2. $r_{IP} \Leftarrow r_{IP}$
3. $r_1 \Leftarrow r_1, m_1$

Chapter 5

Evaluation

To evaluate our approach, we ran our algorithm on a Ryzen 3900X CPU. We determine the *completeness* of our enumeration and the *correctness* of our learned semantics results. By completeness, we mean the ratio of instructions that we find during enumeration out of all supported instructions on our CPU. By correctness, we mean the ratio of instructions for which we have been able to synthesize semantics that produce the same results as the real CPU implementation. We also compute the *coverage* of our semantics, i.e. the ratio of all supported instructions on our CPU that we are able to correctly describe with the learned semantics. Additionally, we searched for undocumented instructions by comparing the encodings we found with the Intel XED disassembler library.

Comparisons to existing work are difficult. We do not use a disassembler library as the basis from which we learn semantics. Therefore, our encodings do not necessarily map to disassembler libraries in a logical manner. This makes direct comparisons with related work difficult. The authors of the formalization of all non-deprecated x86-64 user space instructions are working on formalizing instruction decoding [4], which would make comparisons much easier.

Compared to the manual semantics specified by Dasgupta et al. [4], we also enumerate and try to synthesize deprecated instructions. Since we do not want to rely on manually specified information, we cannot use a list of deprecated instructions that we can ignore. Because of this, it is necessary to enumerate even the deprecated instructions.

We consider all instructions operating only on general-purpose registers and flags in Linux x86-64 user space to be in scope. While we do enumerate over all instructions, our implementation does not support observing other register sets like floating point registers or vector registers. We therefore cannot learn dataflows for other registers.

Our evaluation is based on 322 hours of enumeration, and 44 hours of synthesis. This was not enough time to completely enumerate the x86-64 instruction set. We estimate to have enumerated 36.13% of the x86-64 instruction set. We present the partial results we obtained.

5.1 Enumeration completeness

We aim to determine how many instructions we identify, out of all instructions supported on the CPU. Determining all supported instructions on a CPU is difficult. Disassembler libraries can encode or decode a byte sequence, but cannot *iterate* over all possible encodings. The Intel XED disassembler library contains a grammar that describes instruction encoding, but using that grammar to iterate over all possible instructions is non-trivial.

To determine completeness, we compare the encodings we have found to randomly found instructions, as well as instructions found in the Linux binaries `gcc`, `ls`, `grep`, `perl`, `ssh` and `libxul` (a Firefox library). We choose these binaries because of their relatively big size, as well as their popularity. This allows us to determine how useful our enumerated encodings would be for real-world usage.

To build a list of randomly found instructions, we use randomly generated byte sequences. Because of the huge number of x86-64 instructions, we cannot generate a full list of every instruction. Whenever the byte sequence is decodable on our CPU, we tunnel until the instruction length changes. We add the last instruction we saw before the length changed to our list of instructions. This approach keeps the number of instructions in the list relatively small, while still exploring the full instruction space. We generated a list of 4 million unique instructions on the 3900X.

To extract instructions from Linux binaries, we load the `.text` section of each of the binaries, and extract all instructions using XED. We try to remove duplicate instructions from the list. We consider two instructions that are bit-for-bit identical to be duplicates.

We check completeness for each of the instruction lists separately. We do not filter out-of-scope instructions from the lists. Out-of-scope instructions are, for example, floating point instructions, SIMD instructions, instructions that use segment registers, or instructions that require elevated privileges.

We list completeness for each of the instruction lists in Table 5.1. We enumerate instructions in lexicographical order. Since we could not enumerate the entire instruction space in 322 hours, some instructions have not yet been enumerated. We enumerated instructions up to some instruction C (the "cursor"). We consider an instruction I to be *seen* if $I < C$. We consider I to be *unseen* if $I \geq C$. We will ignore the unseen instructions, and extrapolate from the seen instructions.

Within the seen instructions, we distinguish between *found* and *missed* instructions. Given an instruction I from an instruction list, we consider the instruction found if we can instantiate the dataflows for this instruction from an encodings we found. Missed instructions are all other seen instructions. We express the completeness as $\frac{\text{found}}{\text{found+missed}} * 100\%$.

Missing instructions might exist because we were unable to analyze it, or because enumeration incorrectly skipped it. We might not be able to analyze an instruction if it requires additional privileges, or if it accesses memory in a way we cannot handle. We might skip an instruction during enumeration because of tunneling. We give an example of how this might happen in Section 2.1. In practice, this rarely happens.

Source	Total	Seen	Found	Missed	Completeness
scan	4655355	125466	124461	1005	99.20%
gcc	42749	3055	3054	1	99.97%
grep	13058	457	457	0	100.00%
ls	8362	347	347	0	100.00%
perl	83501	2878	2877	1	99.97%
ssh	36785	1285	1285	0	100.00%
libxul	1616659	43751	43750	1	100.00%

Table 5.1: The completeness of our enumeration.

The encodings that we learned represent many more instructions than the numbers

listed under ‘found’ in Table 5.1. Encodings can contain instructions that have not been seen. For example, we might learn an encoding `00a0 0000` when our cursor C is at `0000 0000`. This encoding also covers the instruction `0010 0000`, which is past our cursor C and therefore unseen. The largest part of the instructions covered by the encodings are unseen. As enumeration progresses, the number of found instructions will start to increase exponentially, since an increasing number of instructions that we consider will already be covered by an encoding. While the table might suggest that only a few percent of all instructions has been found, in reality this is around 36.13%. We do not count these unseen instructions as found, because we do not know how many additional instructions we would need to count as missed. Therefore, counting these instructions would skew the completeness.

5.2 Semantics correctness

Determining whether our learned semantics are correct is complicated. Disassembler libraries do not provide enough information to allow us to verify semantics. Even for dataflows, disassembler libraries are not precise enough. A disassembler library only provides a list of operands, but does not tell us how those operands are used. This causes many false positives. For example, consider an instruction that computes the bitwise OR of a register r and 0. Our encoding analysis will learn that this instruction does nothing (except for possibly some flag changes). This is technically correct. However, a disassembler library will tell us that the instruction reads from r and writes a result back to r.

To avoid these complications we use fuzzing to verify our learned semantics. Fuzzing has a number of drawbacks. Most importantly, we cannot verify what we cannot observe. For example, since we do not observe SIMD registers, we will not be able to observe that all SIMD instruction semantics that we learn are missing SIMD registers in their dataflows. To compensate for this, we exclude any instruction that uses registers that we cannot observe using XED. Fuzzing also cannot *prove* correctness. We can only show that it is likely that we have found the correct semantics. Lastly, fuzzing only allows us to compare our semantics to “reality”. It would be more interesting to compare our semantics to the semantics used in, for example, reference manuals, emulators or compilers, to verify their correctness.

With fuzzing, we also cannot differentiate between undefined behavior and defined behavior. If we cannot synthesize an expression for a dataflow, we assume that we have not learned the correct semantics. In reality, the value might be undefined or defined to be random (for example `RDRAND` on x86-64).

More specifically, by “fuzzing to verify correctness” we mean the following: we take an encoding, and instantiate it with random values for all parts. We then generate a random CPU state for the instantiated encoding. We use the synthesized computations to determine where to place data for memory accesses. We execute the instruction, and observe the result. If the instruction executed successfully, we verify the result. For each storage location, we check if the output value matches the value we expected based on the synthesized computation. If the output value does not match the expected value, we consider the entire encoding incorrect. We repeat this process many times for all encodings.

From the encodings that we learned during enumeration, we generated 99789 computations. We ran synthesis for 44 hours, and were able to synthesize semantics for 60826 computations. Given these computations, we could construct full semantics for

4395 encodings out of 25373 encodings. We used fuzzing to verify the correctness of these semantics, and found that semantics for 4297 encodings were correct.

When encoding analysis fails, it often produces more separate encodings than when encoding analysis is successful. There are usually around 8x more encodings if encoding analysis did not work perfectly, although this varies a lot. Because of this issue, the percentage of correct encodings is not a good indication of the correctness.

To compute the correctness, we use the instruction lists from Section 5.1. Since fuzzing cannot verify the correctness of instructions that use registers we cannot observe, we use XED to filter out all instructions that are out of scope. For each in-scope instruction, there are three possibilities: we did not synthesize semantics (because we did not have enough time, we did not enumerate the instruction, or because synthesis failed), we synthesized correct semantics or we synthesized incorrect semantics. We express the correctness as $\frac{\text{correct}}{\text{corrrect+incorrect}} * 100\%$. This gives an indication of whether we can trust synthesis to synthesize only correct semantics. We present the results in Table 5.2

Because we consider all instructions in an encoding incorrect if one instruction in the encoding is incorrect, the correctness that we compute is an under-approximation. Unfortunately it is difficult to better approximate the real correctness. Determining whether the semantics are correct for each instruction separately would be computationally very expensive.

Source	Correct	Incorrect	Not synthesized	Out of scope	Correctness
scan	18229	107	95181	11949	99.42%
gcc	76	0	2672	307	100.00%
grep	32	0	324	101	100.00%
ls	14	0	315	18	100.00%
perl	124	0	2489	265	100.00%
ssh	54	0	1004	227	100.00%
libxul	1283	0	20780	21688	100.00%

Table 5.2: The correctness of our semantics, expressed as the percentage of instructions that have a correct encoding. We do not have full semantics available for many encodings, because synthesis ran for only 44 hours. The correctness is an under-approximation.

We manually analyzed around a hundred incorrect encodings, and determined the cause of each incorrect encoding. We found that in many cases, it is impossible for our encoding analysis to examine all possible values for a storage location. For example, the x86-64 instruction `mov QWORD PTR [rdi+0x0],rdi` is one such case. This instruction copies 64 bits of data stored in register `rdi` to the memory address computed by `rdi+0x0`. Because `rdi` is used to access memory, it must be a valid address. On the 3900X and many other x86-64 CPUs, valid addresses must have all upper 17 bits either set or unset. Because Linux reserves addresses with the highest bit set for kernel usage, we can only set the lower 47 bits of `rdi`, and must keep the upper 17 bits unset. This means that we cannot detect a dataflow from the upper 17 bits of `rdi` into the memory location, and will assume only the lower 47 bits of `rdi` are used. This problem could be solved by using a bare-metal process, which does not have to adhere to constraints imposed by Linux. We explain this in more detail in Section 6.2.

We did not find any encodings where the synthesized computations were incorrect. This is expected: the program synthesis technique we use makes many observations and

finds a program that produces the correct output for all observations. This process could be considered very similar to fuzzing. To verify the correctness of the individual computations, a direct comparison to existing semantics would be more useful. Our correctness validation mainly validates whether encoding analysis grouped instructions into encodings correctly, whether it found correct dataflows, and whether the integration of synthesized computations into an encoding produces correct semantics.

5.3 Coverage

Correctness and completeness both describe the effectiveness of the individual enumeration and synthesis stages of our approach. To determine how useful our semantics currently are for real-world use cases, we also compute the *coverage* of our semantics. When writing, for example, an analysis tool, the coverage indicates for what percentage of instructions you can use the learned semantics. We present the results in Table 5.3.

We compute the coverage as $\frac{\text{correct}}{\text{total}} * 100\%$. The coverage is the ratio of instructions that our semantics describe, out of all instructions. Note that, unlike completeness and correctness, we are no longer using the distinction between seen and unseen instructions here.

Source	Correct	Total	Coverage
scan	216561	4655355	4.65%
gcc	187	42749	0.44%
grep	52	13058	0.40%
ls	36	8362	0.43%
perl	231	83501	0.28%
ssh	82	36785	0.22%
libxul	3489	1616659	0.22%

Table 5.3: The coverage of our semantics. The total includes both in-scope and out of scope instructions.

The difference between the instruction scan and the Linux binaries can be explained by looking at which part of the instruction space we enumerated. Most notably we did not have enough time to enumerate some common operations like pushing and popping values to and from the stack, as well as some common operations like bitwise ANDs, XORs and subtractions. These operations are located right after a huge group of instructions starting with `0F`. Many recent instructions were added to this group, for example MMX and SSE instructions. Since we enumerate instructions in lexicographical order, we needed to enumerate all instructions in this group before enumerating the much more common instructions found after the group.

5.4 Undocumented instructions

We compare the encodings we found with XED. If XED is able to decode an instruction, we assume it is documented. We manually inspected the instructions that XED was not able to decode.

Most notably, we have identified a set of instructions that fail to decode in XED, `objdump`, Capstone and various other disassemblers. It is a VEX-prefixed instruction with

opcode 00. The AMD reference manual (March 2021) does not list any VEX instructions with opcode 00. One instruction belonging to this group of encodings is C4037D000000. Our encoding analysis has identified no dataflows besides a single memory access. We therefore suspect that the instruction is operating on floating point or vector registers, which we considered out of scope. It is likely that this instruction is only present on AMD CPUs. We have tried executing the instruction on multiple CPUs, and only found it working on AMD CPUs:

1. AMD Ryzen R9 3900X: Executes
2. AMD EPYC (2nd gen): Executes
3. Intel i7-8700: Illegal instruction
4. Intel Celeron 847: Illegal instruction
5. Intel Xeon (Skylake): Illegal instruction

We also encounter the opposite of the disassembly “bug” that Sandsifter[6] identified: Intel XED ignores the operand size prefix when decoding a jump instruction. Therefore, it always reads a 4-byte immediate jump offset, regardless of the operand size. This is correct, as far as Intel processors are concerned. Our AMD CPU (and therefore our encodings as well) does take the operand size prefix into account and decodes a 2-byte instead of a 4-byte jump offset. This is not an undocumented instruction, but an implementation difference.

Chapter 6

Conclusion

This thesis presents libLISA, an approach to learning the semantics for entire instruction sets. We rely only on a minimal set of manually generated information. We need to know about CPU registers, flags, memory, and how to observe the execution of an instruction. From this, we can learn the semantics of some instructions automatically.

Instead of using a disassembler library, we infer properties of instructions by fuzzing. We identify several dataflows for each instruction. Each dataflow describes how a single CPU register, flag or memory location is updated by the instruction. We group many instructions together into *encodings* by looking at the similarities and differences between the dataflows. Encodings describe dataflows for a group of instructions. By using program synthesis techniques, we can then synthesize a computation for each dataflow. These computations, together with the dataflows, form the semantics of an instruction.

We implemented this approach for instructions that can be executed in x86-64 Linux user space. We ran our implementation for 322 hours. In that time we found 25373 encodings, which represents around 36.13% of decodable x86-64 instructions. We were able to synthesize full semantics for 4395 encodings. Our generated semantics were correct for 99.42% of instructions for which we were able to synthesize semantics.

6.1 Discussion

Our encoding analysis will not work for all imaginable CPUs. Our encoding analysis requires that instruction semantics fit our model. In particular, we expect a fixed number of memory accesses to fixed-size memory regions. Conditional memory accesses are supported, as we can consider them as a memory read or write that is only used if a condition is met.

The specific implementation for x86-64 has more limitations. The implementation relies on page faults, which means it also requires memory accesses to either occur in ascending or descending order, or be at least a full page size apart. When learning the semantics of, for example, an emulator, this restriction would not be necessary. An emulator could be modified to report all the memory accesses it performed, so that we would not need to rely on page faults to detect them.

In some cases, it is impossible to properly analyze an instruction. For example, consider the instruction `xor r8, [r8]`, which XORs the value stored in register 8 with the value stored in memory at the address stored in register 8. Our observations will now be constrained to whatever valid memory addresses we can put in r8. When making observations from x86-64 Linux user space with `ptrace` on a common consumer AMD or Intel CPU, we can only set the lower 47 bits of the memory address. This means that we

cannot properly observe how the XOR affects the upper 17 bits of r8. Encoding analysis might therefore return an incorrect result.

While we implemented libLISA for a variable-length instruction set, it should work on fixed-length instruction sets as well. We are mainly using an encoding-based enumeration technique rather than a length-based enumeration technique. Since encoding-based enumeration does not rely on instruction lengths, we can consider a fixed-length instruction set to be a variable-length instruction set where all instructions have the same length. The overhead to determine the instruction length is negligible.

We cannot guarantee that, given enough time, we can enumerate all instructions of an instruction set. We use length-based enumeration as a fallback for when we are not able to successfully analyze an instruction. As described in Section 2.1, it is possible for instructions to be missed when tunneling. We suspect that it is very rare for this to happen, at least on x86-64. We have not seen any missed instructions that were missed because of tunneling during the 322 hours of enumeration we did. The only alternative to tunneling is exhaustive enumeration. This might work on fixed-length instruction sets like ARM, but for x86-64 it is infeasible.

Since our correctness evaluation is based on fuzzing, we cannot prove correctness. This is not necessarily an issue. Without access to the hardware designs of the CPU, proving correctness is impossible. Other approaches ultimately also end up either comparing semantics to manually specified semantics, or use fuzzing and manual test cases to compare results. The fact that we also use fuzzing to learn the semantics does not make the results less reliable. Using fuzzing for validation has some problems, which we described in Section 5.2, but none of these problems become worse by also using fuzzing for analysis.

The effectiveness of our encoding analysis depends on assumptions we make about the structure of instructions. Most importantly, we assume that many bits will have a single purpose. We believe that this assumption will hold for most instruction sets. If bits have a single purpose, no complex logic is required to decode the values. This means that instruction decoding uses less transistors, and will be faster and more energy-efficient.

The current implementation is limited in scope. It will only look at general-purpose registers and flags. This means that we have an incomplete view of the behavior of, for example, floating point, SSE and AVX instructions. We also cannot fuzz the segment registers `fs` and `gs` efficiently, since modifying their values is more complicated than modifying the values of normal registers.

To the best of our knowledge, this thesis presents the first approach that splits an instruction into many dataflows. Godefroid and Taly [7] used just two templates for two fixed purposes. One template for the "main calculation" and one to update the flags register. Heule et al. [10] synthesized programs for the entire instruction in one go. By splitting the semantics into many dataflows, we are able to reduce the complexity of the expressions that we need to synthesize. This makes it much more feasible to synthesize the semantics for thousands of instruction variants.

Because we considered large parts of the x86-64 instruction set out of scope, the generated semantics are not immediately useful for use in other tools. Since we were not able to enumerate the full instruction space, our semantics also lack many common instructions. However, we believe that with some of the improvements listed in Section 6.2 the generated semantics could become useful enough for use in analysis tools. We also think our semantics could be used to build an emulator, although all privileged instructions would still have to be implemented manually. Using the semantics in compilers is the most difficult, as compilers need to be able to produce a program that runs correctly on multiple architectures. Undefined behavior and implementation-dependent behavior

would have to be annotated manually for each instruction.

6.2 Future work

Currently, only a single configuration of a CPU can be analyzed. For example, for floating point operations, the CPU has additional configuration bits for rounding modes, floating-point register size, etc. By treating all these configuration bits as a "virtual prefix" for every instruction, we could re-use the existing analysis to efficiently learn semantics for multiple processor configurations. Configuration bits that do not affect the execution of an instruction (for example, floating point rounding mode for a non-floating point instruction) would be identified as such. This reduces the total number of instructions that need to be enumerated.

A similar approach could be taken to reduce the overhead of real prefixes. By turning a real prefix into a single bit in a mandatory "virtual prefix", the logic that determines what purpose each bit serves would work for entire prefixes as well. Bits for prefixes that do not affect a particular instruction would be identified as *Don't Care* bits, and no separate analysis of the prefixed and non-prefixed instruction variants would be necessary.

Our implementation currently only supports general-purpose registers. Modern CPUs often also have floating point and vector instructions. x86-64 in particular also still uses some segment registers, even in x86-64. In order for the learned semantics to be useful in real-world scenarios, we would have to learn semantics for all these instructions as well.

The learned semantics themselves could be extended with more information. For example, the semantics could describe when CPU faults occur, the exact execution time of an instruction or the memory locking and/or ordering behavior. Given the right hardware, information like energy usage or EM emissions could also be learned automatically. On some CPUs, it is also possible to measure the power usage of subsystems like the instruction decoder individually [11]. It is likely that our encodings correspond relatively well with simple CPU implementations that directly execute the instructions. However, for advanced implementations that re-order instructions, use register rewriting, etc. characteristics like timing, energy usage or EM emissions of an instruction depend on the surrounding instructions. In such a case, we would need to look at sequences of instructions rather than a single instruction.

Our approach could be extended to other architectures. For example, ARM architectures found in devices like phones, Chromebooks and Apple M1 Macbooks. Our implementation contains only a small module with architecture-specific code. Adapting the implementation to a relatively similar architecture like 64-bit ARM or RISC-V would therefore be easy.

Using the semantics, detailed comparisons could be made between various CPU implementations. Differences in behavior could be caused by CPU bugs or by differences in implementation of undefined details in the specification.

The current implementation, which uses the Linux `ptrace` API, requires multiple context switches and system calls to make an observation. Making a single observation takes up to 30us. We suspect that most of this time is overhead. To remove most of the overhead caused by the Linux kernel, these observations could be done in a bare-metal process running on a hypervisor. Additionally, by using a bare-metal process we could map memory into the region that is normally reserved for kernel-only usage. This could make analysis faster and more accurate.

The fact that we do not use a disassembler library, poses problems for validation. Ideally, we would like to compare our results directly with previously specified semantics,

both manual and automatic. The authors of the formalization of all non-deprecated x86-64 user space instructions are working on formalizing instruction decoding [4]. The formalized instruction decoding together with the formalized x86-64 semantics could be used to verify our results much more accurately.

With the learned semantics and some extensions mentioned here, accurate emulators for specific CPUs could be generated mostly automatically. Only privileged instructions and instructions that are impossible to synthesize (for example, an instruction that loads a random value into a register) would have to be implemented manually. While existing emulators often already provide many different choices of CPU implementation, these choices often only enable or disable some CPU features. For example, choosing the CPU type 'x86 486' in QEMU will disable features like SSE and AVX, but it will still use the same instruction decoding as every other CPU. Using our approach, separate semantics could be learned for each CPU.

We use a simple program synthesis technique to synthesize semantics for instructions. More advanced program synthesis techniques could speed up synthesis, as well as synthesize more computations. The grammar could also be tweaked to improve synthesis. For example, a grammar operating on bitvectors rather than 128-bit numbers would likely perform better at expressing bitshifts and rotations, although it might perform worse for mathematical operations.

To make the learned semantics more useful for real-world use cases, the enumeration strategy could be changed. Currently, we enumerate the instruction space in lexicographical order. This will not enumerate the most used and most useful instructions first. A simple improvement could be to enumerate the shortest instructions first. Shorter instructions usually perform the most commonly-used operations, while longer instructions perform operations that are not needed as often. A good example of this in the x86-64 instruction set is stack pushing and popping instructions, which are of the form `5X` where the X is a hexadecimal digit indicating the register and the stack operation (pop/push). A less common instruction like the `AESENC` instruction that performs a single round of AES encryption takes at least 5 bytes.

Bibliography

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2017, pp. 319–336.

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. "Search-Based Program Synthesis". In: *Commun. ACM* 61.12 (Nov. 2018), pp. 84–93. ISSN: 0001-0782. DOI: 10.1145/3208071. URL: https://doi.org/10.1145/3208071.

[4] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. "A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1133–1148. ISBN: 9781450367127. DOI: 10.1145/3314221.3314601. URL: https://doi.org/10.1145/3314221.3314601.

[5] Rens Dofferhoff, Michael Göebel, Kristian Rietveld, and Erik Van Der Kouwe. "iScanU: A Portable Scanner for Undocumented Instructions on RISC Processors". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE. 2020, pp. 306–317.

[7] Patrice Godefroid and Ankur Taly. "Automated Synthesis of Symbolic Instruction Encodings from I/O Samples". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 441–452. ISSN: 0362-1340. DOI: 10.1145/2345156.2254116. URL: https://doi.org/10.1145/2345156.2254116.

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

[9] Shilpi Goel, Warren A Hunt, Matt Kaufmann, and Soumava Ghosh. "Simulation and formal verification of x86 machine-code programs that make system calls". In: *2014 Formal Methods in Computer-Aided Design (FMCAD).* IEEE. 2014, pp. 91–98.

[10] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stratified Synthesis: Automatically Learning the X86-64 Instruction Set". In: *SIGPLAN Not.* 51.6 (June 2016), pp. 237–250. ISSN: 0362-1340. DOI: 10.1145/2980983.2908121. URL: https://doi.org/10.1145/2980983.2908121.

[11] Mikael Hirki, Zhonghong Ou, Kashif Nizam Khan, Jukka K Nurminen, and Tapio Niemi. "Empirical study of the power consumption of the x86-64 instruction decoder". In: *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16).* 2016.

[12] Xixing Li, Zehui Wu, Qiang Wei, and Haolan Wu. “UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching”. In: *IEEE Access* 7 (2019), pp. 149224–149236.

[14] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: `10.1145/1273442.1250746`. URL: `https://doi.org/10.1145/1273442.1250746`.

[16] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[17] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic Optimization of Floating-Point Programs with Tunable Precision”. In: *SIGPLAN Not.* 49.6 (June 2014), pp. 53–64. ISSN: 0362-1340. DOI: `10.1145/2666356.2594302`. URL: `https://doi.org/10.1145/2666356.2594302`.

Articles and sources that are not peer reviewed

[3] *Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals*. Last accessed on 28/04/2021. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[6] Christopher Domas. *Breaking the x86 ISA*. Last accessed on 22/04/2021. 2017. URL: https://github.com/xoreaxeaxeax/sandsifter.

[13] William Mahoney and J Todd McDonald. *Enumerating x86-64–It's Not as Easy as Counting*. Last accessed on 14/06/2021. URL: https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf.

[15] *Pin - A Dynamic Binary Instrumentation Tool*. Last accessed on 14/06/2021. URL: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.