libLISA: Instruction Discovery and Analysis on x86-64

JOS CRAAIJO, Open Universiteit, Netherlands
FREEK VERBEEK, Open Universiteit, Netherlands and Virginia Tech, USA
BINOY RAVINDRAN, Virginia Tech, USA

Even though heavily researched, a full formal model of the x86-64 instruction set is still not available. We present LIBLISA, a tool for automated discovery and analysis of the ISA of a CPU. This produces the most extensive formal x86-64 model to date, with over 118 000 different instruction groups. The process requires as little human specification as possible: specifically, we do not rely on a human-written (dis)assembler to dictate which instructions are executable on a given CPU, or what their in- and outputs are. The generated model is CPU-specific: behavior that is "undefined" is synthesized for the current machine. Producing models for five different x86-64 machines, we mutually compare them, discover undocumented instructions, and generate instruction sequences that are CPU-specific. Experimental evaluation shows that we enumerate virtually all instructions within scope, that the instructions' semantics are correct w.r.t. existing work, and that we improve existing work by exposing bugs in their handwritten models.

CCS Concepts: • **Software and its engineering** → *Semantics*; • **Computer systems organization** → *Complex instruction set computing*; • **Hardware** → Semi-formal verification.

Additional Key Words and Phrases: instruction semantics, instruction enumeration, synthesis

ACM Reference Format:
Jos Craaijo, Freek Verbeek, and Binoy Ravindran. 2024. libLISA: Instruction Discovery and Analysis on x86-64. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 283 (October 2024), 29 pages. https://doi.org/10.1145/3689723

1 Introduction

A proper understanding of the microarchitectural behavior of the instructions that can be executed on a CPU is crucial for any effort related to binaries. Binary verification, binary security analysis, binary patching, decompilation, and compiler construction/verification each require – first and foremost – a semantic model of each instruction that is executed by the binary. Semantics are needed that are *trustworthy*, *executable* and *complete*.

Even with all the research efforts that have been put into this topic [8, 12–15, 18, 20], to this day there is no complete and trustworthy model of the x86-64 architecture. This is caused by the sheer complexity of the x86-64 architecture: the informal specification found in Intel manuals is roughly 4700 pages, and even these are known to be not trustworthy [15]. Specifications of CPU architectures often rely heavily on manual work, which is error-prone and labor-intensive. This situation becomes even more dire when taking into account that different x86-64 machines will behave differently: not only can they have different instruction sets, but behavior is also allowed to be undefined, in which case the same instruction has different behavior on different machines.

Accurate semantics must inherently be CPU-specific. Consider for example the assembly program in Figure 1. Nowhere in the literature of the current state-of-the-art can semantics be found that

Authors' Contact Information: Jos Craaijo, Open Universiteit, Heerlen, Netherlands, jos.craaijo@ou.nl; Freek Verbeek, Open Universiteit, Heerlen, Netherlands and Virginia Tech, Blacksburg, USA, freek@vt.edu; Binoy Ravindran, Virginia Tech, Blacksburg, USA, binoy@vt.edu.

ACM 2475-1421/2024/10-ART283
https://doi.org/10.1145/3689723

```
f0:
    48 89 f8     mov     %rdi,%rax
    48 31 ff     xor     %rdi,%rdi
    c0 d0 09     rcl     $0x9,%al
    71 04        jno     40115f <skip>
    b0 3c        mov     $0x3c,%al
    0f 05        syscall
skip:
    c3           ret
```

Fig. 1. An example of a function in an x86-64 binary. For convenience, the output of the objdump disassembler is listed next to the bytes. The function moves its first argument RDI into RAX, clears RDI with an XOR, then performs a rotate-with-carry (rcl) of 9 on RAX, and finally conditionally jumps (jno) to the end of the function if the overflow flag is unset, or executes a syscall before returning otherwise.

accurately describe the behavior of this function. This is because the value of the overflow flag after RCL is undefined. For example, on an AMD 3900X the code will execute the SYSCALL when RAX is 0x80, while on an Intel Xeon Silver 4110 the SYSCALL is never executed. Existing semantics will either be undefined, or incorrect (see Section 5.2.2).

In this paper we introduce libLISA: a tool that can fully automatically scan a large part of the instruction space of an x86-64 CPU, discover instructions, and synthesize their semantics. The result is *CPU-specific* semantics, i.e., semantics that define what the current CPU actually does even in the case of instructions whose behavior is considered "undefined" by manually written specifications. Both discovery and synthesis are completely automated. We rely on as little human specification as possible. Notably, we do not rely on a handwritten disassembler to dictate which bitstrings are valid instructions.

For each CPU, the entire space of possible executable bitstrings is grouped into *encodings*: bitstrings where certain bits are found to be indicating a certain *operand*. A hypothetical example could be the encoding 00a1010aa0 where the three a bits of the bitstring are found to be indicating an input register. The intuition is that each encoding models a group of actual instructions (i.e., bitstrings) that each perform the same operation but on different operands. Note that there is no relation between the concept of encodings (which are *derived* bottom-up) and existing concepts such as mnemonics or instruction variants (which are *manually created* top-down). Subsequently, we try to synthesize semantics for each encoding. The result is a partial *mapping from encodings to semantics*: partial, as synthesis may fail.

We have run libLISA on five different x86-64 CPU architectures. This enumerates roughly 118 000 encodings per architecture; this number differs based on which x86-64 extensions the CPU supports. For roughly 88% of all encodings we can synthesize semantics. One run takes roughly three to four months. We have exposed executable instructions that were *undocumented* and synthesized their semantics: these are executable bitstrings that are not in the AMD and Intel manuals and that are unknown to both assemblers and disassemblers. We mutually compare the five x86-64 machines by partitioning the set of encodings based on whether their semantics are equivalent. This allows one to construct instruction sequences that behave differently on different architectures, e.g., that behave benign on an AMD Ryzen R9 3900X but behave maliciously on an Intel Core i9-13900.

Natural questions to ask are: "Did we find all executable instructions?" and "Are the generated semantics correct?" We provide extensive evaluation of libLISA's output in Section 5, but it is important to note that there is no ground truth. Instruction semantics are notoriously subtle, so even if we could manually check the semantics for over 100 000 encodings, that kind of verification effort would in itself be highly error-prone and untrustworthy. We therefore evaluate these questions relative to best-effort oracles (e.g., by enumerating all instructions in binaries such as `ssh` and `gcc`) and relative to the most extensive x86-64 specification currently available in related work [8]. This shows that we find virtually all instructions in scope.

To the best of our knowledge, the work of Dasgupta et al. is the most extensive specification of x86-64 userspace instructions currently available [8]. Their work is based on the earlier work of Heule et al. [15] on Strata which provides a way to synthesize the semantics for 1795 instruction variants. Dasgupta et al. manually extend this work to 3155 instruction variants, which required a manual effort of eight man-months [8]. That set of instruction variants is not complete: libLISA enumerates 11 152 encodings that cannot be mapped to instruction variants from this set. A mutual comparison between the semantics generated by libLISA and their semantics exposed bugs: cases where their semantics did not represent what actually happens when instructions are executed on a CPU. Moreover, there are notable differences: 1.) libLISA is fully automated, 2.) libLISA *discovers* the instructions instead of leveraging a pre-existing set of instruction variants taken from a handwritten assembler, and 3.) libLISA generates CPU-specific semantics. A more thorough comparison to the state-of-the-art can be found in Section 3.

We restrict the enumeration scope of our work to x86-64 userspace instructions. We choose x86-64, as it is a complex and widely-used instruction set architecture with variable length instructions between 1 and 15 bytes. Other modern architectures such as ARM or RISC-V have significantly simpler fixed-length or mostly fixed-length instructions. Additionally, there is no complete and trustworthy model of the x86-64 architecture.

We restrict instruction prefixes to a limited set of allowed prefixes. We do not analyze concurrency, memory ordering or timing behavior. We also consider instructions using segment selectors and segment descriptors out-of-scope, as well as instructions that produce an undefined instruction exception (e.g., `UD`). Note that an "undefined instruction exception" is not in any way related to an "instruction with undefined behavior". In Section 2.3 we define a clear and unambiguous scope in more detail.

To summarize, this paper contributes:

(1) a method for automated discovery and analysis of the ISA of a CPU;
(2) an implementation of this method, called libLISA, for CPUs implementing the x86-64 architecture; and
(3) an extensive evaluation concerning completeness, correctness and the relation between the output of libLISA and the state-of-the-art.

2 Overview

The input-output relations of libLISA are depicted in Figure 2. It uses a *CPU observer*, and produces a set of *encodings*, as well as *semantics* for each encoding.

2.1 CPU Observers

A CPU executes *instructions*: bitstrings of $8n$ bits. Examples are the bitstrings `00000000 11011000` and `01000000 00000000 11011000`, which correspond to the human description `ADD AL, BL`.

A CPU instruction operates on a *CPU state*. CPU state consists of all stored data a CPU can access: registers, flags, memory, and internal microarchitectural state like caches. We only need a small part of the CPU state for our analysis. A CPU state is represented as a bitstring. That bitstring contains the contents of in-scope registers, flags and memory. We only store memory areas when we have determined that this memory is accessed, as it would be infeasible to store all 2^{64} bytes of

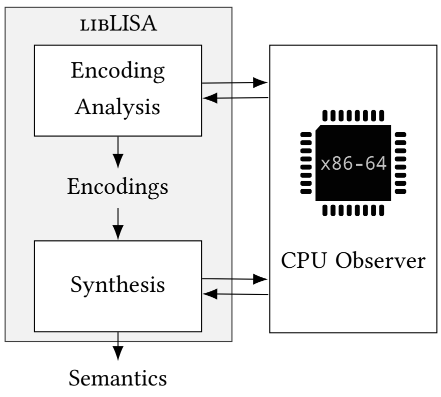

Fig. 2. The input-output relations of libLISA.

memory a 64-bit CPU could access. A CPU observer is a function that takes as input an instruction and a CPU state, and produces as output a new CPU state.

Our CPU state representation for x86-64 contains 769 bytes (excluding memory). It includes general-purpose registers (RAX..R15), RFLAGS, FS and GS. It also includes state from processor state components 0 (x87), 1 (SSE) and 2 (AVX): the 16 YMM registers (including the exception flags and DAZ from the MXCSR register), and the ST/MMX registers (including FSW and FTW).

The x86-64 CPU state representation does not contain unused segment registers (CS, DS, SS, ES) or state from other processor state components not mentioned above (e.g., virtualization registers, MPX, or AVX-512).

Anything not listed above is not part of the CPU state. It is therefore not observable. This does not mean that instructions using this state cannot be analyzed. Instead, the resulting semantics describe the instruction as if the parts of the CPU state that are not in our CPU state representation do not exist. For example, the semantics for a CLFLUSH (cache flush) instruction will look like an instruction that does nothing.

2.2 Encodings and Semantics

We introduce the concept of an *encoding*: a group of instructions that differ only by operands. The operands must remain of the same type, i.e., registers, flags, immediates or memory. In order to obtain an encoding, it must have been established what the operands are, and thus what the in- and outputs of the group of instructions are. An encoding consists of a *bitpattern* (for grouping instructions), as well as *dataflows* (for operand identification).

Example 1. ***(Encoding)*** *Consider the x86-64 instruction* 00000000 11011000. *An encoding contains the following information:*

$$\begin{array}{llcl}
\textbf{\textit{Bitpattern:}} & \multicolumn{3}{l}{\texttt{00000000 110}\underline{\texttt{bb}}\texttt{0}\underline{\texttt{aa}}} \\
 & \multicolumn{3}{l}{\underline{\texttt{aa}}\text{: } [\texttt{00} \mapsto \mathsf{RAX}, \texttt{01} \mapsto \mathsf{RCX}, \ldots]} \\
 & \multicolumn{3}{l}{\underline{\texttt{bb}}\text{: } [\texttt{00} \mapsto \mathsf{RAX}, \texttt{01} \mapsto \mathsf{RCX}, \ldots]} \\
\textbf{\textit{Dataflows:}} & \underline{\texttt{aa}} & \coloneqq & \square_1(\underline{\texttt{aa}}, \underline{\texttt{bb}}) \\
 & \mathsf{RIP} & \coloneqq & \square_2(\mathsf{RIP}) \\
 & \mathsf{CF} & \coloneqq & \square_3(\underline{\texttt{aa}}, \underline{\texttt{bb}}) \\
 & \mathsf{ZF} & \coloneqq & \square_4(\underline{\texttt{aa}}, \underline{\texttt{bb}})
\end{array}$$

To formalize these notions we introduce various concepts, summarized in Table 1. Immediate values and registers (and flags) are straightforward. An *address computation* of type A is a function over registers and immediate values (e.g, RAX + 4 ∗ ECX). A *memory access* of type M is a tuple with an address computation and a size.

Table 1. The components of libLISA's instruction semantics

Immediate Value	I
Register / Flag	R
Address Computation	A
Memory Access	$M = A \times \mathbb{N}$
Part	$P = [\mathbb{N}]$
Bitpattern	$B = P \times [\mathbb{B}] \rightharpoonup (I \cup R \cup A)$
Destination	$D = P \cup R \cup M$
Source	$S = D \cup I$
Dataflow	$F = D \times \{S\}$
Computation	C
Encoding	$E = B \times [F]$
Semantic	$\Sigma = E \times [C]$

Bitpatterns. The bitpattern identifies *parts* of the bitstring, as well as the constituents these parts are mapped to, given concrete instantiations. We name the parts using underlined letters. Formally, a part can be modeled as a list of indices within the bitstring. A *bitpattern*, then, is a mapping from parts and bitstrings to constituents: either immediate values, registers or address computations. Reconsidering Example 1, we formally have the parts $[0, 1]$ and $[3, 4]$ and the bitpattern:

$$\langle [0,1], \texttt{00}\rangle \mapsto \texttt{RAX}, \langle [0,1], \texttt{01}\rangle \mapsto \texttt{RCX}, \ldots$$
$$\langle [3,4], \texttt{00}\rangle \mapsto \texttt{RAX}, \langle [3,4], \texttt{01}\rangle \mapsto \texttt{RCX}, \ldots$$

However, we use notation $\underline{aa}$ as in Example 1 when possible.

Dataflows. A *dataflow* identifies a list of *sources* that are used as inputs to a computation that produces a value stored in a *destination*. Destinations are represented by parts, registers, or memory accesses. Sources can be immediate values as well. A dataflow can thus be instantiated using the part mapping of the bitpattern. Note that it does not define the computation that actually occurs. In Example 1, these computations thus have been denoted with undefined boxes.

The generated *semantics* consist of encodings together with defined *computations* for all dataflows. These computations describe how new values for destinations are computed using a set of sources. Effectively, the boxes in Example 1 are replaced with actual functions.

Example 2. ***(Semantics)*** *The semantics for the encoding from Example 1 are as follows:*

$$\begin{aligned}
\square_1(x,y) &= x + y \; mod \, 256 \\
\square_2(x) &= x + 2 \\
\square_3(x,y) &= x + y > 255 \\
\square_4(x,y) &= (x + y \; mod \, 256) = 0
\end{aligned}$$

Normally, instructions increment RIP by the instruction length to advance to the next instruction. Branch instructions are considered as normal instructions that update RIP by (conditionally) incrementing RIP with the jump offset. Repeating instructions, such as REPNZ STOSB, perform one iteration of the repetition at a time, but do not increment RIP as long as the repeat condition holds.

2.3 Scope

We restrict the enumeration scope to keep the runtime feasible. We exclude instructions with the following prefixes from being analyzed: `REPNZ` (`F2`), `REPZ` (`F3`), segment overrides (`26`, `2E`, `36`, `3E`, `64`, `65`), and data overrides and address size overrides (`66`, `67`). We enforce an ordering on instruction prefixes: a lock prefix (`F0`) must always appear before `REX` (`40-4F`) prefixes.

The primary reason for the restrictions on prefixes is running time. These prefixes can appear in front of any (non-VEX prefixed) instruction. Even when excluding invalid sequences of prefixes, including these prefixes would increase runtime by at least a factor of 84×. Segmentation, looping instructions, and data and address size overrides are excluded because these are the least commonly used prefixes. Four out of six segment registers are hard-coded to 0, while the other two have limited uses. The looping prefixes can only be applied to a handful of instructions, and are ignored for all other instructions. The data size overrides are used for legacy encodings of SSE operations and 16-bit arithmetic. The address size overrides are only relevant when using 32-bit pointers. As shown in Table 5 in Section 5, the scope still covers 97.36% of instructions found in Linux binaries.

Furthermore, instructions are deemed out of enumeration scope in the following cases: they

(1) perform a variable number of memory accesses,
(2) do not perform the memory accesses in a fixed order,
(3) always fault (e.g., with the undefined instruction exception `UD`),
(4) access registers not included in the CPU state representation described in Section 2.1 (e.g., `MXCSR`),
(5) perform operations involving segment selectors (e.g., `LAR`, `LSS`),
(6) require privileges (i.e., they do not run in CPU ring 3)
(7) save or load CPU state (e.g., `XSAVE` or `FRSTOR`)

The rationale behind this scope is a trade-off between the additional implementation complexity and additional running time that adding support for a larger scope would entail, versus the yield. For example, implementing support for a variable number of memory accesses is very hard and will likely impact the running time, but to the best of our knowledge there is only a single instruction in the x86-64 architecture that exhibits such behavior.

We would like to stress that, even if instructions are out of enumeration scope, they can still be analyzed on-the-fly. For example, if one wants to analyze a binary and encounters an instruction that is outside the enumeration scope, semantics for this instruction can still be generated on-the-fly, as long as it is within synthesis scope. We demonstrate this in Section 5.4.

We do not synthesize semantics for instructions that perform floating-point arithmetic. We exclude these, because floating point operations can be approximate. We found two approaches in related work. The first consists of using uninterpreted functions and leaving the exact semantics up to the user [8]. This is not suitable for libLISA, as it is impossible to synthesize uninterpreted functions. The second is to define floating-point semantics in terms of the semantics of floating-point instructions [15]. We considered this unsuitable for libLISA, as this means the same semantics might produce different results on different CPUs.

For all other enumerated encodings, i.e., all encodings not using floating-point operations, we expect synthesis to produce semantics. However, this may fail, e.g., due to a time-out (we have a bound of 2 tries of at most 7.5 minutes per encoding).

3 Related Work

There is existing work on *CPU fuzzing* (finding instructions inconsistent with the CPU specification) and on *instruction variant synthesis* (producing semantics from instruction variants). Combining these efforts would require producing instruction variants from bitstrings. That is not possible, as instruction variants are manually created top-down and cannot be reconstructed bottom-up from a CPU. This is exactly the gap that this paper aims to fill: to recover *encodings* from bitstrings, making them amenable to synthesis.

CPU fuzzing. Sandsifter [11] introduced *tunneling*, a technique for efficiently scanning an instruction space. By comparing the found instructions with a disassembler library, undocumented instructions can be identified. This approach was later improved by UISFuzz [19] and extended to RISC-V and ARM in iScanU [10]. These tools produce lists of instructions that disassemblers are unable to disassemble. In order to determine what these instructions do, manual analysis is needed.

SiliFuzz [17] generates test cases by fuzzing CPU simulators or disassemblers, and then executes these test cases on large amounts of CPUs. This allows it to leverage the semantic information encoded in CPU simulators or disassemblers to verify whether real CPUs are behaving correctly. Martignoni et al. [21] use a similar approach, but in the opposite direction. A real CPU is used as the ground truth, and compared against the behavior of emulators using fuzzing. The Cascade [28] fuzzer relies on a similar idea. It is a RISC-V fuzzer that generates valid, long, complex programs and uses these to verify the correctness of CPUs.

All the tools mentioned above, assume a correct (partial) specification of the CPU exists, and aim to verify whether a CPU conforms to this specification. libLISA, on the other hand, assumes no such specification exists, and attempts to infer the specification from the CPU behavior.

Instruction semantics. There have been many attempts to obtain formal semantics for x86-64. In this section we summarize related work and compare it to libLISA (see Table 2).

Goel et al. provide an x86-64 ACL2 model [13] which covers mainly one-byte and two-byte x86-64 instructions, consisting of roughly a third of all non-privileged instruction variants. Morrisett et al. [22] developed a Coq model for a subset of 32-bit x86, which was used to implement a static analysis tool for Google's Native Client (NaCl).

The CompCert compiler [18] has a Coq model for x86-32 and x86-64, which is used to prove equivalence between C source code and the generated assembly. It only defines semantics for the instruction variants and behavior used by the compiler. For example, only the 32-bit and 64-bit variants of the `CMP` mnemonic are specified, because the 8-bit and 16-bit variants are not used. Similarly, the semantics of the flags for the `SHL` instruction variants are not specified.

Godefroid and Taly [12] were the first to introduce automatic synthesis for CPU instruction semantics. Using templates, they synthesized instruction semantics for 534 x86 (32-bit) ALU instruction outputs (a typical instruction might have 6 outputs: 1 main output and 5 flag updates).

Heule et al. [15] used a more advanced program synthesis technique, STOKE [25]. From the semantics of 58 base instructions, STOKE automatically synthesizes semantics for 1764 instruction variants. It uses the `x64asm` library as a source of instruction variants. The semantics can be exported to standard SMTLib syntax [4] using a tool provided by the authors. libLISA also provides tools to export semantics to SMTLib syntax.

STOKE supports both bitwise, integer and floating point instructions. It excludes x87, MMX, cryptographic, system-level and string instructions. libLISA does not support synthesizing floating point operations, but does analyze all instructions with in-scope prefixes.

Dasgupta et al. manually extended the work of Strata, producing the semantics to all 3155 "non-deprecated" x86-64 instruction variants supported on the Haswell microarchitecture [8]. This process took 8 man-months. The semantics are specified in the K framework. While it is possible to import SMTLib assertions for program verification, Dasgupta et al. do not provide a tool that can export the semantics to SMTLib format. We discuss this in more detail in Section 5.2.1.

Table 2. Comparison of related work. (†): Dasgupta et al. augment Strata's synthesized semantics with manually written semantics. (‡): Strata overlays a manual specification of undefined behavior on the synthesized semantics to make them non-CPU-specific.

	Goel	CompCert	Strata	Dasgupta	libLISA
Automatic	-	-	✓	-	✓
Source of instructions	Human	Human	Disassembler	Disassembler	Enumeration
Source of semantics	Human	Human	Synthesis	Human[†]	Synthesis
CPU-specific	-	-	-[‡]	-	✓
SMTLib export	-	-	✓	-	✓
Executable	✓	✓	✓	✓	✓

Table 3. Comparison of types of instructions with semantics covered by STRATA, Dasgupta et al. and libLISA. (†): The REP prefix is out-of-scope, but non-repeating string instructions are synthesized.

	STRATA	Dasgupta	libLISA
Integer, bitwise, control flow	✓	✓	✓
Privileged instructions	-	-	-
x87	-	-	-
MMX	-	-	✓
String instructions	-	✓	✓[†]
SSE/AVX (floating point)	✓	✓	-
SSE/AVX (integer)	✓	✓	✓

For both Strata and Dasgupta et al., we present a comparative overview of the types of instructions for which they provide semantics in Table 3.

For other architectures, such as ARM and RISC-V, complete formal models do exist. SAIL [3] contains formal semantics for ARM, RISC-V and CHERI-MIPS. The ARM semantics have been derived from the ARM Specification Language [23], while the RISC-V and CHERI-MIPS semantics have been handwritten. It would be interesting to implement libLISA for these architectures and compare libLISA's results against the formal models.

Formal models such as x86-TSO focus on defining concurrent memory accesses [26, 29]. While these models also include instruction semantics, their main contribution is the memory model itself. The models are largely orthogonal to our work: x86-TSO relies on instruction semantics to determine what kinds of memory accesses an instruction performs. The x86-TSO model only describes *observable* memory ordering behavior of a CPU. This suggests that libLISA could be expanded to generate semantics that define memory accesses in terms of x86-TSO's model.

4 Approach

We present an approach for systematically discovering and analyzing instructions on a CPU. We provide an overview of the three main components of this approach: 1.) the CPU observer, 2.) enumeration based on encoding analysis, and 3.) synthesis.

4.1 CPU Observer

In order to analyze instructions, we need a CPU observer. This CPU observer needs to be *fast*, *sandboxed* and *unrestricted*. It needs to be fast, because we will perform tens of millions of observations for each instruction we analyze. Instruction execution must be sandboxed, such that it does not affect our analysis tool or the operating system in unintended ways. There must be as few restrictions on the input CPU state as possible, so that we can freely observe as much of the behavior of the instruction as possible.

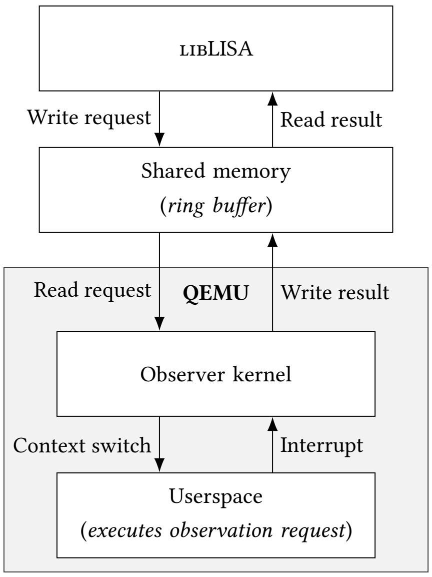

Fig. 3. The CPU observer uses QEMU with KVM hardware-acceleration to run an observation kernel, and execute observations in userspace inside the virtualized environment.

There are two common ways to observe instruction execution: *in-process observation* [10, 11, 15, 19] and *out-of-process observation* [10]. Neither of these methods fulfill all requirements. Therefore, we have developed a new instruction observation method.

Both in-process observation and out-of-process observation ultimately execute an instruction in a userspace process. After execution, control over program flow is regained using the *trap flag*, *guard pages* or *interrupt instructions*. A guard page is an unmapped page that is placed in memory right after the instruction. When the CPU has executed the instruction and attempts to load the next instruction, a page fault is triggered. This page fault is intercepted by the program, and used to regain control. The trap flag is a debugging flag available on many modern CPU architectures, including x86-64. It triggers a CPU interrupt after executing a single instruction. This interrupt is intercepted by the program, and used to regain control. Interrupt instructions are special debugging instructions that trigger a debugging interrupt. On x86-64, the `INT3` instruction is commonly used for this purpose. After execution, the result is saved and normal program execution is resumed by loading the original program state from memory.

In-process observation consists of storing the program state in memory, placing the instruction at a known memory address, and then jumping to that address. This approach is fast, but not sandboxed, and the input CPU states cannot use the address space used by the program itself or reserved by the kernel (Linux reserves half of the address space).

Out-of-process observation consists of spawning a separate observation process. This observation process is then instrumented using a debugging interface like `ptrace`. CPU state can be modified through this interface, and memory can be mapped and unmapped by placing assembly for the correct system calls in the memory of the observation process and executing it via the debugging interface. While this approach provides some sandboxing, it shares many of the same restrictions on input CPU states as in-process observation, and is very slow because the debugging interface has a lot of overhead.

We have developed a new observation approach based on hardware-accelerated virtualization and fast communication via shared memory. The approach is depicted in Figure 3. It consists of two components: a process running on the host machine, and a small bare-metal observer binary running in a virtual machine. Using a ring buffer [5], these components can communicate without the overhead of syscalls. The observer running in the virtual machine performs context switches to userspace to observe instructions. This provides hardware-enforced protection to the observer from the effects of the instruction execution.

The observer repeatedly reads an observation request from the ring buffer, executes the request, and then writes the result back to the ring buffer. Observation requests are executed similar to how an operating system performs context switches between processes. Whereas an operating system typically restores CPU state from a *process control block*, the observer restores CPU state from the observation request in the ring buffer. It then switches to userspace, allows the CPU to execute an instruction, and then regains control. Finally, the observer saves the CPU state directly to the observation result in the ring buffer.

To regain control, the observer uses `INT3` interrupt instruction by default. If we detect that the instruction does not increment the program counter by the length of the instruction (e.g., the instruction is a branching instruction), we are unable to predict the address where we need to place the `INT3` interrupt instruction. In that case, we fall back to using the trap flag. We use the `INT3` interrupt instruction by default, because it does not require writing a flag in the debugging registers and is therefore faster.

4.2 Enumeration

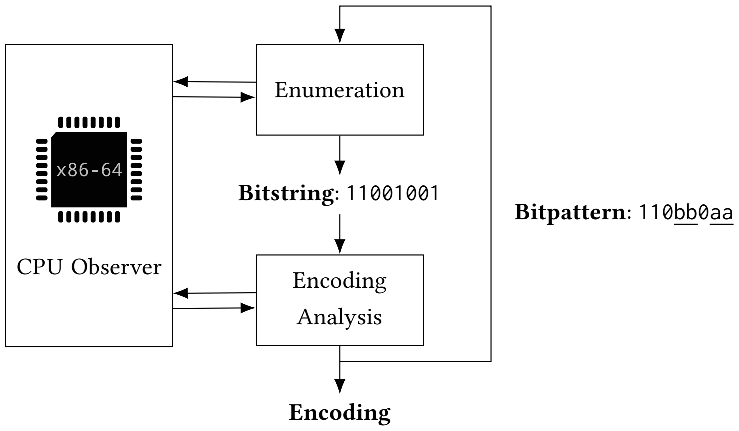

Fig. 4. A feedback loop from Encoding Analysis to enumeration makes it possible to fully enumerate large instruction spaces.

The goal of enumeration is discovering all in-scope instructions on a CPU. We do this by repeatedly selecting an uncovered instruction, and analyzing it.

For large instruction sets like x86-64, it is impossible to enumerate every individual bitstring. For example, the `MOVABS RAX, 0x152`, instruction contains a 64-bit immediate value. It is impossible to exhaustively enumerate all 2^{64} values of this immediate. We instead *skip over parts of the instruction space* using 1.) bitpatterns from encodings, 2.) randomized search and 3.) tunneling.

As described in Section 2, every encoding represents a group of instructions described by its bitpattern. During enumeration, we run encoding analysis on each valid instruction. We then use the bitpattern from the resulting encoding to skip all instructions it matches. This is depicted in Figure 4. For example, for `MOVABS RAX, 0x152` encoding analysis will yield an encoding with a bitpattern containing two parts: a 64-bit part for the immediate value and a 4-bit part for the destination register. This allows us to skip the $2^{68} - 1$ other instructions covered by this encoding.

Algorithm 1: Enumeration.

```
Result: a set of enumerated encodings $E$
$E \leftarrow \emptyset$;
$S \leftarrow \emptyset$;
$I \leftarrow$ NextUncoveredInstruction($S$);
while $I \neq$ None do
    if IsValidInstruction($I$) then
        $e \leftarrow$ AnalyzeEncoding($I$);
        if $e \neq$ Err then
            $E \leftarrow E \cup \{e\}$;
            $S \leftarrow S \cup \{i \mid i$ matches bitpattern of $e\}$;
        else
            $S \leftarrow S \cup$ Tunnel($I$);
        end
    else
        $S \leftarrow S \cup$ RandomizedSearch($I$);
    end
    $I \leftarrow$ NextUncoveredInstruction($S$);
end
return $E$
```

The enumeration algorithm is depicted in Algorithm 1. It takes no inputs, and produces a set of encodings E, covering all valid instructions in the instruction space. It makes use of five functions: `NextUncoveredInstruction`, `IsValidInstruction`, `RandomizedSearch`, `Tunnel` and `AnalyzeEncoding`.

The function `NextUncoveredInstruction` takes as input a set of covered instructions S. It returns an instruction I which is not in the set of covered instruction S. In our implementation, we start at the byte sequence `00` and then sequentially return all other byte sequences in lexicographical order. We picked a lexicographical ordering because it is simple to implement. The actual order is not relevant for the algorithm.

The function `IsValidInstruction` takes as input an instruction I and checks if it is *valid*, i.e., if executing it does not cause the CPU to throw the undefined instruction exception.

The function `RandomizedSearch` takes as input an instruction I and performs a randomized search for the first valid instruction after I. We use I as lower bound and the highest possible instruction (`FFFFFFFF`...) as upper bound for the search. We then repeatedly pick random byte sequences within the search range. If the random byte sequence is a valid instruction, we reduce the upper bound. When the upper bound has not changed for 250 000 iterations, it returns the set of instructions between I and the upper bound.

The function `Tunnel` takes as input an instruction I and performs *tunneling* [10, 11, 19] to find the first valid and analyzable instruction. Tunneling initially steps through byte sequences one-by-one. Every 2^8 steps the step size is multiplied by 2^8. Whenever the instruction length changes, the step size is reset to zero. This makes the number of steps needed to skip over an instruction with an invalid 64-bit immediate value $2^8 * 8$ instead of $(2^8)^8$. Once a valid and analyzable instruction is found, it returns the set of instructions between I and the valid and analyzable instruction.

The function `AnalyzeEncoding` takes as input an instruction I and runs encoding analysis on the instruction. Encoding Analysis is described in Section 4.3. Encoding analysis returns an encoding e. This encoding has a bitpattern, which represents the set of instructions covered by the encoding.

The algorithm repeatedly selects the next instruction not yet covered, and analyzes it. If the next instruction is not a valid instruction, randomized search is used to skip it and all consecutive invalid instructions. If it is a valid instruction, encoding analysis is run. Normally, encoding analysis produces an encoding. If this is the case, the bitpattern is used to skip over all other instructions covered by the encoding. Finally, in some cases encoding analysis might be unable to analyze the instruction. This can happen for example when the instruction violates some of the assumptions we have listed in Section 2.3. In such a case, we do not have a bitpattern, and we also cannot use randomized search. Instead, we resort to tunneling [10, 11, 19].

Skipping using bitpatterns guarantees that we do not skip over valid instructions. Randomized search and tunneling may cause us to skip over valid instructions. We therefore use bitpattern based skipping whenever possible.

As the instruction's length is often determined by a few bits in a single byte of the instruction, tunneling generally works correctly. However, there are edge-cases where this does not work. For example, consider the case where byte sequences `1b00-1cff` are invalid, except for `1c05`. In this case, tunneling will step through `1b00-1bff` in steps of 1, but after checking `1c00` the step size is increased to 256, which means the next instruction checked is `1d00`. This skips over the valid instruction `1c05`. Because of these limitations, we skip instructions using bitpatterns or randomized search whenever possible.

Randomized search is more reliable than tunneling, but only applicable for invalid instructions. It might still skip over valid instructions. This can be the case when the chance of finding the valid instruction in 250 000 tries is low. We picked the threshold through testing against tunneling. Randomized search with a threshold of 250 000 iterations performs better or equivalent to tunneling for all in-scope areas of the x86-64 instruction space. In particular, it correctly handles the tunneling edge-case described above.

4.3 Encoding Analysis

The goal of encoding analysis is to generate an encoding, i.e., parts and dataflows, from a concrete instruction bitstring. We propose a novel infer-generalize-specialize approach. This approach consists of three steps: *inferring* dataflows, *generalizing* the dataflows into an encoding, and *specializing* the resulting encoding. The infer step produces dataflows which are consistent with all concrete observations. The generalization step uses these dataflows as a basis to form a generalized encoding.

The generalization step is purely speculative, and might produce an encoding with incorrect generalizations. Generalizations are incorrect when the dataflows in the resulting encoding are not consistent with actual CPU behavior. To counter this, the specialization step removes these by specializing the encoding for cases where a generalization is proven incorrect via an observation.

Each of these three steps require the generation of random CPU states. In the rest of this subsection, we first describe how this is done, and then provide details on each of the three steps of the infer-generalize-specialize approach.

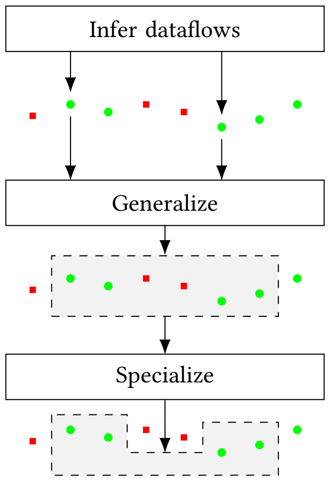

Fig. 5. The infer-generalize-specialize approach. The green circles are correct dataflows, the red squares are incorrect dataflows, and the dashed boxes represent encodings. The infer step produces some (in this case, two) correct, concrete dataflows. The generalize step combines these concrete dataflows into an encoding, but might make incorrect generalizations. The specialize step removes these incorrect generalizations.

4.3.1 Random CPU State Generation. For small CPU architectures, it is sometimes possible to exhaustively verify all possible input states. For modern architectures such as x86-64, exhaustive verification is impossible without access to hardware designs. We therefore can only verify the semantics that we generate on a subset of all possible input states.

Uniformly distributed random input states are often not useful for fuzzing. For example, when incrementing a 32-bit value we only have a 1 in 2^{32} chance of seeing a carry in the highest bit if we pick uniformly distributed random numbers (the only case where this happens is 0xFFFFFFFF). Interesting inputs often contain many consecutive zeros or ones. Therefore, our random generation is based on the computation $(R_{64} << R_z) >> R_k$ where $<<$ and $>>$ are bit shifts, R_{64} is a uniformly distributed random 64-bit number and R_z, R_k are random uniformly distributed valid bit shift counts. The resulting number is then randomly negated or kept as-is. This computation produces numbers where the number of leading and trailing zeros and ones are approximately uniformly distributed. This random generation needs to run billions of times for each encoding we analyze. Improving quality of the random numbers reduces the speed at which we are able to generate new random numbers. We have therefore opted to use this simple expression consisting of only two bit shifts.

4.3.2 Inferring Dataflows. For a given an instruction I consisting of N bits, we analyze the instruction I as well as the bit-flipped variants $\text{flip}(I, n)$ for each $0 \leq n < N$, where $\text{flip}(I, n)$ bit-flips the nth bit in I. This produces dataflows for $N + 1$ instructions that we will generalize into an encoding. The analysis is run separately for each of the $N + 1$ instructions.

Inferring dataflows consists of analyzing memory accesses and dataflows. Analyzing memory accesses consists of finding a set of accesses M as defined in Section 2 that encompass all memory accessed by an instruction. This makes it possible to reduce the CPU state to include just the parts of memory that are read or written by the instruction. Analyzing dataflows consists of finding a set of source and destination tuples (S, D) that accurately describes all data-dependencies between the input and output CPU state of the instruction.

The algorithm for identifying memory accesses is shown in Algorithm 2. It takes as input an instruction I and produces the set of memory accesses M performed by I. It makes use of three functions: `FindPageFaults`, `FindAddressComputations` and `AccessGeneratesPageFaultOnNextPage`.

The function `FindPageFaults` takes as inputs the instruction I and the current set of memory accesess M. It generates the set of page faults P that happens when instruction I is executed on a set of randomly generated CPU states where all accesses in M are mapped.

The function `FindAddressComputation` takes as input instruction I, the current set of memory accesses M and a set of page faults P. It aims to find an address computation a that is consistent with all page faults P. A computation is consistent with a page fault it the computation either returns the same address as the page fault, or if there is a computation in M that already maps to the same page.

The function `AccessGeneratesPageFaultOnNextPage` takes as input instruction I, the current set of memory accesses M, the new address computation a and a size n. It generates a CPU state which places the new access a exactly n bytes away from the end of a page, and executes instruction I on this state. It returns whether the execution of instruction I caused a page fault at the first address of the next page.

Algorithm 2: Memory access identification.

```
Input: an instruction $I$
Result: a set of memory accesses $M$
$M \leftarrow \emptyset$;
$P \leftarrow$ FindPageFaults$(I, M)$;
while $P \neq \emptyset$ do
    $a \leftarrow$ FindAddressComputation$(I, M, P)$;
    $n \leftarrow 1$;
    while AccessGeneratesPageFaultOnNextPage$(I, M, a, n)$ do
        $n \leftarrow n + 1$;
    end
    $M \leftarrow M \cup \{(a, n)\}$;
    $P \leftarrow$ FindPageFaults$(I, M)$;
end
return $M$
```

Algorithm 2 infers memory accesses iteratively. Each iteration, we generate a set of page faults P that occur with the current set of memory accesses M. If P is not empty, this means that the instruction I performs memory accesses that we have not covered in M. We find a computation that is consistent with all page faults P, and then determine the size of the access by generating input states that place the access near the end of a page. If we place the access too close to the end of the page, such that it does not entirely fit on the page, we will get a page fault. By iteratively increasing the distance to the end of the page until the access fits on the page, we can determine the size of the access. We then extend the set of memory accesses M with the new computation a and size n, and repeat this process.

After inferring all memory accesses, we can infer the dataflows.

Algorithm 3: Dataflow analysis.

```
Input: an instruction $I$ and a set of memory accesses $M$ for $I$
Result: a set of dataflows $D$
$D_b \leftarrow \emptyset$;
df $\leftarrow$ FuzzForDataflow($I, M, D_b$);
while $df \neq$ None do
    $D_b \leftarrow D_b \cup \{\text{df}\}$;
    df $\leftarrow$ FuzzForDataflow($I, M, D_b$);
end
return Reduce($D_b$)
```

The algorithm for inferring dataflows is shown in Algorithm 3. It takes as input an instruction I and the set of memory accesses M for this instruction, produced by Algorithm 2. It makes use of two functions: `FuzzForDataflow` and `Reduce`.

The function `FuzzForDataflow` takes as input the instruction I, the set of memory accesses M, and the set of found byte-wise dataflows D_b. It returns a dataflow (b_s, b_d), which represents a dataflow between byte b_s in the input state and byte b_d in the output state. We say that there is a dataflow (b_s, b_d) if there is some input CPU state for which a change to the value of b_s causes the value of b_d in the output CPU state to change. The function only returns dataflows that are not already present in the set of found byte-wise dataflows D_b. It uses fuzzing to try and find a pair of CPU states which demonstrates the existence of a dataflow. If no dataflow is found, it returns `None`.

The fuzzing strategy consists of generating a random CPU state, and then generating another state by randomly changing some part of the state. This can be a single byte, a subset of all bytes not present in the sources in the hypothesis, or all bytes except some subset of the sources in the hypothesis. We exclude some sources of the hypothesis to ensure that we can find sources for destinations that are already present in the hypothesis. That is, if we already know that a modification in b_s causes a change in b_d, we must generate pairs of states where b_s is identical to be able to discover that a modification in b'_s also causes a change in b_d.

The function `Reduce` reduces the dataflows between individual bytes in the CPU state to dataflows between storage locations, by merging byte dataflows of consecutive bytes, and translating byte indices to storage location names. It takes as input the set of found byte-wise dataflows D_b, and returns a set of dataflows D as described in Section 2.2. This makes the dataflows usable in encodings and reduces noise. Because we rely on fuzzing, sometimes not all dataflows are found. By merging dataflows, the chances of this showing up in the resulting dataflows are reduced.

4.3.3 Generalizing Dataflows into Encodings. We generalize an encoding from the set of inferred dataflows from the previous Section. For each bit-flipped instruction flip(I, n), we compare the dataflows and memory accesses against those of the original instruction I.

Bit-flipping has been shown to be effective for guiding disassembler fuzzing [16]. Rather than guiding a fuzzer, we use bit-flipping to determine which bits are likely to belong to parts of the encoding. This requires more extensive analysis of the changes that occur when flipping a bit.

In modern general-purpose instruction sets like x86-64, bits in an instruction often serve a single purpose. For example, there can be a bit in an instruction that is used in the selection of the source register. That bit typically is not *also* used for other things such as a memory computation or a destination register. When changing those bits, only the source register changes. This reduces the complexity of the instruction decoder in the CPU, as those bits can be wired directly to the register bank without further processing. We check for changes that are common uses of bits that serve a single purpose:

(1) If a register in the dataflows changes, the bit is a register-bit candidate
(2) If the output of some dataflows changes, the bit is an immediate-bit candidate
(3) If the offset of a memory access changes, the bit is an immediate-bit candidate
(4) If a memory computation changes, the bit is a memory-computation-bit candidate
(5) If more than one of the above apply, the bit is unknown

We form parts from using the candidates found from comparing the bit-flipped instruction variants. Each candidate affects certain storage locations in the dataflows or memory accesses. We consider candidates to be similar if they affect the same storage locations in the dataflows and memory accesses. Each set of similar register-bit candidates forms a register part. Consecutive and similar immediate-bit candidates from immediate value parts. Each set of similar memory-computation-bit candidates form memory computation parts. Two parts conflict if they both share one or more affected storage locations. We repeatedly remove the smallest part until there are no conflicting parts.

For register parts, we additionally enumerate every value to determine the exact register. We do this separately for each register part. For example, if we have three register parts, each 4 bits in size, we will run per-instruction dataflow analysis on 33 more instructions, 11 for each register part. The other five possible values for each register part have already been analyzed during the bit-flipping phase.

We do not aim to recover the exact encoding such as described in, for example, the Intel Reference Manual. This would be impossible, as we infer encodings bottom-up, rather than top-down. We also do not aim to identify every part in an encoding. The primary goal is to produce encodings that make enumeration feasible. This only requires identifying some subset of parts that is large enough to allow efficient skipping of instructions.

Example 3. *Consider instruction $I =$ `0100`. Let us assume that we have inferred that it performs no memory accesses, and has the following dataflows:*

$$\begin{array}{lll} \texttt{RIP} & \coloneqq & \square_2(\texttt{RIP}) \\ \texttt{BX} & \coloneqq & \square_2(\texttt{AX}) \end{array}$$

In order to generalize this dataflow into an encoding, we inspect the dataflows of the four flipped variants, determine the change compared to the original dataflow, and determine if the bit is a candidate for a part. This comparison is summarized as follows:

Variant	`1100`	`0000`	`0110`	`0101`
Dataflows	*(Invalid)*	$\texttt{RIP} \coloneqq \square_2(\texttt{RIP})$ $\texttt{AX} \coloneqq \square_2(\texttt{AX})$	$\texttt{RIP} \coloneqq \square_2(\texttt{RIP})$ $\texttt{BX} \coloneqq \square_2(\texttt{CX})$	$\texttt{RIP} \coloneqq \square_2(\texttt{RIP})$ $\texttt{BX} \coloneqq \square_2(\texttt{BX})$
Change	*N/A*	$\texttt{BX} \mapsto \texttt{AX}$	$\texttt{AX} \mapsto \texttt{CX}$	$\texttt{AX} \mapsto \texttt{BX}$
Location	*N/A*	*Destination* `BX`	*Source* `AX`	*Source* `AX`
Candidate	*N/A*	*Register-bit*	*Register-bit*	*Register-bit*

There are candidate bits for two parts: a 1-bit register part that determines the destination register, and a 2-bit part that determines the source register. These parts do not conflict. We therefore do not need to remove any of the parts.

Finally, we also inspect the dataflows for `0111` *to fully cover the possible register mappings for the 2-bit part that determines the source register. From this information, we can build the encoding:*

$$\begin{array}{llll}
\textbf{\textit{Bitpattern:}} & \multicolumn{3}{l}{\texttt{0}\underline{\texttt{b}}\underline{\texttt{aa}}} \\
 & \multicolumn{3}{l}{\underline{\texttt{aa}}\text{: } [\texttt{00} \mapsto \texttt{AX}, \texttt{01} \mapsto \texttt{BX}, \texttt{10} \mapsto \texttt{CX}, \texttt{11} \mapsto \texttt{DX}]} \\
 & \multicolumn{3}{l}{\underline{\texttt{b}}\text{: } [\texttt{0} \mapsto \texttt{AX}, \texttt{1} \mapsto \texttt{BX}]} \\
\textbf{\textit{Dataflows:}} & \texttt{RIP} & \coloneqq & \square_2(\texttt{RIP}) \\
 & \underline{\texttt{b}} & \coloneqq & \square_1(\underline{\texttt{aa}})
\end{array}$$

4.3.4 Specializing Encodings. Specialization aims to fix any incorrect generalizations that might have occurred. Dataflows are generalized into an encoding based on heuristics. This happens without any verification: generalization is based on heuristics, it does not use observations. Since generalization only looks at single bit-flips, more complex interactions between multiple bits are not accounted for.

The original instruction I is used as the ground truth for specialization. We try to find another instruction I' also covered by the encoding, where the sources (i.e., registers or memory) can be assigned values such that at least one destination has a different output after executing I compared to I'. If we find such an instruction I', the encoding is not describing the instruction correctly. We fix this by removing bits from parts to ensure that the encoding no longer covers I'.

To determine which bit to remove, we first find many instructions using the method above. We then find the index of the bit that most often differs from the original instruction I, and remove this bit. To remove a bit, we simply replace it with the concrete value from I. For example, consider the case where $I =$ `00000000 11000001` which we have generalized into an encoding with bitpattern `00000000 110`$\underline{\texttt{bb}}$`0`$\underline{\texttt{aa}}$. To remove the last bit, we replace it with the last bit of I, which is a 1: `00000000 110`$\underline{\texttt{bb}}$`0`$\underline{\texttt{a}}$`1`. Additionally, the part mapping of the bitpatterns and the dataflows have to be updated accordingly to account for the change of $\underline{\texttt{aa}}$ into $\underline{\texttt{a}}$.

We use fuzzing to identify incorrect generalizations. The fuzzing strategy consists of generating pairs of CPU input states that are identical except for the instruction that is executed.

Example 4. *Consider the encoding from Example 3. Through fuzzing, we find the following two input-output examples:*

Instruction	*Input state*	*Output state*
`0110`	CX = 5	BX = 10
`0010`	CX = 5	AX = 37

According to the encoding, the value of `BX` *after executing* `0110` *should be equal to the value of* `AX` *after executing* `0010`*. This is not the case. Therefore, we have found an incorrect generalization. To resolve this, we need to repeatedly remove bits from parts until there we can find no more incorrect generalizations.*

We determine that in the cases where we find incorrect generalizations, 100% *of the time bit 3 (counting right-to-left, starting at 1) is* 0*, which is different from its value in the original bitstring (*0*). The other bits are different around* 25% *of the time. We therefore remove bit 3 from the encoding. This gives the following specialized encoding:*

$$\begin{array}{llll}
\textbf{\textit{Bitpattern:}} & \multicolumn{3}{l}{\texttt{01}\underline{\texttt{aa}}} \\
 & \multicolumn{3}{l}{\underline{\texttt{aa}}\text{: } [\texttt{00} \mapsto \texttt{AX}, \texttt{01} \mapsto \texttt{BX}, \texttt{10} \mapsto \texttt{CX}, \texttt{11} \mapsto \texttt{DX}]} \\
\textbf{\textit{Dataflows:}} & \texttt{RIP} & \coloneqq & \square_2(\texttt{RIP}) \\
 & \texttt{BX} & \coloneqq & \square_1(\underline{\texttt{aa}})
\end{array}$$

4.4 From Encoding to Semantics

We implement existing program synthesis techniques to demonstrate the amenability of encodings to automated synthesis. Program synthesis consists of generating a program from a specification. As we assume no access to the hardware designs of the CPU, the only specification we can generate are *input/output examples* (I/O examples). An I/O example is a tuple consisting of the inputs provided to the program, and the corresponding correct output. For example, an I/O example for a function $f(x_0, x_1) = y$ might be $x_0 = 4, x_1 = 2, y = 6$.

For each encoding, synthesis has a timeout of 7.5 minutes. This timeout has been chosen such that it allows sufficient time for synthesizing more complex semantics, while keeping the runtime acceptable. Because synthesis uses randomly generated I/O examples, runtime varies based on the quality of the I/O examples. If synthesis failed or timed out, this might be because the first few I/O examples were low-quality. This might have caused synthesis to spend most of the allotted time searching in the wrong direction. Starting with a clean slate with different random I/O examples can resolve this. Therefore, we re-run synthesis a second time if the first attempt failed or timed out.

An encoding consists of multiple dataflows. Each dataflow can be synthesized independently. We represent the dataflow as a function $\square(x_0, x_1, \dots) = y$, where x_0, x_1, etc. are the sources of the dataflow, and y is the destination. We use program synthesis to find an implementation for function $\square$.

We need a synthesis algorithm that can operate on I/O examples, is efficient, and is suited for synthesizing CPU semantics. By combining existing techniques, we construct such an algorithm.

We use Counter-Example Guided Inductive Synthesis (CEGIS) [2, 27]. CEGIS consists of two parts, a *learner* and an *oracle*. The learner repeatedly forms hypotheses, and the oracle provides counterexamples to these hypotheses. This process repeats until the hypothesis is correct, i.e., the oracle cannot present a counterexample.

Our learner is an I/O example-based synthesis algorithm. It generates hypotheses based on the I/O examples it has received from the oracle. Our oracle is a fuzzer that will verify the hypothesis against at most 2 million randomly generated input states. It uses the CPU as the ground truth, and tries to find counterexamples that show that the hypothesis is not equivalent to the actual CPU behavior.

The fuzzer uses three random generation techniques: normal generation, generation with equalities and generation from interesting inputs. Normal generation re-uses the random generation from encoding analysis, described in Section 4.3. Generation with equalities also uses the random generation from encoding analysis, but chooses one storage location at random and copies its value to another storage location chosen at random. This produces an input state where two storage locations are equal. Finally, generation from *interesting* inputs randomly picks an interesting input state, and randomly modifies a storage location, a single byte in a storage location, or a single bit in a storage location. An input state is interesting if it disproved any of the previous hypotheses.

Our learner uses decision trees to scale synthesis to larger problems, and uses *enumerative* program synthesis to synthesize individual expressions in the decision tree. We synthesize Boolean decision trees using a divide-and-conquer technique based on work by Alur et al. [1]. Decision trees have conditions (integer expressions returning only 1-bit values) on non-leaf nodes, and integer expressions on leaf nodes.

Since we need to synthesize programs from only I/O examples, our only choice of search technique is *enumerative program synthesis*. Rather than enumerating all possible programs from a grammar, we enumerate over programs derived from a list of *templates*. A template is an expression that contains zero or more holes. Synthesis consists of enumerating all possible ways to fill holes, for each template. In contrast to synthesis using grammars, a hole can only be filled with a constant or an input. We use separate lists of templates for integer expressions and conditions.

The choice for template-based synthesis is primarily motivated by performance benefits. By using a set of templates that is known and enumerable at compile-time, we are able to compile the templates to machine code. This significantly reduces the overhead of template evaluation, and increases synthesis performance.

We use 143 templates for integer expressions, and 553 templates for conditions (of which 351 are derived automatically from 39 integer expressions). These templates are handwritten. The integer expression templates consist of (combinations of) arithmetic operations such as addition or multiplication, and logical operations, such as AND, OR, or bitshifts. The Boolean templates primarily consist of zero checks, sign checks and parity computations derived from the integer expressions. They also contain more generic conditions such as $\_ + \_ < (\_ << \_)$. We see that these conditions end up being used, for example, to saturate addition $(A + B < (1 << 8))$.

Compared to Godefroid and Taly [12], we define many more templates. This is explained by counting differences, scope differences and flexibility differences. Godefroid and Taly's templates are parameterized by operand size and operation, while we use separate templates for each combination of parameters. For example, Godefroid and Taly's "Bit-wise group" flag output template corresponds to around 30 of our templates. While Godefroid and Taly's scope is limited to x86-32 ALU instructions, our scope is all userspace non-floating point instructions. Additionally, we include x86-64 extensions such as BMI that Godefroid and Taly do not support. Godefroid and Taly's flag templates are tailored to the specific operation that is being synthesized. For example, all flag outputs of Godefroid and Taly's "Bit-wise group" must be a combination of up to three Boolean terms over the main output. Our templates are defined independently of a main output, which is necessary because we do not have access to a disassembler library to identify the main output.

Expressions are synthesized by filling in templates with holes. Each template contains zero or more holes. A hole can be filled with a constant or an input x_i. Although individual expressions are constrained to predefined templates, the combination of multiple templates in decision trees makes this approach efficient. For example, a conditional jump instruction might perform a computation similar to $\textbf{if } (\neg a \wedge \neg(b \oplus c)) \oplus d \textbf{ then } x + n \textbf{ else } x + 2 \textbf{ fi}$ to update the instruction pointer. It would be infeasible to generate such an expression in one go. However, because we generate a decision tree we can split this expression into six sub-expressions: a, b, c, d, $x + n$ and $x + 2$. All of these expressions are very easy to synthesize.

An expression consists of function calls and terms. Expressions always operate on signed 128-bit integer values. Function calls are simple built-in operations, e.g., addition or bit shifting. Terms are either constant values or inputs. Constant values C can be 0..16, $8n - 1$ and $8n$, where $1 \leq n \leq 8$. An input is the value of a source, i.e., x_i, and an interpretation.

Dataflow sources and destinations can be integer values smaller than 128 bits (e.g., 64-bit general-purpose registers) or byte sequences (e.g., memory). Therefore, inputs must be converted to 128-bit integer values, and outputs must be converted back to the right size. Inputs can be interpreted as signed or unsigned, and big-endian or little-endian values. Outputs are converted back by taking the lower N bits of the 128-bit output of the expression. When a destination is a byte sequence, the output can either be encoded to bytes as big-endian or as little-endian.

5 Results

We analyzed the x86-64 CPUs listed in Table 4. The AMD 3900X and AMD 7700X use the Zen 2 and Zen 4 microarchitecture respectively. The Intel i9-13900 has two different kinds of cores. Performance cores (A_2) use Raptor Cove, while efficiency cores (A_3) use the Gracemont microarchitecture. The Intel Xeon 4110 Silver uses the Skylake microarchitecture, a predecessor of Raptor Lake.

Table 4. Overview of the CPUs we analyzed.

	Name	Microarchitecture
A_0	Ryzen R9 3900X	AMD - Zen 2
A_1	Ryzen R7 7700X	AMD - Zen 4
A_2	Core i9-13900 (p)	Intel - Raptor Cove
A_3	Core i9-13900 (e)	Intel - Gracemont
A_4	Xeon Silver 4110	Intel - Skylake

Table 5 provides an overview of some of the results. Per architecture, it provides the number of generated encodings and the total time it took to run libLISA.

Table 5. Overview of the results.

	Encodings	Synthesized	Undocumented	Runtime	C_{in} (%)	C_{out} (%)	$C_{\text{random,in}}$ (%)
A_0	118 025	106 002	2569	7 weeks	99.99	97.36	99.99
A_1	118 019	105 205	2570	14 weeks	99.99	97.36	99.99
A_2	118 135	105 666	0	21 weeks	99.99	97.36	100.00
A_3	117 605	105 445	0	13 weeks	99.99	97.36	99.99
A_4	117 229	104 499	0	14 weeks	99.99	97.36	100.00

5.1 Validation

To validate our approach, we aim to answer two questions:

(1) Do libLISA's encodings cover all instructions on the CPU?
(2) Is libLISA able to synthesize semantics for undefined behavior?

Each of these questions is impossible to answer definitively, without access to an oracle that provides the ground truth (e.g., a trustworthy hardware design). Such oracles do not exist for x86-64 CPUs, e.g., there is no trustworthy complete overview of the set of all valid instructions for each of the CPUs. We therefore in this section devise *best-effort oracles* and answer these questions relative to those oracles. In that section, we provide an evaluative comparison to related work instead of relative to a best-effort oracle.

5.1.1 Instruction Coverage. As best-effort oracle, we generate lists of instructions. The first list consists of instructions extracted from the Linux binaries `ls`, `libxul.so`, `grep`, `gcc`, `ls`, `perl` and `ssh`. This generally produces documented instructions, although these instructions are not always valid instructions on the CPU that is being analyzed.

From the Linux binaries we extracted 3 190 703 instructions. On average, these instructions covered 4975 distinct encodings. Per architecture, we compute the *in-scope coverage* C_{in} as the percentage of instructions in enumeration scope from the oracle that are covered by an encoding. The out-of-scope coverage C_{out} is computed as the percentage of all oracle-instructions that are covered. Table 5 presents results. On average, libLISA achieves 99.99% in-scope coverage. Uncovered instructions consist of instructions with the EVEX prefix, which are not supported by any of the architectures we tested, and the SHA instructions, which are not supported by A_4. These instructions are conditionally executed only when CPU support is detected.

The second approach is to randomly generate byte sequences. This discovers instructions that can be undocumented or not commonly used in real-world programs, but it is biased to simpler instructions. We randomly generated an average of 7 719 372 instructions per architecture, covering 8053 encodings on average. Table 5 shows the coverage $C_{\text{random,in}}$, the percentage of in-scope oracle-instructions discovered by libLISA, which is 99.9% on average.

5.1.2 Undefined Behavior. We aim to determine how well our approach is able to synthesize semantics for undefined behavior. By determining the encodings which have undefined behavior using a best-effort oracle, we can compute the percentage for which libLISA has successfully synthesized semantics.

The best-effort oracle is a manual translation of the undefined behavior specified in the Intel Reference Manual to a machine-readable specification. This specification relies on the Intel XED disassembler library to map bitstrings to instruction variants listed in the Intel Reference Manual.

There is no one-to-one mapping from instruction variants produced by the Intel XED disassembler library to encodings. This makes it unfeasible to delineate exactly which parts of the instruction space covered by an encoding exhibit undefined behavior. We therefore define the term "undefined behavior" conservatively: if even one instruction covered by the encoding has at least one input for which at least one output is undefined, we consider the encoding to have "undefined behavior".

We determine if an encoding has "undefined behavior" by randomly sampling instructions covered by the encoding. Then, we query the oracle separately for each sampled instruction. If the oracle determines that at least one of the sampled instructions exhibits undefined behavior, we consider the encoding to have undefined behavior.

The results are shown in Table 6. On average, 90% of the encodings marked as having undefined behavior by the oracle were synthesized.

Table 6. The number of encodings with undefined behavior that libLISA was able to synthesize.

	Synthesized	Encodings	Percentage
A_0	18 548	20 316	91.2%
A_1	18 087	20 204	89.5%
A_2	17 805	19 578	90.9%
A_3	18 123	20 105	90.1%
A_4	17 699	19 767	89.5%

5.2 Comparisons with Existing Work

In this section we aim to answer two questions:

(1) Do our semantics cover the same instructions semantics provided by related work?
(2) Are our semantics correct relative to semantics provided by related work?

As related work, we consider the work of Dasgupta et al. [8] (see Section 3 for a more in-depth discussion on related work). The work provides a mapping of *instruction variants* to semantics. Examples of instruction variants are `ADD R8, IMM8` and `XCHL R32, EAX`.

Instruction variants do not easily translate to encodings, or the other way around. An instruction variant is not a subset of an encoding, nor is an encoding a subset of an instruction variant. It is also not possible to query all bitstrings that are described by a certain instruction variant. This makes it impossible to compare instruction variants and encodings directly.

5.2.1 Approach. To map encodings to variants, we randomly pick instantiations (i.e., concrete bitstrings). For each encoding, we generate 10 000 instantiations and filter them such that there are at most three instantiations with the same mnemonic, operand types and operand equality, according to `objdump` (also used by Dasgupta et al.). Then, we find the right instruction variant in the related work (if it exists) and instantiate it. We export both semantics to the SMTLib format [4], and check for equivalence using Z3 [9].

It is difficult to export Dasgupta et al.'s semantics, which are specified in the K framework [24], to SMTLib format. Dasgupta et al. have implemented a conversion to the SMTLib format by constructing a program containing a single instruction followed by `RETQ`, then recompiling the semantics, executing the K prover on the program, extracting the last K state from the output log, converting this K state to Z3 by parsing the state, generating a Python program that uses the Z3 library to reconstruct the expressions, and then running the Python program. This process is too slow for 118 000 encodings.

To be able to extract semantics quickly, we have written a minimal parser and rewrite engine that can process the original K semantics. There are 62 variants which we were unable to process. Four variants contain incorrect rules (`LOOPNE`, `[V]PCMPISTR[I/M]`). We exclude the `CLD`, `STD` variants because these variants use the direction flag, which libLISA does not synthesize. The rest of the 62 variants require functionality that we did not implement, e.g., late-evaluation of RSP in memory addresses.

It is not always clear which variant should be picked. Dasgupta et al.'s semantics do not specify immediate value sizes. They attempt to fix this during compilation by always deleting the `imm8` variants of instructions that also have `imm32` variants, forcing the `imm32` variant to be used. However, their list of instructions is incomplete (for example, `sbb_r32_imm8` is missing) We instead inspect variable names to determine the intended size of immediate values. The semantics also contain overlapping variants (e.g., `shl_r32_one` and `shl_r32_imm8`), we use a heuristic to score variants and select the most applicable variant.

Some of Dasgupta et al.'s instruction variants are aliases of other variants. We copy the comparison results from the variant returned by the disassembler to other aliases.

Table 7. Comparative results. We were unable to compare semantics from 62 variants of Dasgupta et al.'s semantics.

	A_0	A_1	A_2	A_3	A_4
Variants from Dasgupta et al. that…					
…agree with libLISA	1539	1469	1558	1536	1493
…disagree with libLISA	32	28	25	23	21
Dasgupta et al. incorrect	28	24	18	18	18
libLISA incorrect	4	4	7	5	3
…are incorrectly specified	6	6	6	4	4
…are out of enumeration scope for libLISA	744	744	744	744	744
…are enumerated but not synthesized by libLISA	635	709	623	649	694
…are not discovered by libLISA's enumeration	0	0	0	0	0
Encodings found by libLISA that…					
…are not covered by Dasgupta et al.	11 027	11 308	11 304	11 426	10 853

5.2.2 Comparison Results. Table 7 provides the results. A variant *agrees with* libLISA when for all concrete bitstrings generated from the encodings, Z3 is able to prove equivalence between libLISA and Dasgupta et al.'s semantics. A variant *disagrees with* libLISA when this is not the case. We discuss the differences between libLISA and Dasgupta et al. in more detail in the rest of this section.

Disagreements (Dasgupta et al. incorrect). We identify errors in 28 variants of the semantics of Dasgupta et al.:

(1) The overflow flag (OF) of `RCLB`/`RCRB` is undefined when the masked rotate count is not 0 or 1. However, Dasgupta et al. specifies the OF as undefined when the masked rotate count *modulo the operand size + 1* is not 0 or 1. (10 variants)
(2) In `vmpsadbw_xmm_xmm_m128_imm8`, the result is written to the source operand (R3) instead of the destination operand (R4). (1 variant)
(3) In all bit test variants (`BT`/`BTS`/`BTR`/`BTC`) on memory with a register bit offset, the bit offset is computed incorrectly. The bit offset is converted to a byte offset by shifting right by 3, then zero-extending the result to 64 bits. It should be sign-extended, to preserve the sign bits of negative offsets. (8 variants)
(4) The `CMPS` variants perform a comparison by setting flags according to $m_2 - m_1$, but they should be set according to $m_1 - m_2$. (6 variants)
(5) The instruction `XCHGL EAX, EAX` can be encoded as both `87C0` and `90`. The second encoding has the semantics of `NOP` (do nothing), while the first has the semantics of `XCHGL` (set the upper 32 bits of `RAX` to zero). The disassembler used by Dasgupta et al. incorrectly disassembles `90` with a `REX` prefix as `XCHGL` instead of `NOP`. (1 variant)
(6) The `MULX` instructions write a result to two destination operands. The destination operands can be equal. Dasgupta et al.'s semantics have not taken this possibility into account, causing the K prover to crash when `MULX` with equal destination operands is executed. (2 variants)

Disagreements (libLISA incorrect). Three to seven disagreements are errors in the semantics generated by libLISA. Our synthesis accepts semantics as correct when a hypothesis is correct w.r.t. two million consecutive random observations. In rare cases, these observations do not encompass all behavior of the instruction, which can lead to incorrect semantics. We see that the same kind of instruction is more often synthesized incorrectly across different architectures, but the exact variant differs. For example, A_0 synthesized `adc_rax_imm32` incorrectly, while A_1 synthesized that variant correctly and synthesized `adcq_r64_imm32` incorrectly instead. This indicates that these errors could be prevented by increasing the amount of random observations, or by improving the quality of the random observations. Some examples of errors are:

(1) The zero flag of the `SHLDQ M64, R64, 0x9` instruction is synthesized incorrectly for A_0. The semantics correctly check that the part of the result from `M64` is zero, but incorrectly check only the lower 8 instead of 9 bits shifted in from `R64`. This happened because the synthesizer did not encounter any cases where the lower 8 bits were 0, but the 9th bit was 1.
(2) The overflow flag of the `ADC` instruction is synthesized incorrectly. The overflow flag being 1 is relatively rare, and the synthesizer has not seen enough cases to form a good hypothesis.
(3) The zero flag of the `VPTEST` instruction with identical operands is synthesized as always zero. The synthesizer did not encounter any cases where all 256 bits of the register were zero, and has therefore not seen any evidence that the zero flag can be non-zero.

Incorrect specifications. There are `rel32` variants of the `JRCXZ` and `JECXZ` instructions, but these should only have `rel8` variants. One variant of `VCVTDQ2PD` accepts two `YMM` operands, while it should accept one `XMM` and one `YMM` operand instead. Two variants, `vcvtpd2ps_xmm_m256` and `vcvttpd2dq_xmm_m256`, have `m128` variants that always match the same instructions. Instructions from both the `m128` and `m256` variants will incorrectly use the semantics of only one of the two variants. Finally, the `vpinsrq_xmm_xmm_m64_imm8` variant is a copy of `vpinsrq_xmm_m64_imm8`. This is incorrect, as this variant should accept one more `XMM` register.

Out-of-scope. 744 variants are out of enumeration scope for libLISA, as described in Section 2.3. Most variants in this category are non-`VEX` versions of SSE/AVX instructions, which re-use the data size override prefix `66`. For around 396 out-of-scope variants, similar variants without the data size override prefix agree with libLISA's semantics. This means that by extending scope and doubling runtime, libLISA would be able to generate semantics where around 1900 variants would agree.

Synthesis failure. For 623 to 694 variants synthesis failed. This concerns mostly floating point operations, for which we did not implement support in our synthesis.

Not covered by Dasgupta et al. These encodings include both instructions that Dasgupta et al. considered out-of-scope, and instructions that Dasgupta et al. considered in-scope. Examples of variants outside Dasgupta et al.'s scope are: undocumented instructions, instructions operating on the MMX registers, instructions using segment registers, and recent ISA extensions like the SHA1 and SHA256 instructions. Missing variants within Dasgupta et al.'s scope include: YMM variants for the `VPSIGNB`/`VPSIGNW`/`VPSIGND`/`VPMINSW` instructions, `vcvtdq2pd_xmm_ymm` and `retq_imm`.

5.3 Use Cases

We demonstrate the feasibility of three use cases:

- comparing CPU-implementations
- discovering and analyzing undocumented instructions
- emulating userspace binaries

5.3.1 Comparing CPU Implementations. By comparing the semantics of different CPU microarchitectures, we can find interesting differences. We describe the differences between the semantics we found in Table 8.

Most encodings are part of group 0, which have identical semantics across all 5 architectures. All other groups show differences between architectures. Architecture 0 and 1 often share the same semantics. This is likely because architecture 1 is a successor of architecture 0. Similarly, architecture 2 and 4 also often share semantics, and architecture 2 is also a successor of architecture 4.

Group 1, 3, 5, 20 and 28 consist primarily of differences between microarchitectural implementations. When manually inspecting these groups we find bit shifts, rotates, multiplication, division and bit manipulation instructions. These are common instructions that can exhibit undefined behavior.

Groups 2, 6-9 and 19 show differences in instruction support. Upon manual inspection, we found that group 2 contains the undocumented AMD-only instructions. Group 6 contains the SHA1 and SHA256 instructions, which were introduced after architecture 4 was released. Group 7, 8 and 9 contain various other instructions introduced in x86-64 ISA extensions. Group 19 contains the `VMCALL` virtualization instructions, which are Intel-only.

We can also use this table to construct fingerprinting programs. Such a program can identify the architecture it is running on, among all the analyzed architectures. For example, to distinguish between the five architectures we analyzed, we could use group 1 and group 7. By observing an instruction from group 1, we can distinguish between A_0, A_1 or A_2, A_4 or A_3. Then, by also observing group 7, we can distinguish between A_0 and A_1 or A_2 and A_4. This would require executing at most 3 instructions and some logic to decode the result.

5.3.2 Discovery and Analysis of Undocumented Instructions. Undocumented instructions are valid instructions which have not been specified by the manufacturer in their documentation. For

Table 8. Architecture comparison. Each row describes a group of instructions that differ in a certain way between architectures. Each symbol represents a different implementation for that specific group of instructions. For example, there are two implementations for the instructions in group 5: A_0 and A_1 share the same implementation, and A_2, A_3 and A_4 share another implementation. We re-use the same symbols for each row. When a cell is left blank, this indicates a missing implementation. For example, the instructions in group 2 are only supported by A_0 and A_1.

Group	Encodings	A_0	A_1	A_2	A_3	A_4
Group 0	95 170	○	○	○	○	○
Group 1	4777	○	○	□	△	□
Group 2	2571	○	○			
Group 3	1602	○	○	□	○	□
Group 4	604	○	□	△	▽	⊗
Group 5	581	○	○	□	□	□
Group 6	101	○	○	○	○	
Group 7	40		○	○	○	
Group 8	29			○	○	
Group 9	24					○
Group 10	16	○	○	○	○	□
Group 11	16	○	□	□	□	□
Group 12	12	○	○	□	○	○
Group 13	9	○	○	○	□	○
Group 14	7	○	□	○	○	○
Group 15	5	○	□	○	□	○
Group 16	5	○	□	□	○	△
Group 17	4	○	□	□	○	○
Group 18	3	○	○			○
Group 19	2			○	○	○
Group 20	2	○	□	△	▽	△
Group 21	1		○			
Group 22	1		○	○	○	○
Group 23	1	○				○
Group 24	1	○		○	○	○
Group 25	1	○	○		○	○
Group 26	1	○	○	□	□	○
Group 27	1	○	□	○	△	△
Group 28	1	○	□	△	△	△
Synthesis failed	14036					

example, on Intel CPUs there used to be an undocumented instruction `D6`, which was only added to the Intel reference manual as the `SALC` instruction in 2017.

We use the Intel XED disassembler library as an oracle to determine whether instructions are documented. To eliminate false positives, we manually verified the undocumented instructions we found against the Intel and AMD reference manual and `objdump`.

The results are listed in Table 5. We identified one group of undocumented instructions on AMD CPUs. We did not identify any undocumented instructions on Intel CPUs. It is possible that our results are favoring Intel CPUs because we are using a disassembler library created by Intel. While we were able to manually confirm that all undocumented instructions are indeed undocumented, there might be undocumented instructions on Intel CPUs that XED incorrectly decodes successfully.

The semantics of the group of undocumented instructions on the AMD CPUs match the semantics of the `VPERMQ` instruction. The `VPERMQ` expects `VEX.W` to be 1. These undocumented instructions are bit-for-bit identical with `VPERMQ` variants except for `VEX.W`, which is 0. We suspect that the decoding logic for the VEX prefix does not check the value of `VEX.W`, and causes the instructions to be treated as if they were valid `VPERMQ` instructions.

5.4 Emulating Userspace Binaries

We implement a proof-of-concept emulator that uses libLISA's semantics as-is to emulate x86-64 ELF binaries. The emulator uses the semantics from the 118 000 encodings, and runs encoding analysis and synthesis on-the-fly for instructions that are out-of-prefix-scope. This makes it possible to fully emulate some Linux binaries that do not use floating-point instructions. Table 9 presents the binaries we have successfully emulated on an AMD 3900X.

The emulator stores the emulated CPU state in a data structure. During execution, it repeatedly modifies the emulated CPU state. To execute an instruction, it reads memory at the address stored in the emulated `RIP`. The semantics for the instruction are found by searching through libLISA's semantics. If no suitable semantics are found, on-the-fly encoding analysis and synthesis is invoked.

The semantics are executed by fetching inputs, computing results, and then storing results. First, the values of all sources in the dataflows are fetched by reading the value from the data structure storing the emulated CPU state or memory. Then, the new values for all destinations are computed using the synthesized computations in the semantics. Finally, the new values are written to the destinations.

The emulator provides handwritten implementations for the `SYSCALL`, `XGETBV` and `CPUID` instructions. Additionally, we treat four additional instructions as `NOP`s: `ICEBP`, `RDTSC`, `XSAVEC`, and `XRSTOR`. For all other instructions we use semantics generated by libLISA.

We verify the emulated instructions against the real CPU behavior using the CPU observer. For each instruction we emulate, we compute the next CPU state, and then observe the real next CPU state. If these differ, we abort execution. In total, we have verified 1 244 385 instruction executions against the real CPU behavior.

Table 9. Binaries that we are able to emulate successfully on an AMD 3900X CPU.

Binary	Number of instructions
Hello world	98 813
`/bin/true`	120 969
`/bin/ls /dev/null`	321 084
`/bin/ls -hla /dev/null`	401 037
`/bin/date`	172 827
`/bin/echo 'abc'`	129 655

We emulate the binary itself, the dynamic linker (`/lib64/ld-linux-x86-64.so.2`), and all dynamically loaded binaries (e.g., `libc.so`). This reduces the manual implementation effort, as we can rely on the dynamic linker to link dependencies automatically, instead of having to implement these manually. It also significantly increases the amount of instructions that are executed for simple binaries. The hello world binary consists of one call to `printf`, consisting of around 100 instructions, with the rest of the executed instructions being in the dynamic linker and the C standard library.

6 Discussion and Conclusion

We presented libLISA, a tool for automated discovery of instructions executable on a given CPU, as well as synthesis of their semantics. At its core, libLISA is based on fuzzing, which means that synthesis may produce incorrect semantics for an instruction. We evaluate that this happens for about three to seven instruction variants per run. This could be reduced by increasing the amount of random inputs that are generated, or by changing the distribution of the random numbers that are generated such that the rare cases are more observable. However, it is difficult to do this in a way that generalizes to any instruction. We argue that this is inherent to bottom-up derivation of instruction sets and their semantics. Without access to an oracle that provides a trustworthy and formal semantics of the entire instruction set of a CPU, libLISA provides the most extensive and trustworthy formalization of the x86-64 ISAs to date.

Trustworthy semantics are the base of any binary-level effort. For example, BAP [6] (Binary Analysis Platform) has – for the x86 architecture – a manually written translation from instructions into an intermediate language. They test their semantics against a CPU. It would be interesting to combine libLISA's semantics with BAP, reducing the trusted code base.

We believe libLISA's approach would work on other modern general-purpose CPU architectures. Only the CPU observer is architecture-specific, and would have to be implemented from scratch for a new architecture. Architectures such as ARM or RISC-V would be suitable candidates. With these architectures, a bit in an instruction often also serves a single purpose.

We ran our analysis on an Intel i9-13900 CPU, which was affected by the Reptar bug.[1] Unfortunately, the REP prefix is part of the instruction prefixes we consider out of scope (see Section 2.3). We argue that libLISA would have discovered this bug if the REP prefix was considered in-scope. The main characteristic of this bug is that it causes non-deterministic output. This would be picked up upon during dataflow analysis, where the entire CPU state would be considered input. Such instructions (e.g., RDTSC or RDRAND) are flagged automatically because synthesis is known to fail.

In Table 7, there are around ten fewer disagreements with Dasgupta et al. on Intel CPUs than on AMD CPUs. This is because of the incorrect specification of undefined behavior in the RCLB/RCRB variants. The behavior specified by Dasgupta et al. matches Intel's implementation of this undefined behavior, not AMD's. This highlights the importance of verifying specifications on multiple CPUs.

We only discover and analyze userspace instructions. The primary reason for this is that the CPU observer needs to execute instructions in ring 3 (userspace) for sandboxing. It would be possible to analyze privileged instructions by implementing a CPU observer as hypervisor. Whether automatic analysis would be effective for privileged instructions is unclear. In userspace, we can reasonably assume that an instruction, for example, does not change CPU configuration, modify the page table base register PTBR or unmap memory. Our analysis depends on these assumptions. Significant, non-trivial modifications are needed to analyze privileged instructions.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which have greatly improved the paper.

This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028.

[1]https://lock.cmpxchg8b.com/reptar.html

Data-Availability Statement

The analysis results are available in an easily browsable format on https://explore.liblisa.nl/. The libLISA implementation is available under the AGPLv3 open source license on https://github.com/liblisa. A self-contained reproduction package is available on Zenodo [7].

References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017)*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93. https://doi.org/10.1145/3208071

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (jan 2019), 31 pages. https://doi.org/10.1145/3290384

[4] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[5] Andrew Barrington, Steven Feldman, and Damian Dechev. 2015. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. Association for Computing Machinery, New York, NY, USA, 1321–1328. https://doi.org/10.1145/2695664.2695924

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

[7] Jos Craaijo, Freek Verbeek, and Binoy Ravindran. 2024. Reproduction package (Docker container) for the OOPSLA 2024 Article "libLISA: Instruction Discovery and Analysis on x86-64". Zenodo. https://doi.org/10.5281/zenodo.13380062

[8] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. 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* (Phoenix, AZ, USA) *(PLDI '19)*. Association for Computing Machinery, New York, NY, USA, 1133–1148. https://doi.org/10.1145/3314221.3314601

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] Rens Dofferhoff, Michael Göebel, Kristian Rietveld, and Erik van der Kouwe. 2020. iScanU: A Portable Scanner for Undocumented Instructions on RISC Processors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)*. 306–317. https://doi.org/10.1109/DSN48063.2020.00047

[11] Christopher Domas. 2017. Breaking the x86 ISA. https://github.com/xoreaxeaxeax/sandsifter Accessed on 02/05/2024.

[12] Patrice Godefroid and Ankur Taly. 2012. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/2254064.2254116

[13] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Lausanne, Switzerland) *(FMCAD '14)*. FMCAD Inc, Austin, Texas, 91–98. https://doi.org/10.1109/FMCAD.2014.6987600

[14] Niranjan Hasabnis and R. Sekar. 2016. Extracting instruction semantics via symbolic execution of code generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/2950290.2950335

[15] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[16] Nathan Jay and Barton P. Miller. 2018. Structured random differential testing of instruction decoders. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 84–94. https://doi.org/10.1109/SANER.2018.8330199

[17] Doug Kwan, Kostik Shtoyk, Kostya Serebryany, Maxim L Lifantsev, and Peter Hochschild. 2021. SiliFuzz: fuzzing CPUs by proxy. *Google Research* (2021). https://doi.org/10.48550/arXiv.2110.11519

[18] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[19] Xixing Li, Zehui Wu, Qiang Wei, and Haolan Wu. 2019. UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching. *IEEE Access* 7 (2019), 149224–149236. https://doi.org/10.1109/ACCESS.2019.2946444

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

[21] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/1572272.1572303

[22] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 395–404. https://doi.org/10.1145/2254064.2254111

[23] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 161–168. https://doi.org/10.1109/FMCAD.2016.7886675

[24] Grigore Roșu and Traian Florin Șerbănută. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012 Membrane computing and programming.

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

[26] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (jul 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[27] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[28] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5341–5358.

[29] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (jun 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

Received 2024-04-04; accepted 2024-08-18