InstrSem: Automatically and Generically Inferring Semantics of (Undocumented) CPU Instructions

Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern CPUs implement complex Instruction Set Architectures (ISAs), yet machine-readable semantics are often incomplete. Worse, many CPUs support undocumented instructions, i.e., bitstrings that execute on hardware but are absent from specifications, leading to potential security vulnerabilities.

In this paper, we present InstrSem, an ISA-agnostic, modular, fully automated approach to infer instruction *semantics* from execution behavior alone and provide semantics that are understandable by both, humans and machines. Starting from a raw encoding, InstrSem executes it under systematically varied architectural states and synthesizes compact mathematical functions that explain every changed state component. By mutating encoding bits and correlating induced behavioral changes with bit positions, InstrSem then *generalizes* from a single encoding to a full instruction, recovering register and immediate fields. In contrast to prior work focusing on a single ISA, InstrSem is generic. It requires only a lightweight ISA model and a per-architecture user-space runner and supports fixed- and variable-length encodings (RISC and CISC), memory accesses, and conditional behavior. We evaluate InstrSem on RV64I, AArch64, and LA64, and additionally showcase CISC applicability on a Logitech macro language and partial x86-64. InstrSem automatically recovers correct semantics for over 97.81 % of the RV64I base instruction set, and 136 instructions covering 1 009 055 744 instruction encodings within 77 h for the LA64 instruction set. InstrSem discovers undocumented vector instructions, inconsistencies between QEMU and Loongson hardware, and instructions that crash QEMU. InstrSem enables scalable recovery of instruction semantics, substantially automating reverse engineering across commodity and niche targets and strengthening the foundations for emulation, verification, and security analysis. With minimal requirements to support new architectures, its modular design, and human-readable output, InstrSem can aid future security analysis.

1 Introduction

Modern CPUs implement highly complex and evolving ISAs. These ISAs define how software can interact with hardware, specifying instruction formats, encodings, and semantics. While ISA manuals often describe the behavior of standard instructions in a human-readable form, they typically lack formal, machine-readable semantic specifications [21]. This absence hinders automated analysis, tool generation, and formal reasoning. Moreover, ISA documentation may not reflect the true behavior of hardware: Many CPUs include undocumented instruction encodings – bitstrings that are executable but not described in any specification [10, 11, 32, 33].

However, such undocumented encodings matter for security. These instructions are invisible to disassemblers, static analyzers, and formal verification tools. Moreover, ISA-aware fuzzers typically ignore them [31]. Recent research has shown that undocumented instructions have enabled privilege escalation [12] on x86, denial-of-service on x86 [28] and RISC-V [33], and bypassed memory protections on RISC-V [33]. Still, despite the critical security impact on non-x86 ISAs, previous work primarily focused on x86 [7, 21], where vast human effort was spent on formalizing instruction semantics.

Unfortunately, recovering the semantics of undocumented instructions is a difficult and largely manual task. What is missing is a generic and *modular* way to recover semantics across architectures, without having to tailor the approach to a specific ISA. Existing techniques address only parts of the problem. Fuzzing-style discovery identifies encodings that execute without crashing [10, 11, 23, 32, 39], but cannot explain *what* they do. Program-synthesis approaches, such as STRATA [21] and libLISA [7] infer formal semantics for x86 instructions but rely on hundreds of ISA-specific templates or complex setups which are hard to port. Recent reverse-engineering efforts on RISC-style instruction sets, such as AMD microcode [13], Intel ME [14], VIA C3 [12], NVIDIA GPUs [19], and keyboard macros [34], required vast manual efforts even for subsets of the ISA. A practical solution must be able to generically add a new architecture, whether that is a commodity ISA or a proprietary macro language.

In this work, we present an ISA-agnostic approach that automates semantic reverse engineering from execution behavior alone and is explicitly designed for extensibility and modularity. We start from a raw, unknown instruction encoding and infer a precise, machine- and human-readable description of its behavior, and then generalize from that single encoding to a full instruction with operands. We state two research questions:

Can we automatically infer the semantics of arbitrary instruction encodings without relying on existing documentation or templates? Can we generalize individual instruction encodings into a formal description of how instructions map bits to behavior?

We answer these questions with a black-box approach. Given a single instruction encoding and a description of the architectural state (e.g., registers, memory), we execute the encoding under different architectural inputs and record the resulting outputs. We then synthesize compact mathematical functions that explain each changed component of the state. These functions form the instruction encoding's semantics: they describe how outputs (e.g., destination registers, memory) depend on inputs (e.g., source operands, immediate values). To generalize to instruction semantics, we mutate the encoding bits and correlate induced semantic changes with specific bit positions to recover which fields encode registers and immediates. This requires no disassembler, specification, or predefined architecture-dependent instruction templates, and works fully automated with user-space runners. Our assumptions are minimal: a runner that can execute an encoding and capture the architecturally visible state, and a lightweight ISA model that lists state component names, bit-widths, and basic constraints (e.g., canonical addresses or a zero register).

We implement this approach in a modular prototype, InstrSem. InstrSem is ISA-agnostic by design, requiring only the minimal ISA model and a per-architecture runner. This modular interface makes it easy to support new targets, as only a runner and a minimal model need to be provided. InstrSem supports fixed- and variable-length encodings and thus applies to both RISC and CISC ISAs. It supports instructions that access memory and handles conditional behavior and function synthesis over a generic set of integer bit-vector operations. We evaluate InstrSem on three widely used real-world ISAs: RISC-V (RV64I), ARMv8-A (AArch64), and LoongArch64 (LA64) and additionally showcase CISC applicability on the Logitech macro language and partial x86-64. On documented instructions, it automatically recovers correct semantics for over 97.81 % of the RV64I base instruction set. When applied to the entire LA64 encoding space, InstrSem correctly reverse-engineers 136 instructions covering 1 009 055 744 instruction encodings within 77 h, discovers undocumented vector instructions, inconsistencies between QEMU and Loongson hardware, and instructions that crash QEMU, affecting sandboxing security. InstrSem further uncovers 305 undocumented instructions on the SiFive P550 and automatically reverse-engineers Apple's proprietary Mul53 extension. With only slight modifications to the RISC-V runner, InstrSem can automatically reverse-engineer the semantics of GhostWrite [33] and generalize it into an instruction covering 65 536 encodings, demonstrating it can be a useful tool for security analysis. For the Logitech macro language, we successfully recover the semantics of 13 instructions. Our results show that a large portion of the reverse engineering process for instruction semantics can be fully automated, even for undocumented or non-standard instruction sets.

Our contributions can be summarized as follows:

1. We introduce a generic, automated approach to infer instruction encoding semantics from input-output behavior alone, without templates or documentation.
2. We propose an algorithm to recover instruction semantics by mapping instruction encoding bits to semantics.
3. We present InstrSem, a modular implementation of this approach that supports multiple ISAs, and makes it easy to add new architectures, including more obscure architectures such as macro languages.
4. We evaluate InstrSem on RV64I, AArch64, and LA64, recovering semantics for thousands of instruction encodings and discovering undocumented behaviors in commodity CPUs from Apple, SiFive, and Loongson.
5. We demonstrate how InstrSem can aid security analysis by reversing GhostWrite using InstrSem.

Responsible Disclosure. We reported the QEMU crashes and QEMU-only instructions to QEMU. The issues were addressed and fixed.

Availability. We will open-source InstrSem upon acceptance of the paper and provide it as an artifact for the review process.

2 Background

2.1 Instruction Set Architectures

An *Instruction Set Architecture* (ISA) defines the interface between software and hardware [20]. It specifies the set of instructions a CPU understands, how these instructions are encoded as bitstrings, and how they affect the architectural state, such as registers and memory. Each instruction is a fixed-width bitstring, with specific fields identifying the operation (e.g., `ADD`) and its operands (e.g., registers or immediates).

While ISA manuals provide high-level descriptions of these instructions, they are usually written for human readers. Machine-readable semantics that precisely define the behavior of instructions are rarely available. Moreover, CPUs often implement *undocumented* instructions: bitstrings that the CPU accepts and executes, but that are not described in the official ISA documentation. Such instructions may have legitimate purposes (e.g., testing, debugging, vendor extensions) but can also pose security risks.

The mapping from an instruction's bitstring to its functional behavior is known as its *instruction encoding*. For example, an instruction that adds two registers and stores the result in a third might be encoded with fields specifying the source and destination registers and an opcode identifying the operation. An *architectural state* is a snapshot of the CPU's visible state. The architectural state is an abstraction of the microarchitectural state, comprising internal CPU states, e.g., cache state or physical register files. The *semantics* of an instruction describe how it transforms an architectural state.

Instructions vs. Instruction Encodings. This paper distinguishes between a concrete *instruction encoding* (or *encoding*) and a more general *instruction*. The instruction encoding is one instance of the instruction with specific register and immediate values. The instruction describes a family of encodings sharing the same format and semantics, with variations based on operand values. For example, the encoding is `addi $r0, $r0, 17`, with specific fields for the registers and immediates, while the instruction contains "placeholders" for these fields (`addi` $\texttt{reg}_\texttt{a}$, $\texttt{reg}_\texttt{b}$, `imm12`).

2.2 Constraint Solving and Z3

Constraint solvers, such as Microsoft's Z3 [9], are indispensable tools for many problems in planning [17, 37], resource allocation [1], and network analysis [3]. These solvers take logical formulas over bit vectors (fixed-width binary values) and determine whether assignments (known as models) exist that satisfy a given specification. Z3 can *verify* whether a constraint can be fulfilled and *find* values for symbolic constants that satisfy given constraints. In IT security, constraint solvers have successfully been utilized for symbolic execution and automatic exploit generation [5, 38].

2.3 CPU Vulnerabilities

With the growing complexity of CPUs, researchers have uncovered numerous attacks targeting hardware instead of software bugs. One class of such attacks are microarchitectural side channels that can leak secrets across security boundaries [6, 15, 25, 30]. With Meltdown [24] and Spectre [22], a new class of CPU vulnerabilities called transient execution attacks was discovered. Those attacks target side effects of transient instructions that are erroneously executed due to out-of-order or speculative execution. Since their inception, a multitude of transient execution attacks has been unveiled [4, 22, 24, 29, 35, 36]. However, recent research has discovered a growing number of architectural bugs [2, 28, 33, 40]. Such bugs do not require observing microarchitectural effects through side channels but directly manifest in the architecturally observable state, e.g., malformed or undocumented instructions affecting register or memory contents [28, 33].

3 Inferring Semantics Automatically

This section introduces our generic approach to automatically infer the semantics of instruction encodings and generalize them into instruction semantics. Section 3.1 briefly introduces the general idea. Section 3.2 formally describes the primitives used in our approach. Section 3.3 describes how encoding semantics can be recovered. Section 3.4 shows how encoding semantics can be generalized to instruction semantics. Section 3.5 outlines implementation challenges to overcome.

3.1 Overview

At a high level, we infer an instruction encoding's semantics by executing an unknown encoding under different randomized architectural input states and observing the corresponding outputs. For each component of the state that changes, we attempt to synthesize a compact function that maps relevant inputs to the observed output.

The central idea is that many instruction behaviors, such as arithmetic, bitwise, or conditional operations, can be expressed with small, human-readable functions over a limited set of inputs. If we can find such a function that consistently predicts the output given the inputs, we assume it describes the encoding's semantics.

3.2 Primitives

Our approach relies on the primitives outlined in this section.

Architectural State and State Variables. An *architectural state* is a collection of values that can influence the semantics of an instruction. This state contains register values and the memory state. As obtaining the complete architectural state may be infeasible, our approach also works with a subset of it. Formally, we describe an architectural state A as a set of *state variables*. A state variable v is a tuple $(\text{name}, \text{value})$. "name" describes the name of an architectural component (e.g., register x_5), and "value" denotes the current value. For instance, $A_\text{sample} = \{(x_0, 0), (x_1, 5), (x_2, 3), (\text{address}_0, 100), (\text{memory}_0, 1823), (\text{address}_1, 200), (\text{memory}_1, 42)\}$ describes the architectural state of an architecture with three registers and two memory regions. We use square brackets when referring to the value of a state variable of an architectural state. Hence, $A_\text{sample}[x_1] = 5$ in the above example.

Output Function. An *output function* describes how a new state variable that changes due to instruction encoding execution can be derived from the architectural state. Intuitively, it describes how an output is calculated from a given architectural input state. Formally, an output function f is a function $f : A \mapsto \mathbb{N}$. For instance, $f_{\text{mov}\,x_1}(A) = A[x_1]$ describes the operation of copying the value from register x_1.

Encoding Semantics. *Encoding semantics* are a set of output functions that describe how the architectural state changes on instruction encoding execution. We omit functions for outputs that do not change and assume an identity function. Formally, encoding semantics are a set S of tuples $(\text{name}, \text{function})$. "name" describes the name of a changing architectural component, while "function" is an output function that describes how the new value can be derived. We refer to the architectural state A' that can be derived by applying all output functions of S to the initial architectural state A using $S(A)$:

$$A' = S(A) := \left\{ \begin{cases} (x, f(A)) & \exists f : (x, f) \in S \\ (x, A[x]) & \text{otherwise} \end{cases} \middle| (x, \_) \in A \right\}$$

Further, we refer to all occurrences of a specific register or immediate using triangular brackets. Consider an instruction encoding "example" that computes $x_2 = x_1 + x_1$ and zeroes x_1. This yields: $S_{\text{example}} = \{(x_1, f(A) = 0), (x_2, f(A) = A[x_1] + A[x_1])\}$, $S_{\text{example}} \langle x_1 \rangle = \{x_1, x_1, x_1\}$ and $S_{\text{example}} \langle 0 \rangle = \{0\}$.

Instruction Semantics. *Instruction semantics* generalize encoding semantics to entire instructions. They map from instruction encoding bitstrings to semantic descriptions. Formally, instruction semantics E are partial functions $E : \mathbb{N} \to S$, meaning that they are defined only for bitstrings that belong to a specific instruction. For instance, the instruction semantics below describe that the instruction bitstring $0b1010xxxx$ sets the register encoded in the least significant 4 bit ($xxxx$) to '0':

$$E_{zero}(i) = \begin{cases} \{(x_{i\%16}, f(A) = 0)\} & (i \gg 4) = 0b1010 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Runner. A *runner* is an architecture-specific execution harness that takes an initial architectural state A and an instruction encoding i, executes the encoding, and returns the resulting state A'. It serves as an oracle for encoding semantics and provides the ground truth for their recovery. Formally, a runner is a function $R : A \times \mathbb{N} \mapsto A$.

ISA Model. An *ISA model* describes the relevant properties of an ISA including the names and sizes of registers and how registers and immediates can be represented in the instruction bitstring (e.g., x_3 as `00011`).

3.3 Inferring Encoding Semantics

Under ideal conditions, we can identify the correct output function for each output by eliminating incorrect candidates through differential testing. For a given output name, we start a set of possible output functions F. Our algorithm makes no assumptions on how F is obtained. It only assumes that F is a set of valid, distinct (i.e., $\forall f, f' \in F : (\nexists A : f(A) \neq f'(A)) \implies f = f'$) output functions, and that a correct output function for each state variable is contained. We repeatedly pick two functions, f and f', from the candidate set F, find an input state A where they differ, execute the instruction encoding on A, and compare the actual output to the predictions from each function. Any function that disagrees with the observed result is removed from the candidate set. This process continues until only one function remains.

If F initially contains the correct function, the functions are all distinguishable, and we can always find a distinguishing input, this approach is guaranteed to identify the correct semantics. Algorithm 1 formalizes this procedure.

Our approach relies on one important assumption: Output functions of instruction encodings are usually mathematical operations that can be described with a compact mathematical description. Identifying the correct mathematical operation of an output function is feasible by covering only a tiny fraction of the possible architectural states. This intuitively holds if the covert architectural input and output states suffice to rule out all but one mathematical operation that an output function could implement.

Note that using a runner R and executing an instruction encoding i with all possible architectural states A and recording $A' = R(A, i)$ could trivially recover the encoding semantics S. However, even for architectures where the state solely consists of eight 32-bit registers and no memory, this would mean executing the encoding under test 2^{256} times and recovering a table with 2^{256} rows, which is infeasible in terms of computational power and memory.

Formal Description. Formally, our approach relies on a set of possible output functions F where no two functions of F are semantically equivalent ($\forall f, f' \in F : f \neq f' \implies \exists A : f(A) \neq f'(A)$). We assume a function correctly describing each output function of an instruction encoding i is contained in F. If these assumptions hold and a runner R is available, S can be recovered. This is done using the runner R on architectural states A where at least two possible functions disagree for an output *name*. Only the functions f that correctly describe how to derive *name* from A, i.e., $f(A, i) = R(A, i)[name]$, are kept in the set of possible functions. This step is repeated until only a single function for each output remains. Since we assume that a correct output function exists in F, the set of possible functions for an output is never empty. Further, as all functions of F are distinct, there is always a state A where $f, f' \in F$ with $f \neq f'$ leads to $f(A) \neq f'(A)$. Thus the set of possible functions is reduced in each iteration. While identifying such architectural states might not be straightforward, this generic algorithm assumes it is possible. An implementation could, for instance, rely on a SAT-solver for this. The set of all recovered output functions for each output then describes the encoding semantics. The pseudocode of this algorithm is outlined in Algorithm 1.

Relaxed Algorithm. In practice, the assumptions of the ideal algorithm (Algorithm 1) are difficult to guarantee. It is often infeasible to determine whether two functions are semantically equivalent or to identify distinguishing inputs. Furthermore, many functions may behave identically across most states. To overcome these issues, we implement a relaxed algorithm based on randomized testing. Instead of searching for differentiating inputs, we draw a fixed number of random architectural states from a domain-specific distribution. These samples are designed to expose edge cases such as overflows,

Algorithm 1: EncSem

```
Data: Set of possible output functions $F$, runner $R$,
      output names $O$, instruction encoding $i$
Result: Encoding Semantics $S$
$S \leftarrow \emptyset$ ;
for $name \in O$ do
    $F_{name} \leftarrow F$;
    while $|F_{name}| > 1$ do
        $f, f' \leftarrow choose\_any(F_{name})$;
        $A \leftarrow f(A) \neq f'(A)$;
        if $f(A) \neq R(A,i)$ then
            remove $f$ from $F_{name}$ ;
        end
        if $f'(A) \neq R(A,i)$ then
            remove $f'$ from $F_{name}$ ;
        end
    end
    $S \leftarrow S \cup \{(name, choose\_only(F_{name}))\}$ ;
end
```

Algorithm 2: EncSemRelaxed

```
Data: Set of possible output functions $F$, runner $R$,
      output names $O$, distribution of architectural
      states $D$, instruction encoding $i$
Result: Encoding Semantics $S$
$S \leftarrow \emptyset$;
for $name \in O$ do
    $F_{name} \leftarrow F$;
    for $n$ times do
        $A \leftarrow$ choose randomly from $D$;
        for $f \in F_{name}$ do
            if $f(A) \neq R(A,i)$ then
                remove $f$ from $F_{name}$ ;
            end
        end
    end
    add any $(name, f \in F_{name})$ to $S$;
end
```

sign transitions, and zero values. For each output, we keep only the functions in F that match the observed outputs for all sampled inputs. If multiple functions remain, we pick one arbitrarily.

This heuristic approach trades completeness for scalability and works effectively in practice, as shown in Section 5. Algorithm 2 summarizes this algorithm.

While this algorithm cannot guarantee correctness, it significantly reduces computational costs and supports large-scale analysis. Our evaluation shows that despite its heuristic nature, it reliably recovers accurate encoding semantics for diverse architectures and instruction sets.

3.4 Generalizing Encoding Semantics

Inferring the semantics of a single instruction encoding is only the first step. Our ultimate goal is to generalize this information into a description of the entire instruction, i.e., to understand how operand values (registers, immediate values) are mapped into the instruction bitstring and vice versa. In this section, we present an approach to generalize encoding semantics S for an instruction encoding i into instruction semantics E. This is done by determining which instruction bits encode immediate values and register numbers used in an output function. The algorithm allows us to identify a parameterized instruction format (e.g., `add reg`$_a$`, reg`$_b$`, reg`$_c$) and define a single instruction that represents many instruction encodings with similar behavior.

We again describe an idealized algorithm first and then introduce a relaxed variant used in practice. The intuition is simple: if modifying certain bits in the instruction bitstring causes corresponding changes in the recovered semantics, then those bits are likely encoding operand values. For example, if changing bits 10–14 results in a different destination register in the recovered semantics, we assume bits 10–14 encode the destination register.

Ideal Generalization Algorithm. The generalization to instruction semantics relies on a mapping from instruction encodings to encoding semantics $M : \mathbb{N} \mapsto S$. Such mapping can be computed lazily using the approach in Section 3.3. Given an encoding i and an ISA model, the algorithm can generalize the encoding semantics to instruction semantics. The algorithm uses a set of known encapsulations K_{known} which is initially empty. Such an encapsulation describes how an immediate or register is encoded in a subset of the instruction bitstring. The algorithm also computes a set of possible encapsulations $K_{possible} = \bigcup_{u \in U(s), b \in Enc(u,i)} \{(b,x) | x \in \mathcal{P}(s\langle u \rangle) \setminus \{\emptyset\}\}$ where $U(s)$ denotes all registers and immediates used in s and $Enc(u,i)$ describes all possible substrings of i that can encode u according to the ISA model. In each iteration of the algorithm, one entry k in $K_{possible}$ is removed and checked whether it actually encodes the register or immediate. The helper function *CheckEncoding* in Algorithm 5 (Appendix A) performs this checking. The algorithm computes all instruction encodings i' and corresponding expected semantics s' that are covered by the encapsulations $K_{possible} \cup \{k\}$. If the actual semantics $M(i')$ always matches the predicted semantics s', the encapsulation k is added to K_{known}. Otherwise, it is discarded. This is repeated until $K_{possible} = \emptyset$. Finally, instruction semantics constructed from K_{known} are returned. Since $K_{possible}$ is a finite set and one item is removed in each iteration, the algorithm is guaranteed to terminate. The pseudocode of the algorithm is outlined in Algorithm 3. Further, the semantics of all encodings covered by K_{known} are checked

Algorithm 3: InstrSem

```
Data: Mapping from instruction encodings to
      encoding semantics $M$, instruction encoding $i$
Result: Instruction Semantics $E$
$K_{known} \leftarrow \emptyset$;
$K_{possible} \leftarrow \emptyset$;
$s_{original} \leftarrow M(i)$ ;
for $name \in U(s_{original})$ do
    for $usages \in \mathcal{P}(s_{original}\langle name\rangle) \setminus \{\emptyset\}$ do
        for $b \in Enc(name, i)$ do
            add $(b, usages)$ to $K_{possible}$ ;
        end
    end
    for $k \in K_{possible}$ do
        if $CheckEncapsulation(K_{known} \cup \{k\}, M)$ then
            add $k$ to $K_{known}$ ;
        end
    end
end
$E \leftarrow To\_Semantics(K_{known})$ ;
```

for correctness in each iteration. Thus, the algorithm guarantees that the resulting instruction semantics are correct.

Relaxed Algorithm. The algorithm above requires recovering the encoding semantics of all instruction encodings of an instruction before the semantics can be reported. While this is necessary to ensure formal correctness, we propose a heuristic approach that only tests a small subset of the instruction encodings covered by the instruction. Intuitively, if some bits of i correctly encode a few randomly chosen immediates or registers, it is likely that these bits are actually used to encode the immediate or register. Further, we also incorporate the idea behind the relaxed algorithm for recovering encoding semantics and do not compute $M : \mathbb{N} \mapsto S$ but instead test the expected encoding semantics s' against randomly chosen architectural input states A using the runner. If $s'(A) = R(A, i')$ for all tested states, we assume s' is correct. The pseudocode of the relaxed algorithm is shown in Algorithm 4. With these heuristic optimizations, the runtime of each iteration is significantly reduced since only a fixed number of instruction encodings is tested instead of all encodings covered by the instruction. While this relaxed approach cannot guarantee correctness, our empirical results (Section 5) show that it consistently recovers accurate instruction semantics. In practice, only a handful of samples per candidate are sufficient to validate an operand mapping, provided that the architectural input states are well-distributed.

3.5 Challenges for Implementation

While the previous sections outline a generic approach to automatically reverse-engineer semantics of instruction en-

Algorithm 4: InstrSemRelaxed

```
Data: Set of possible output functions $F$, runner $R$,
      output names $O$, distribution of architectural
      states $D$, instruction encoding $i$, amount of
      tested encodings $n_i$, amount of architectural
      states $n_a$
Result: Instruction Semantics $E$
$K_{known} \leftarrow \emptyset$;
$K_{possible} \leftarrow \emptyset$;
$s_{original} \leftarrow EncSemRelaxed(F, R, O, D, i)$ ;
for $name \in U(s)$ do
    for $usages \in \mathcal{P}(s_{original}\langle name\rangle) \setminus \{\emptyset\}$ do
        for $b \in Enc(name, i)$ do
            add $(b, usages)$ to $K_{possible}$ ;
        end
    end
    for $k \in K_{possible}$ do
        if
         $CheckEncapsulationRelaxed(R, D, K_{known} \cup$
         $\{k\}, n_i, n_a)$ then
            add $k$ to $K_{known}$ ;
        end
    end
end
$E \leftarrow To\_Semantics(K_{known})$;
```

codings and cluster them into instruction semantics, the high level of abstraction hides some challenges that must be overcome when implementing our approach. In this section, we outline these key challenges.

Challenge 1: Working with Memory. Apart from registers, the architectural state of most architectures consists of a large memory region. While the whole memory address space could, in principle, be modeled as a single register, the sheer size of addressable memory makes this infeasible. Further, instruction encodings usually only access a small portion of memory based on an address. Thus, values in main memory should be represented as two registers: one storing the address and one storing the value. The size of the register storing the address should match the architecture's pointer size, while the size of the register storing the value should be the size accessed by the encoding. An implementation of our approach needs to determine the addresses and sizes of memory accesses. Further, the runner needs to be able to construct memory mappings at arbitrary addresses with arbitrary values of any size and to record their values after execution.

Challenge 2: Collecting Noise-free Samples. A runner must be able to reliably execute instruction encodings under arbitrary initial architectural states and record the complete architectural state after execution. If a single register or memory mapping is not set up correctly or is modified by the framework code, the semantics of an encoding likely cannot be recovered correctly.

Challenge 3: Efficiently computing and checking the Set of Output Functions F. For our proposed algorithm to work, F needs to contain all actual output functions of an instruction encoding. However, a larger size of F increases the runtime as more possible output functions have to be checked. An implementation needs to find a good tradeoff between the size of F and performance. Further, it is infeasible to check a possible output function containing an immediate for each possible value of the immediate by brute force. An implementation needs further optimizations to check each possible output function efficiently.

Challenge 4: Dealing with Conditional Semantics. Conditional semantics can be represented as output functions combining simple mathematical operations. However, this entails encoding the condition and both outcomes into a single function, inevitably growing the complexity of the function. With more complex functions, the size of F immensely increases, making it infeasible to enumerate. Thus, correctly recovering conditional semantics might require additional optimizations.

Challenge 5: Providing a good Distribution. As described in Section 3.3, our approach requires a distribution of random initial architectural states that can efficiently distinguish possible output functions in F. An implementation must use domain knowledge of F to provide a good distribution.

Challenge 6: Dealing with SIMD and partial Registers. Same Instruction Multiple Data (SIMD) [16] instructions often split one big architectural register into multiple logical registers and operate on multiple logical registers simultaneously. Such a split is again possible but highly infeasible to model as combinations of simple mathematical functions. Similarly, some instructions only perform operations on parts of the register (e.g., the lower 32-bit of a 64-bit register). Thus, special care is needed to recover the semantics of SIMD instructions, and optimizations for instructions targeting only part of a register can be made.

Each of these challenges affects the design and optimization of our implementation, which we describe in Section 4.

4 Implementation: InstrSem

In this section, we describe InstrSem, our proof-of-concept implementation of our approach described in Section 3. InstrSem is modular, extensible, and designed to recover encoding semantics and generalize them into instruction semantics across multiple ISAs. InstrSem is implemented in C, Python, and minimal architecture-specific assembly and supports reverse engineering for 64-bit LoongArch, AArch64, and RISC-V. InstrSem works with instructions accessing memory and features a plethora of optimizations to make reversing feasible. InstrSem is split in two parts: The reverser (Section 4.1), which recovers encoding semantics, and the clusterer (Section 4.2), which generalizes them into instruction semantics.

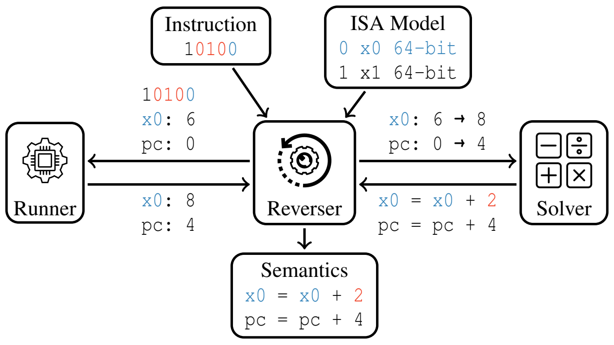

Figure 1: Overview of InstrSem's reversing: The reverser is instantiated with an architecture-specific ISA model describing register names, sizes, encodings, and constraints and a runner able to execute and collect arbitrary architectural states. Whenever the reverser is fed an instruction encoding, it detects changing outputs and their corresponding inputs. For each input-output pair, the reverser collects samples and feeds them to the solver. The solver responds with an output function correctly describing the mapping from inputs to outputs for all samples. When output functions for all outputs are found, they are returned as encoding semantic.

Section 4.3 describes how we tackle the challenges outlined in Section 3.5 and reduce the runtime of InstrSem.

4.1 Reverser

Figure 1 shows how the reverser works: it takes a raw instruction encoding, uses the architecture-specific runner to execute it under randomized architectural states, and synthesizes output functions that explain changes in the state. These output functions form the encoding's semantics. The reverser consists of multiple parts that all run in userspace:

Runner. The runner is responsible for executing instruction encodings under an arbitrary given architectural input state and recording the resulting architectural state, including register and memory content. It is written in architecture-specific assembly and minimal C and supports user-mode execution.

Solver. The solver is responsible for inferring output functions and constraints from input-output samples and creating random input samples under constraints. It contains code to generate the set of possible output functions F from a list of simple mathematical operations (Table 1) and an implementation to efficiently check F against a list of input-output samples. Further, the solver is responsible for efficiently creating random architectural input states likely to trigger corner cases of functions in F and satisfy a set of constraints.

Reverser Core. The reverser core contains the high-level code that uses the solver and runner to automatically reverse-engineer a given instruction encoding. It implements the algorithm described in Algorithm 2 alongside different opti-

Table 1: Overview of operations used by InstrSem to construct candidate output functions. If two expressions of different bitwidth are combined an additional sign- or zero-extension is added to the smaller expression.

Kind	Operations
Primitive	register, immediate
Arithmetic	addition, subtraction, multiplication, division, modulo
Logic	and, or, xor, unary not
Unary	extraction of least-significant 2^n bytes
Conditions	equals, not equals, less than, negation

mizations to overcome challenges outlined in Section 3.5 and speed up the reverse-engineering process.

ISA Model. For each architecture, the ISA must be defined and passed to the reverser. This definition contains register names, sizes, constraints, and generic information such as constraints on canonical addresses. Further, this model describes how immediates and registers can be encoded in an instruction bitstring. This information is required to create random architectural states that conform to the architecture's constraints, to interface with the architecture-dependent runner, and to generalize encodings to instruction semantics.

Workflow. The interactions between the components are visualized in Figure 1. The reverser is instantiated with an ISA model and an instruction encoding to reverse. It then uses the runner to collect architectural states for multiple partially random inputs the solver chooses. Then, the solver is queried to obtain output functions for registers and memory values O that the encoding modifies. If functions f_o for all changing outputs $o \in O$ can be recovered, the encoding semantics $S = \{(o, f_o) | o \in O\}$ are reported.

4.2 Clusterer

Once the reverser finds the formal semantics for an instruction encoding, the clusterer is invoked to generalize the single encoding to an instruction, implementing a variant of the algorithm described in Algorithm 4 in Appendix A. The clusterer is depicted in Figure 2. It detects possible bits that can encode registers used in the instruction encoding according to the ISA model. By mutating these bits to encode a different register and replacing the register in the encoding semantics, the clusterer obtains a new encoding and new predicted semantics. Then, the clusterer runs this modified encoding, gathers the resulting architectural state from the runner, and verifies that the semantics match the predicted semantics. If the predicted semantics are correct for 8 replacement registers, the clusterer assumes the bits actually encode the register and generalizes the instruction encoding. Immediates are detected and generalized similarly. However, the encoded bits do not need to match

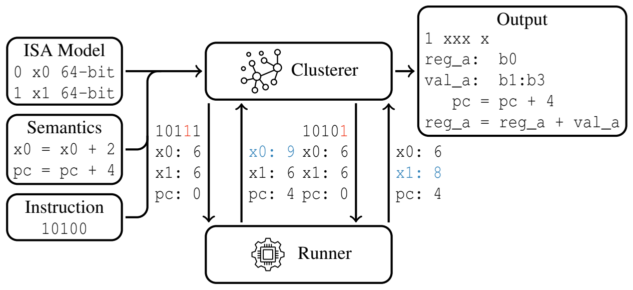

Figure 2: Overview of InstrSem's clustering: The clusterer is instantiated with an architecture-specific ISA model and runner. Given an instruction encoding and encoding semantics recovered by the reverser, the clusterer generalizes the encoding semantics to instruction semantics describing how registers (`reg_a`) and immediates (`val_a`) are encoded and the semantics of the instruction.

the immediate used in the semantics exactly. Instead, a possible left-shift, sign-extension, or off-by-one value is allowed by default. Additionally, architecture-dependent immediate encodings like immediates split into multiple bitstrings can be defined. By locating possible encodings of registers and immediates and verifying them, the clusterer can generalize the encoding semantics S into instruction semantics E that can be instantiated for multiple instruction encodings.

4.3 Challenges

InstrSem addresses the practical challenges (cf. Section 3.5).

Challenge 1: Working with Memory. Instructions may operate on memory, but modeling the entire address space is impractical. During analysis, InstrSem identifies memory accesses by capturing segmentation faults and treats memory locations as register-like pairs (address and value).

To identify the base address of memory mappings, a signal handler is installed that records the crashing address on a segmentation fault. This address can then be treated as an output variable and an output function for it can be inferred. To infer the function, random input states and the corresponding segmentation fault address are collected using the runner. Then, $f_{\text{crash\_address}}$ is learned by the solver from these samples, and a mapping can be created. We treat mappings as two registers: r_{addr} containing the address and r_{val} containing the value. r_{addr} is constrained such that $r_{\text{addr}} = f_{\text{crash\_addr}}(A)$ must be true for each initial architectural state A. r_{addr} must further contain an address that the userspace runner can map.

On some platforms, the signal handler is only provided with a segfault address if the address is canonical. Otherwise, '0' is provided to the signal handler. In such cases, the bit-width of values that may be placed in registers when randomly sampling them is constrained to fewer and fewer bits until enough samples with non-zero segfault addresses are collected. These constraints can be removed once $f_{\text{crash\_addr}}$ is known, since r_{addr} is itself constrained to contain a canonical address, which implicitly constrains the values of registers affecting $f_{\text{crash\_addr}}$.

The bit-width of r_{val} is initially set to a high value that exceeds all reasonable memory operations. Further, the mapping is initially mapped with *read*, *write*, and *execute* permissions. The actually required permissions are systematically inferred by changing the permissions and observing segmentation faults. If the permission contains write permissions, the write bit-width is determined by observing the maximum number of bits in r_{val} that change when executing the instruction encoding under random inputs. If the permission contains read permissions, the read bit-width is determined by sampling the same states but flipping bit i of r_{val} and observing whether the state after execution changes. The maximum i that induces changes is the read bit-width. If the permission contains execute permissions, the execute bit-width is set to a platform-dependent fixed value, and r_{val} is fixed to an architecture-specific undefined instruction encoding that leads to an illegal instruction exception. Finally, r_{val}'s bit-width is set to the maximum of the read, write, and execute bit-width.

With r_{addr} constrained to the mapping's address, r_{val} set to the correct bit-width, and the required mapping protection determined, the memory mapping can be added. For the remainder of the solving code, r_{addr} and r_{val} are treated like any other register. Only the runner is aware of the memory mapping. Before execution, the runner creates a mapping with r_{addr} as address and r_{val} as value and stores the value after execution in r_{val}. Similarly, an implicit mapping with the *pc* (program counter) register as address and the instruction encoding under test as value is always created.

Challenge 2: Collecting Noise-free Samples. Our approach relies on the execution of instruction encodings under arbitrary input states A, i.e., arbitrary memory mappings and register values, and the collection of the state A' after execution. Thus, program execution must not be influenced by other memory mappings, i.e., the full initial state can be prepared, and the full resulting state can be recovered.

To recover the state, we use Linux signal handling. On each tested platform, all register values (including vector registers) and required information about signals are passed to the signal handler. To reduce the influence of uncontrolled mappings, our runner is a statically linked binary with a single link-time-controlled base address for code, data, and stack. The code does not use the C standard library but instead directly invokes the syscalls for signal handling, memory mapping, and I/O operations. This removes the need for a heap and enables our code to use reserved registers. This further allows mapping the entire code and data segment read-only during instruction execution to protect the framework from modifications through executed encodings. Only the signal-handling stack needs to be mapped as writable, but information to this stack is only written after the tested encoding is executed and can thus not be modified by the encoding under test. Most of this code can be implemented as a platform-independent runner. Only syscall numbers, registers, a way to extract register values during signal handling, and a way to set register values from memory must be implemented per platform. Additionally, the registers, including constraints, bit-width, and possible encodings, must be modeled per platform using Python code.

Challenge 3: Efficiently Computing and Checking the Set of Output Functions. An easy way of of computing a set of possible output function is to define a set of simple expressions and mathematical operations and combining them up to a maximum depth for the resulting tree. However, suppose the simple expressions contain all possible immediate values up to a certain bit-width and all possible register and memory values. In that case, the search space quickly becomes infeasible to handle. Moreover, it fails to cover many common operations, such as branches with large immediate values as offset. InstrSem employs multiple optimizations to reduce the search space and traverse it efficiently.

Before solving for functions or constraints, InstrSem identifies all registers that influence the output being solved. By doing this, InstrSem can exclude functions that use other registers from the search. Further, InstrSem uses a symbolic constant value for an immediate. Then, when a function $f_?$ is checked whether it is an output function for o (i.e., $f_?(x) = R(x,i)[o]$ holds for any $x \in A$) for a set of input samples $a \in \{A_1, A_2, ..., A_n\}$, a configurable heuristic is used to determine how to perform the checking efficiently. If $f_?$ does not contain any symbolic immediates, $f_?$ can just be evaluated for all values of a to check if $f_?(a) = R(a,i)[o]$. If $f_?$ contains exactly one symbolic immediate, $f_?$ is instantiated with all immediates that can reasonably be encoded in the instruction encoding, leading to $f_?^c$ where c is such an immediate. Then, all $f_?^c$ are individually checked whether $f_?^c(a) = R(a,i)[o]$ for collected samples a. If more than one symbolic immediate is used in $f_?$, we use the *z3* constraint solver to determine whether a value for all immediates exists such that $f_?^{c_1,...,c_n}(a) = R(a,i)[o]$ for all a where $c_1, ..., c_n$ are the values of the immediates used in the function that must all come from the set of reasonably encodable immediates for the instruction encoding. If any of the above methods determine that $f_?$ (or $f_?^c$ or $f_?^{c_1,...,c_n}$ if symbolic immediates are used) correctly computes $R(a,i)[o]$, we use $f_?$ as the output function for o. Constraint solving can be deactivated to trade coverage of F for performance.

To enumerate $f_?$, InstrSem constructs binary trees of expressions with a configurable maximum depth. The primitive values that can be used are either a symbolic immediate or any of the registers determined as input, and the operations are any from the default set of arithmetic, logic, and unary operations (Table 1). These operations are provided as z3 BitVector expression and Python functions for optimized evaluation. Further, if two registers of different sizes are combined in an operation, the smaller register can be sign- or zero-extended to match the bigger bit-width. If the final output is smaller than the target register, the value can also be extended. If it is bigger, it is truncated to match the target register. Using the method described above, InstrSem can reasonably iterate and check functions with a syntax tree depth up to three. For an output with two input registers a and b as well as an output register x of the same size, InstrSem generates three expressions for depth 1: a, b, and immediate. For depth 2, the number of expressions grows to 90, and for depth 3 to 81 000.

In general, the number of expressions for depth d can be approximated as $expr(1) = \text{ins} + 1$ with "ins" denoting the number of input variables and $expr(n) = expr(n-1)^2 * k$ where k is the number of possible binary operations. Since this does not take unary negation and possible sign- or zero-extensions into account, this is a slight underapproximation. Still, $expr(4)$ is already 160 000 000 for $\text{ins} = 1$ and $k = 10$. Thus, InstrSem can only feasibly recover semantics for an output that can be represented using a function $f_?$ with a syntax tree depth below 4 directly.

Challenge 4: Dealing with Conditional Semantics. Some instructions, such as conditional branches, rely on conditional semantics. While they can be represented with the operations outlined in *Challenge 3*, the resulting required syntax trees have greater depth as they must encode the condition and both possible outcomes. Since syntax trees with depths greater than 3 are infeasible to enumerate, InstrSem includes an additional optimization to deal with conditional semantics.

InstrSem splits output samples into two classes if solving fails, using one of the following simple heuristics: (i) The output value matches a specific input value in at least 2 % but no more than 98 % of samples and (ii) there are exactly two different output values. The first heuristic detects conditional semantics in which one case directly uses an input value, such as conditional branches, in which the memory address of the mapping is used in case a branch is taken. The second heuristic targets conditional semantics where both cases produce a fixed output value. For instance, instructions setting an output register to *0* or *1* depending on a condition. When a split is performed, the solver finds a constraint on inputs that matches the split. Constraints are enumerated and checked similar to output functions. Constraints are built by combining up to two possible output functions with (in)equality operators (Conditions in Table 1). For both sides of the split, the constraint (or negated constraint) is added to the architectural input state generation, and the reverser is invoked recursively. If the configurable recursion depth (i.e., number of nested conditions) is exceeded, the reverser aborts.

If semantics for both cases are found, the constraint and the resulting semantics are recorded. Importantly, conditional splitting is only performed for outputs that cannot be solved otherwise. Thus, output functions for some outputs are only reversed once and not in each split. For example, the output of InstrSem for a conditional branch is

```
mem_addr = (pc + 18640)
if ($a0 < $a1) pc = mem_addr else pc = (pc + 4)
```

InstrSem determines that the output value of *pc* matches a specific input value, namely *mem_addr*, for greater than 2 % but no more than 98 % of cases. Thus, the inputs are split accordingly. Then, InstrSem determines that the condition *$a0 < $a1* describes this split. When recursively solving with this condition applied when sampling input architectural states, the output function for *pc* can now be recovered as *mem_addr*. When recursively solving with the negation of the condition (i.e., ¬*($a0 < $a1)*) applied when sampling input states, the output function for *pc* is recovered as *pc + 4*.

Challenge 5: Providing a Good Distribution. When creating random architectural states as input, the performance of the algorithm outlined in Algorithm 2 strongly depends on the provided distribution. This distribution must cover corner cases to distinguish potential output functions to provide correct results. While at first glance, most mathematical operations we provide are easily distinguishable when using random inputs (e.g., the chance that a division produces a different output than a multiplication is high), some corner cases might be hard to distinguish. For instance, the two operations $r_1 >> r_2$ (right shift of register r_1 by the unsigned value stored in register r_2) and $r_1 \oplus r_1$ (register r_1 XORed with itself) likely both yield '0' if sampling is performed uniformly at random over a 64-bit space even with 1 000 000 samples. This is the case since sampling $r_2 < 64$ is unlikely for a 64-bit uniform distribution. Similarly, the conditions $r_1 \leq r_2$ and $r_1 < r_2$ are only distinguishable if $r_1 = r_2$ which is unlikely with uniformly random sampling. Thus, for each architectural input state sample, InstrSem creates a list of (i) 3 uniformly random values to provide entropy to the samples, (ii) 6 values uniformly randomly chosen from immediates that may be encoded in the instruction encoding to trigger corner cases of instructions using immediates, and (iii) 0, 1, -1, and the maximum signed integer value. Then, each register and memory value is chosen uniformly at random from this list and adjusted to the correct bit-width. Since this initial selection of values may violate some constraints applied during architectural input state sampling, an additional post-processing step is used. This post-processing step uses z3 to re-assign some register and memory values to fulfill all required constraints while trying to maximize the number of registers and memory values that are kept random. The random architectural input state sampling can trigger corner cases while also allowing for enough entropy to create a large number of distinct samples.

Challenge 6: Dealing with SIMD and Partial Register Operands. Many ISAs support instructions working with smaller operands than the register size. The result is then sign- or zero-extended to the register size. If the initial solving fails, InstrSem detects the output size, and if it is smaller than the register size, it tries solving again while limiting the operand and output sizes. Similarly, SIMD instructions perform the same operation on multiple data operands, often treating a single register as multiple values. InstrSem tries to detect SIMD operations if initial solving fails. It splits registers into smaller pseudo-registers, trying to solve for a single operation that produces correct results on all pseudo-registers.

5 Evaluation

This section evaluates InstrSem on RV64I (RISC-V) and LA64 (LoongArch). We measure how accurately and efficiently InstrSem recovers encoding semantics and generalizes them into instruction semantics. Section 5.1 focuses primarily on documented instructions for which ground truth is available. This allows us to quantify the performance of InstrSem and verify the correctness of the results. Section 5.2 evaluates InstrSem on the whole LA64 encoding space, showing that it can characterize undocumented behavior.

5.1 Coverage and Correctness on RV64I

The RV64I base ISA contains 38 instructions that all 64-bit RISC-V CPUs must implement. We evaluate InstrSem on a single instance of each encoding using QEMU version 9.2. The reverser is provided with an ISA model describing register sizes, how registers and immediates may be encoded in an instruction bitstring, and constraints.

Methodology. We execute InstrSem on a single instance of each documented instruction in the RV64 integer instruction set. For each instruction encoding, the reverser attempts to synthesize semantics from randomized input-output samples. If successful, the clusterer generalizes to instruction semantics. We verify correctness by comparing recovered semantics against the official RISC-V specification. If the instruction encoding is correctly reversed and clustered, we mark the instruction as correctly reversed. Finally, we report the percentage of the instruction space that is correctly reverse-engineered. Instructions that contain semantics not modeled by InstrSem, such as memory fences, and instructions where semantics for not all outputs can be recovered automatically, are counted as incorrectly reversed.

We first execute InstrSem with *samples* set to 500 and *depth* set to 2. *Samples* describes the amount of random architectural input and corresponding output states InstrSem should collect whenever recovering an output function or constraint. *depth* limits the depth of the syntax trees of possible output functions and constraints. For instructions that are not recovered, we rerun InstrSem again in a first step with *samples* increased to 25 000, and in a second step, the *depth* is increased to 3. This setup also demonstrates that InstrSem can be used incrementally, quickly recovering simple instructions while spending more resources on complex instructions.

Results. On an AMD Ryzen 7 5700, the RISC-V runner of InstrSem can execute 8791 instruction encodings per second. When randomizing each input state without additional constraints, the runner achieves 6579 encodings per second. With the initial settings, InstrSem correctly recovers and clusters 28 of the 38 instructions that make up 176 390 144 of 193 265 667 (91.27 %) of instruction encodings. InstrSem takes, on average, 261.06 s to fully reverse and cluster an instruction. Of this time, 257.97 s are spent collecting samples. Thus, InstrSem is entirely bottlenecked by sample collection. The execution speed of the runner does not cause the sampling bottleneck, but generating random architectural states that fulfill all required constraints.

Two instructions, *slti* (set less than immediate) and *sltiu* (set less than immediate unsigned), rely on sampling input states containing the encoded immediate in a specific register to successfully reverse and cluster the semantics correctly. In *Challenge 5*, we describe how such immediates are generated. However, since InstrSem reports 9187 possible distinct immediates for the provided *slt* instruction encoding, 500 samples do not suffice to generate the correct immediate or an off-by-one value which would also work.

When increasing the number of samples from 500 to 25 000, the runtime of *slti* and *sltiu* increases to roughly 4270.03 s, but InstrSem can successfully reverse and generalize them, leading to an additional 8 388 608 individual instruction encodings with known semantics.

For instructions that could not be recovered, we run InstrSem with *depth* set to 3. With this change, InstrSem can additionally recover the semantics of 3 instructions that make up 65 536 encodings: *sll* (shift left logical), *srl* (shift right logical), and *jalr* (jump and link register). The shift instructions only use the least significant 6-bit of the register operand as shift value, while the shift primitive in InstrSem uses the whole register. Thus, the additional depth is required to limit the register bit-width with an additional modulo or logical and operation. For *jalr*, the addresses reported to the signal handler are aligned to 2 B. This address is not relative to the pc register, which is aligned to 2 B by default, but an arbitrary register that can contain unaligned values. Thus, the additional depth is required for an additional operation aligning the address. As the alignment operation recovered by InstrSem only works for even immediate values, the instruction recovered by InstrSem is missing one variable immediate bit. If we execute InstrSem again on *jalr* with an odd immediate, InstrSem recovers instruction semantics for the remaining *jalr* instruction encodings with odd immediates. Together, both encodings cover the full *jalr* instruction semantics.

Four instructions (*ebreak*, *ecall*, *fence*, and *pause*) are recovered insufficiently as their semantics are not part of the model. While InstrSem recovers most of them to the capabilities of its model (e.g., fences are recovered equivalently to nops, since they only affect the modelled program counter), we mark them as incorrectly recovered since the recovered semantics do not cover the full instruction semantics.

Appendix B summarizes the instructions that InstrSem can recover without manual effort, the parameters used to recover them, and the total runtime for successful recoveries. InstrSem uncovers 33 of 38 instructions defining 189 038 592 of 193 265 667 (97.81 %) of all valid instruction encodings.

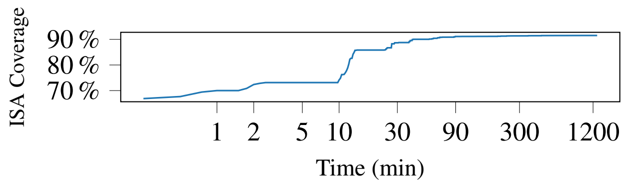

Figure 3: Coverage of the 32-bit LA64 encoding space over time with clustering instructions enabled. Time is measured after filtering undefined instruction encodings.

5.2 Full LA64 Instruction Set

To evaluate InstrSem on a more complex ISA, we analyze the entire 32-bit instruction encoding space of LoongArch LA64. We use a hybrid QEMU and Loongson 3A5000 setup to compare emulator and hardware behavior.

Setup. We first determine the set of used instruction encodings with an approach similar to Armshaker [32]: We execute every possible 4-byte sequence as an instruction encoding and observe whether an invalid instruction signal for this encoding is received. For QEMU, we end up with 1 422 536 156 valid encodings, while the Loongson 3A5000 supports 1 464 902 445 encodings, 42 366 289 more than the emulator. We run InstrSem on these valid encodings, updating the input set every time instruction semantics are recovered. For QEMU, we run the evaluation on an AMD Ryzen 9 7950X3D with 64 parallel processes. On the Loongson 3A5000, we run 8 parallel processes. The *depth* parameter of InstrSem is set to 2 and *samples* is set to 500.

Correctness. InstrSem successfully recovers semantics for 873 instructions. After filtering out redundant instructions, i.e., instructions fully contained within another and those with 5 or fewer variable bits, 136 instructions remain. We filter out instructions with 5 or fewer variable bits, as they are primarily special cases of instructions that could not be uncovered. The remaining instructions cover 1 009 055 744 of 1 009 080 152 analyzed instruction encodings. Manually analyzing these instructions reveals that all semantics of documented instructions are recovered correctly with respect to the ISA model. Additionally, semantics for undocumented instructions that we cannot verify against the documentation are unveiled. They are further discussed in Section 6.2.

Runtime. Figure 3 shows the percentage of instruction encodings InstrSem covers over time when clustering instructions is enabled. InstrSem starts at a coverage of 66 % as we pre-filter unused instruction encodings. Initially, InstrSem has a higher chance of reverse-engineering an instruction comprising many encodings. Thus, InstrSem covers additional 23.54 % of the encoding space over the first hour, while only 1.1 % are covered in the following 9 h.

Impact of Clustering. Without clustering, the portion of covered instruction encodings grows linearly at an average rate of 6.55 encodings per minute. Covering the same 23.54 % of the encoding space that can be covered in one hour with clustering enabled would take approximately 29 367 years. This underlines the performance gain from clustering instructions.

6 Case Studies

In this section, we present case studies that demonstrate InstrSem's ability to recover and generalize semantics for undocumented instructions on RV64I (RISC-V), AArch64 (Arm), and LA64 (LoongArch) CPUs. It further demonstrates the modularity of InstrSem and its applicability to CISC and unconventional ISAs. Section 6.1 shows that InstrSem can recover the semantics of the unprivileged GhostWrite instruction. Section 6.2 describes our findings for the LA64 instruction set and the corresponding QEMU emulator. Section 6.3 outlines an undocumented instruction encoding on Apple M-Series CPUs found by InstrSem. Section 6.4 uses InstrSem to reverse-engineer multiple undocumented instructions on the SiFive P550 RISC-V CPU. Section 6.5 demonstrates that InstrSem can handle CISC ISAs while Section 6.6 shows it can handle unconventional ISAs. These examples highlight its real-world applicability across different ISAs, vendor-specific extensions, ISA inconsistencies, and emulator bugs.

6.1 Reversing GhostWrite

To demonstrate InstrSem's adaptability to real-world CPU vulnerabilities, we use it to reverse engineer the semantics of the GhostWrite CPU vulnerability on the T-Head XuanTie C910 RISC-V CPU. This vulnerability allows an attacker to write arbitrary bytes to arbitrary physical memory locations from unprivileged code. At its core, GhostWrite is an invalid encoding of strided vector-store instructions, which the C910 illegally decodes and executes.

Setup. As InstrSem is designed to only capture virtual memory changes, GhostWrite is invisible to InstrSem. Therefore, we extend InstrSem's RISC-V runner with physical memory support. To keep the changes to the runner small, we map a small reserved physical memory range into the virtual address space of the runner, by mapping `/dev/mem` via `mmap`. As GhostWrite is able to write arbitrary physical memory, running GhostWrite on uncontrolled physical memory might crash the system, e.g., by overwriting kernel code. For this reason we also constrain the values encoded into the registers to this fixed and reserved physical memory region using InstrSem's built-in constraint functionality. In total, the modifications amount to less than 50 lines of code.

Results. With these minimal modifications InstrSem successfully reverse-engineers the correct semantics of GhostWrite, generalizing from a single encoding into an instruction covering 65 536 encodings (Listing 1). The analysis automatically infers semantics, ignored encoding bits, and encoding bits for registers, substantially reducing manual effort. This demonstrates that InstrSem can effectively analyze existing security issues and can be extended with minimal effort for future vulnerability analysis.

6.2 LA64 Instruction Set

On the LA64, InstrSem discovers undocumented instructions, inconsistencies between hardware CPUs and the QEMU emulator, and bugs in QEMU.

Undocumented Vector Instructions. Amongst the instructions recovered by InstrSem are 67 vector instructions. While the *existence* of vector instructions is documented, the instructions are undocumented. The manual is empty and contains only a note "TBD" [26]. The recovered vector instructions comprise 128-bit and 256-bit SIMD operations on 8, 16, 32, 64, and 128-bit values. A list of recovered vector instructions is provided in Appendix C.

Inconsistent Instruction Semantics. For some instructions, different semantics are recovered from QEMU and the Loongson 3A5000 CPU. Most 128-bit vector operations zero the upper 128 bits of the vector register on QEMU, while the operation is applied to the full 256-bit register on hardware.

QEMU-only Instructions and Crashes. InstrSem recovers 761 856 instruction encodings only supported by QEMU and not by the actual hardware. These encodings set a varying number of bits of a vector register to 1. Further, InstrSem finds 49 152 encodings that crash QEMU. For instance, the encoding *0x73e3a000* triggers an assertion regardless of the architectural input state ("`[...] vldi_get_value: code should not be reached`"). All of these crashes are controlled termination due to assertion errors and are not exploitable beyond denial of service. Still, this discovery shows that there is a blind spot in current research that ISA-agnostic tools like InstrSem can help to close.

Hardware-only Instructions. InstrSem discovers an undocumented instruction only on the Loongson 3A5000 and not in QEMU. This instruction stores the most significant byte of a general-purpose register to memory. The relevant output for this instruction is listed in Listing 2 in Appendix D.

6.3 Apple Undocumented Instructions

In this case study, we investigate the proprietary Mul53 extension on Apple M-series CPUs, which is known [27] but not officially documented. On the Apple M1 and M2, this extension includes 2 instructions, each containing 1024 instruction encodings. The instructions together add a 106-bit wide SIMD multiply operation to the Apple CPUs. The first instruction, which is dubbed `mul53lo.2d`, splits 2 128-bit vector registers into parts of 64 bit and computes the product of the 2 registers for each half. The result is then truncated to 53 bit. The second instruction, which is dubbed `mul53hi.2d`, multiplies the halves as well, but right-shifts the output by 53 bit, discarding the least significant 53 bit. Again, the output is truncated to 53 bit. With these two results, the final 2 106-bit outputs can be constructed by shifting the second result (high) left by 53 bit and adding the first result (low).

We first use InstrSem to automatically reverse engineer `mul53lo.2d`. InstrSem successfully discovers that the instruction is a SIMD split instruction and fully recovers the instruction semantics in 15 min. The relevant output of InstrSem is in Listing 3 in Appendix D. For `mul53hi.2d`, we set the width of the vector registers to 53 bit, as the previous run discovers the output truncation to 53 bit. InstrSem successfully recovers the instruction semantics for the lower half of the vector registers in 30 min. The relevant output is also contained in Listing 3 in Appendix D. These partial semantics can easily be expanded to the full-width SIMD split semantics. We conclude that InstrSem successfully recovers the semantics of a proprietary ARM64 ISA extension, highlighting its applicability to more complex and custom extensions.

6.4 SiFive P550 Undocumented Instructions

In this case study, we investigate undocumented instructions discovered on the SiFive P550, one of the fastest RISC-V CPUs currently available. Previous work reported over 40 million undocumented instruction encodings on this CPU [33] without recovering their semantics. We use InstrSem to reverse-engineer their semantics.

Methodology. We randomly pick one of the undocumented instruction encodings and use InstrSem to try to recover the semantics. If InstrSem recovers the semantics, we use its clustering mechanics to recover the more general instruction semantics. We then remove all encodings that match this recovered instruction from the list of encodings to still reverse. We run the experiment for 96 h on a single CPU core.

Setup. We use a HiFive Premier P550 running SiFive FreedomUSDK to run InstrSem's client remotely. We use TCP to connect it to a machine with an AMD Ryzen 9 7940HS running Ubuntu 22.04, which runs InstrSem. The remote setup is required as Z3 is unavailable on the RISC-V machine.

Results. InstrSem recovers the semantics for 305 instructions in 96 h, covering 20 381 696 instruction encodings. A large number of the instructions are signed maximum and minimum instructions. They include register maximum and minimum semantics, and ones where an immediate is used instead of a register. The relevant output of InstrSem when recovering such instructions is in Listing 4 in Appendix D. InstrSem further verifies the suspicion of previous work [33] that some of the encodings encode immediate shift-right instructions.

The instructions partially come from RISC-V's basic bit manipulation extension (`Zbb`). This extension includes signed maximum and minimum operations, though the encoding differs. While the official `Zbb` extension defines the signed maximum with opcode `0x33`, we find instructions with opcode `0xb`. Further, this extension does not include signed maximum or minimum operations with an immediate.

We conclude that InstrSem can also recover the semantics of more complex instructions like minimum or maximum. Further, InstrSem's clustering stage efficiently recovers semantics at scale, a tedious task to do manually.

6.5 Partial x86-64

In the previous case studies, we focused on fixed-length RISC instruction sets. However, InstrSem is not limited to fixed-length or RISC instructions. To demonstrate this, we implement a partial x86-64 backend including only 64-bit integer registers (`rax`, `rbx`, `rcx`, `rdx`, `rdi`, `rsi`, `rbp`, `rsp`, and `r8-r15`) and the program counter (`rip`). This backend amounts to less than 200 lines of code, more than 100 of which are used for a C library defining types, syscalls, and signal handling structs.
Setup. We disassemble the GNU libc standard library and supply instructions with the 20 most common mnemonics.
Results. Out of the 20 mnemonics, InstrSem correctly finds the semantics and generalizes them for 13 of them. The semantics of 5 more instructions rely on flags which we do not model in the minimal backend. Still, they are recovered correctly to the capabilities of the model (i.e., `jne` always jumps while `je` is a nop). InstrSem only fails for two instructions: `ret`, and `call`. Both cases fail because they perform two memory accesses (one to store or retrieve the return address from stack and one fetch at the branch destination). This is currently unsupported by InstrSem but can be extended since InstrSem correctly identifies size and address of the first memory access. For x86, InstrSem falls short compared to tools like LibLisa [7] that are specifically built to infer x86 semantics. Still, InstrSem can handle a decent amount of x86-64 instructions even with a minimal backend.

6.6 Logitech Macros

To demonstrate InstrSem can also handle unconventional ISAs, we implemented a backend for Logitech's Macro language [34]. This macro language implements a variable length instruction set running on keyboards and mouses. For ease of implementation and faster execution, we wrap an emulator of this macro language instead of actual hardware. The wrapping for the emulator consists of only 75 lines of Python code.

Setup We feed one instruction bitstring per opcode supported by the emulator. When InstrSem finds semantics and generalized semantics, we manually check the results.

Results InstrSem can correctly reverse engineer 13 of 15 Logitech macro instructions with respect to its model. One of these instructions is a delay, recovered as nop, as the delay is not part of the ISA model. For two instructions, the generalization step fails, as the immediate is interpreted as an exponent, i.e., $2^{immediate}$, which InstrSem's default constant iterator does not support. We confirm these instructions are correctly generalized with a custom constant iterator implementing such encodings. For the two opcodes we count as incorrect, the semantics are correct for the supplied instruction-bitstring, but do not reflect the whole opcode semantics. Although the Logitech macro language differs from traditional ISAs, InstrSem manages to reverse-engineer a significant portion of its instructions out-of-the-box with a minimal backend.

7 Related Work

Finding Undocumented Instructions. Strupe et al. [32] propose Armshaker, a tool to find undocumented instruction encodings in the ARMv8 architecture. It scans the entire encoding space, logging encodings that fail to disassemble but do not cause a SIGILL on execution. Although this tool did not uncover undocumented instructions, it discovered software bugs in Linux and QEMU. Dofferhoff et al. [10] use a similar technique on ARMv8 and RISC-V, successfully detecting an undocumented instruction on the Freedom U540 RISC-V CPU. Sandsifter [11] is a popular fuzzing tool for discovering undocumented instruction encodings on x86. Instead of exhaustively searching the encoding space, it uses a custom search algorithm that considers the length of a generated encodings. Sandsifter discovered undocumented instructions on CPUs by both Intel and AMD, as well as a series of errata. Later works improve upon Sandsifter's approach, further reducing the runtime and size of the relevant search space [23,39]. These works are orthogonal, as we do not focus on finding instruction encodings but inferring their semantics.
Automatically Unveiling Instruction Semantics. Godefroid and Taly [18] propose a template-based program-synthesis technique to generate semantics for x86 instructions. Their approach successfully synthesizes the semantics of 534 instruction variants. Heule et al. [21] propose Strata, which improves upon this approach by employing *stratification*. This allows it to synthesize the semantics of more complicated instructions from simpler ones, eliminating the effort of handcrafting function templates. Strata successfully recovers the formal semantics of 1905 x86-64 instruction variants, covering 466 distinct mnemonics. Dasgupta et al. [8] manually extend these results to cover the complete x86-64 user-mode ISA. Recently, Craaijo et al. published libLISA [7], which does not require a disassembler and can find the semantics of x86-64 instructions fully autonomously. While the general approach is portable to other architectures, these works rely on architecture-specific hand-written templates or complex execution environments. Thus, they are not generic or easily portable to other architectures.

8 Limitations

8.1 Design

ISA State modelling. Our approach is fundamentally limited by the supplied ISA model: Instructions that rely on inputs which are not part of the ISA model cannot be fully reversed.

For instance, instructions that only execute in a privileged context cannot be correctly reversed if the privilege level is not part of the ISA model. Similarly, instructions that write to unknown outputs such as unmodeled registers are only partially recovered. Adjusting the ISA model to accommodate such cases is theoretically possible, but practically infeasible for most real-world ISAs. For instance, parts of the architectural state can often not be arbitrarily adjusted and not all parts of the state can be read directly. Additionally, parts such as some model-specific registers might not be documented, making them impossible to model.

Incompleteness. Our approach requires the initial set of possible output functions F to contain the actual output function for each output. If this is not the case, incorrect semantics can be recovered. For our relaxed algorithm, an incorrect output function f might be recovered even though a correct function is part of F, as this approach does not cover the whole input space of each output function. This is computationally infeasible and thus a limitation of all black-box approaches.

8.2 Implementation

Complex Semantics. InstrSem is constrained by the expressiveness of its function synthesis. If an instruction's behavior cannot be described by the current operation set (e.g., population count, floating-point arithmetic), it cannot be inferred. Adding floating-point support would require modeling floating-point registers, rounding modes, and IEEE semantics as bit vector transformations—a nontrivial but feasible extension. Additionally, InstrSem is currently limited to instructions that perform at most one memory access.

Handling of Non-determinism. In its default configuration, non-deterministic outputs cause InstrSem to fail when recovering semantics for that output and consequently fail generalizing an instruction encoding. Handling of non-deterministic outputs can be enabled which classifies non-deterministic outputs as counter (non-decreasing values), "few values" (only a few possible values), or randomness (N-bit random output, with N estimated from the collected samples). While this allows InstrSem to proceed with generalization of instruction semantics, the recovered function might lose information about the actual output function. For instance, while the output of a `rdcycle` instruction is identified as counter, the connection to the current cycle count is lost.

Instructions Without Visible Side-effects. The semantics of instructions that have microarchitectural side-effects are not completely recovered. For instance, prefetch instructions that only cache data are recovered as nops, even if the architectural state is fully modeled. While the ISA model can be augmented with microarchitectural information (e.g., the cache state of a memory region), these properties must be added manually.

Privileged Execution Modes. InstrSem's runner is an userspace executable that relies on Linux syscalls and signal handling. Thus, it can only collect samples in userspace. Instructions that only execute in privileged modes cannot be recovered with the current runner implementation. Dealing with such instructions would require developing a runner that can execute privileged instructions under arbitrary initial states and collect the outputs after execution. Since some privileged instructions cause side effects that are hard to account for (e.g., resetting the CPU), this is a non-trivial endeavour.

System Registers. The current runner implementation does not support adjusting or capturing system registers that might have an influence on instruction execution or might be modified by instructions. For instance, instructions that are disabled unless a bit in a system register is set cannot be recovered by the current implementation. While it is possible to add certain system registers to the architectural state and use a kernel module to write and read them, this approach would require knowing which system registers are safe to arbitrarily adjust.

9 Discussion

Extensibility of InstrSem. InstrSem is a modular framework that can be extended to other architectures. To implement a new architecture, it only requires a model of the architectural state and a binary capable of executing instruction encodings with arbitrary input state and collecting the output state. For the latter, a platform-independent C implementation is provided that just needs small code snippets describing platform-specific syscall numbers, syscall wrappers, a way to load all registers from memory, and a way to extract all register contents during signal handling. Memory mappings and runtime isolation are written in a platform-independent way using Linux syscalls. The base set of mathematical formulas used in InstrSem can easily be extended to enable reversing of more complex instructions, e.g., instructions that count the number of set bits. This is valuable when encountering an instruction encoding that InstrSem cannot automatically reverse.

Scalability to Complex ISAs. InstrSem scales with the analysis goal. Modeling a large portion of an ISA, including privilege modes and system registers, is complex but feasible if required. For instance, the privilege level can be represented as an additional state variable, and the runner can be extended to execute encodings and capture the full architectural state for different privilege levels depending on the initial value of this variable. In many scenarios, a targeted analysis is sufficient and significantly simpler. When focusing on specific instruction classes, only the components of the ISA that influence their behavior must be modeled. Finally, InstrSem accepts arbitrary-length byte streams as input for the tested encoding. As a consequence, variable-length instruction encodings and even short instruction streams can be analyzed without additional machinery, enabling support for both RISC- and CISC-style ISAs out of the box.

Complex Instruction Semantics. Even complex instructions often decompose into simpler per-output functions, which InstrSem can infer efficiently. If not all output functions of a complex instruction can be fully recovered, InstrSem still provides valuable information such as relevant parts of the input state that affect the output. Further, InstrSem can aid manual analysis by collecting input and output samples under arbitrary constraints. Once a missing output function is manually recovered and added to InstrSem, InstrSem can automatically generalize the encoding into an instruction semantic covering many encodings, further reducing manual effort.

Using InstrSem for Emulation. The semantics recovered by InstrSem can emulate instructions out-of-the-box since semantics map an arbitrary input state to the correct output state. By manually implementing the semantics of complex instructions such as syscall instructions, InstrSem can build an emulator for architectures without having to manually implement semantics for a large portion of the instruction set.

Splitting Sample Collection and Solving. InstrSem is split between sample collection (runner), semantic inference (reverser), and generalization (clusterer). This modularity supports reuse in different contexts, e.g., running the reverser locally and collecting samples on a remote target.

10 Conclusion

We presented InstrSem, a fully automated framework for discovering and generalizing instruction semantics across architectures, including undocumented and proprietary instructions. By combining black-box execution, semantic inference, and encoding generalization, InstrSem bridges the gap between binary-level execution and formal instruction models. Our evaluation on RISC-V, ARM, and LA64 shows that InstrSem recovers correct semantics for most documented instructions and reveals undocumented behavior in real-world CPUs and emulators. InstrSem provides a scalable and modular foundation for reverse engineering, emulation, and CPU security analysis in an era of increasingly opaque instruction sets.

Ethical Considerations

Affected Stakeholders

Our research affects CPU designers, designers of other software and hardware that rely on ISAs, researchers, and maintainers and users of qemu-loongarch.

How Stakeholders are impacted

CPU designers. CPU designers are affected because our research simplifies the analysis of ISAs implemented on their hardware, including but not limited to undocumented instructions. On the downside, our research can simplify the analysis of proprietary instructions, potentially aiding intellectual property theft. On the upside, our research can enable additional testing to ensure implementations behave as expected.

Designers of other software and hardware that rely on ISAs. As not only CPUs rely on ISAs or similar components that can be modeled and analyzed with InstrSem, our research potentially also affects designers of other hardware, such as GPUs or keyboards that rely on macros, as well as software such as VM-based obfuscation techniques. These stakeholders are affected in a similar manner to CPU designers.

Researchers. Researchers are affected by our work as it simplifies and partially automates analysis of ISAs or specific instructions. Our research likely reduces the manual effort required.

Maintainers and users of qemu-loongarch. Since our research discovers several bugs in qemu-loongarch, maintainers and users are affected. On the negative side, maintainers have to spend time addressing and fixing the identified bugs. Further, malicious parties could exploit bugs discovered in our research to perform denial-of-service attacks against affected qemu-loongarch versions, potentially affecting users of qemu-loongarch. On the positive side, outlining these implementation bugs lead to fixes that make qemu-loongarch more secure and correct.

Mitigations taken for negative impacts to stakeholders. We reported the QEMU crashes to the qemu-loongarch maintainers alongside detailed root-cause analysis and patches. This ensured the issues were quickly fixed and the effort of qemu-loongarch maintainers was minimized.

How the decision to both do and publish the research were reached

We weighed the possible negative impact of aiding intellectual property theft against the positive impact our research might have to simplify future security research. We believe the impact to intellectual property theft of our research to be low: While InstrSem can simplify manual reverse-engineering, a sufficient monetary incentive would also overcome the additional manual effort required. On the other hand, finding possible security issues with our research, responsibly disclosing them, and aiding future research would lead to greater security of CPUs. Thus, we decided to conduct our research.

While publishing our research and open-sourcing InstrSem might aid malicious actors, we believe the positive impact it will have by aiding research on CPU security outweighs the negatives. Withholding knowledge about architecturally-reachable behavior disproportionately advantages attackers while limiting defenders, tool builders, and auditors. Our work does not introduce new instructions or new behavior. It characterizes behavior that already exists on existing CPUs. The presence of undocumented but executable instructions justifies systematic analysis, as these instructions form part of the effective attack surface. Thus, we think publishing our research is ethical.

Open Science

We include our prototype implementation of InstrSem as artifact. The artifact can be viewed at the following URL: `https://zenodo.org/records/17974657`. After download, InstrSem can be executed the following way:

```
  python3 main.py <backend> <instruction>
<port> <instruction bitwidth>.
```

- `<backend>` can be one of: `logitech`, `x86_64-socket`, `riscv64-socket`, `aarch64-socket`, `aarch64-vector-socket`, and `loongarch64-socket`.
- `<instruction>` is target instruction as integer. This number is converted to bytes using little-endian byteorder. For instance, `0x1122` yields the bytes `[0x22, 0x11]`.
- `<port>` should be 0.
- `<instruction bitwidth>` is the bitwidth of the reversed instructions. For instance, the riscv `addi x1, x0, 42` instruction (`0x02a00093`) has a bitwidth of 32.

The artifact further contains a `README.md` file describing how to reproduce the claims of the paper.

References

[1] Amine Benamrane, Imade Benelallam, and El Houssine Bouyakhf. Constraint programming based techniques for medical resources optimization: medical internships planning. 2020.

[2] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*, 2022.

[3] Felix Brandt, Reinhard Bauer, Markus Völker, and Andreas Cardeneo. A constraint programming-based approach to a large-scale energy management problem with varied constraints: A solution approach to the roadef/euro challenge 2010. *Journal of Scheduling*, 2013.

[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*. ACM, 2019.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *S&P*, 2012.

[6] Yun Chen, Lingfeng Pei, and Trevor E Carlson. AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *ASPLOS*, 2023.

[7] Jos Craaijo, Freek Verbeek, and Binoy Ravindran. libLISA: Instruction Discovery and Analysis on x86-64. In *OOPSLA*, 2024.

[8] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *PLDI*, 2019.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[10] Rens Dofferhoff, Michael Göebel, Kristian Rietveld, and Erik Van Der Kouwe. iscanu: A portable scanner for undocumented instructions on risc processors. In *DSN*, 2020.

[11] Christopher Domas. Breaking the x86 isa. *Black Hat*, 2017.

[12] Christopher Domas. Hardware Backdoors in x86 CPUs. *Black Hat US*, 2018.

[13] Josh Eads, Tavis Ormandy, Matteo Rizzo, Kristoffer Janke, and Eduardo Vela Nava. Zen and the art of microcode hacking, 2025.

[14] Mark Ermolov and Maxim Goryachy. Intel VISA: Through the Rabbit Hole. *Black Hat Asia*, 2019.

[15] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[16] Michael J Flynn. Some computer organizations and their effectiveness. 1972.

[17] Antonio Garrido, Marlene Arangu, and Eva Onaindia. A constraint programming formulation for planning: from plan scheduling to plan generation. In *Journal of Scheduling*, 2009.

[18] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. *ACM SIGPLAN Notices*, 2012.

[19] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. Decoding cuda binary. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2019.

[20] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *PLDI*, 2016.

[22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[23] Xixing Li, Zehui Wu, Qiang Wei, and Haolan Wu. Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching. *IEEE Access*, 2019.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[26] Loongson. Loongarch reference manual - volume 2: Vector extensions. https://github.com/loongson/LoongArch-Documentation/releases/download/2023.04.20/LoongArch-Vol2-v1.00-EN.pdf, 2023.

[27] Hoang Trung Nguyen. Apple H10 Mul53 extension, 2022.

[28] Tavis Ormandy. Reptar, 2023.

[29] Tavis Ormandy. Zenbleed, 2023.

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[31] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *USENIX Security*, 2024.

[32] Fredrik Strupe and Rakesh Kumar. Uncovering hidden instructions in Armv8-A implementations. In *HASP*, 2020.

[33] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. In *CCS*, 2025.

[34] Leon Trampert, Lorenz Hetterich, Lukas Gerlach, Mona Schappert, Christian Rossow, and Michael Schwarz. Peripheral Instinct: How External Devices Breach Browser Sandboxes. In *WWW*, 2025.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[36] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[37] Menkes Hector Louis van den Briel, Thomas Vossen, and Subbarao Kambhampati. Loosely coupled formulations for automated planning: An integer programming perspective. 2008.

[38] Fish Wang and Yan Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, 2017.

[39] Jiatong Wu, Baojiang Cui, Chen Chen, and Xiang Long. A high efficiency and accuracy method for x86 undocumented instruction detection and classification. In *IMIS*, 2021.

[40] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

A Algorithms

Algorithm 5 and Algorithm 6 show helper algorithms for encapsulation checks used in Algorithm 3 and Algorithm 4.

Algorithm 5: CheckEncapsulation

```
Data: Set of encapsulations $K$, Mapping from
      instruction encodings to encoding semantics $M$
Result: True or False
$Result \leftarrow True$;
for $i', s_{expected} \in All\_Instructions(K)$ do
    if $s_{expected} \neq M(i')$ then
        $Result \leftarrow False$;
    end
end
```

B RISC-V Results

For RISC-V, Table 3 shows the instructions that InstrSem can recover without any manual effort, the parameters used to recover them, and the total runtime for successful recoveries.

C LA64 Undocumented Vector Instructions

Supplying InstrSem with undocumented instruction encodings yields the undocumented vector instructions listed in Table 2 when running with qemu-user (Loongarch).

Algorithm 6: CheckEncapsulationRelaxed

```
Data: Runner $R$, distribution of architectural states $D$,
      set of encapsulations $K$, amount of tested
      encodings $n_i$, amount of architectural states $n_a$
Result: True or False
$Result \leftarrow True$ ;
for $i', s_{expected} \leftarrow N\_Encodings(K)$ do
    for $a \in N\_States(D, n_a)$ do
        if $s_{expected} \neq R(a, i')$ then
            $Result \leftarrow False$ ;
        end
    end
end
```

Table 2: LA64 vector instructions automatically unveiled by InstrSem. Operand sizes in brackets only apply for 256-bit operations. `reg` and `imm` refer to `register` and `immediate`.

Operation	Output Size	Operand Sizes
add reg	128/256 bit	(128), 64, 32, 16, 8
subtract reg	128/256 bit	(128), 64, 32, 16, 8
multiply reg	128/256 bit	64, 32, 16, 8
modulo reg	256 bit	64, 32, 16, 8
logical or reg	128/256 bit	indistinguishable
logical and reg	128/256 bit	indistinguishable
logical xor reg	128/256 bit	indistinguishable
add imm	128/256 bit	64, 32, 16, 8
subtract imm	128 bit	32, 16, 8
left shift imm	128/256 bit	(128), 64, 32, 16
logical or imm	128/256 bit	indistinguishable
logical and imm	128/256 bit	indistinguishable
logical xor imm	128/256 bit	indistinguishable
load imm	128/256 bit	32, 16, 8

D Sample Outputs

Reversing GhostWrite. InstrSem's relevant output for the GhostWrite strided vector-store instruction on the T-Head XuanTie C910 is shown in Listing 1. This instruction stores the least significant byte of a vector register to physical memory.
Loongson 3A5000-only Instruction. InstrSem's relevant output for an undocumented instruction on the Loongson 3A5000 is shown in Listing 2. This instruction stores the most significant byte of a general-purpose register to memory.
Apple MUL53 Extension. InstrSem's relevant output for the proprietary Apple Mul53 extension is shown in Listing 3. For *mul53hi.2d*, we fixed the vector register size to 53 bits.
P550 Sample Reversed Instruction. InstrSem's relevant output for an undocumented signed maximum instruction on a SiFive P550 CPU is shown in Listing 4. InstrSem successfully performs a conditional split and obtains the correct semantics.

Table 3: Overview of RV64I instruction encodings and whether InstrSem can recover and generalize their semantics. One instruction encoding for each instruction of RV64I is used except for *jalr*, which requires two encodings to uncover the full semantics. `D` is the maximum syntax tree depth.

Encoding		Time	Bits	D	Samples
add s7, ra, t2	✓	7.15 s	15	2	500
addi s10, ra, 0x2a	✓	12.98 s	22	2	500
and ra, tp, s1	✓	8.24 s	15	2	500
andi gp, ra, 0x2a	✓	13.09 s	22	2	500
auipc tp, 0x2a	✓	12.51 s	25	2	500
beq s9, ra, 0x2a	✓	211.74 s	22	2	500
bge ra, sp, 0x2a	✓	215.56 s	22	2	500
bgeu sp, tp, 0x2a	✓	245.40 s	22	2	500
blt ra, sp, 0x2a	✓	213.59 s	22	2	500
bltu gp, sp, 0x2a	✓	252.76 s	22	2	500
bne tp, ra, 0x2a	✓	218.24 s	22	2	500
ebreak	✗		0	2	500
ecall	✗		0	2	500
fence	✗		22	2	500
jal tp, 0x2a000	✓	184.79 s	25	2	500
jalr sp, ra, 0x29	✓	713.91 s	22	3	500
jalr sp, ra, 0x2a	✓	712.54 s	22	3	500
lb ra, 0x2a(sp)	✓	870.79 s	22	2	500
lbu ra, 0x2a(tp)	✓	682.14 s	22	2	500
lh sp, 0x2a(gp)	✓	928.47 s	22	2	500
lhu ra, 0x2a(gp)	✓	584.24 s	22	2	500
lui gp, 0x2a	✓	12.65 s	25	2	500
lw tp, 0x2a(ra)	✓	796.39 s	22	2	500
or tp, gp, s7	✓	8.82 s	15	2	500
ori tp, a0, 0x2a	✓	14.64 s	22	2	500
pause	✗		0	2	500
sb sp, 0x2a(ra)	✓	490.31 s	22	2	500
sh gp, 0x2a(ra)	✓	462.71 s	22	2	500
sll ra, sp, s3	✓	143.61 s	15	3	500
slt tp, a7, a3	✓	323.88 s	15	2	500
slti sp, gp, 0x2a	✓	4283.28 s	22	2	25 000
sltiu ra, gp, 0x2a	✓	4256.78 s	22	2	25 000
sltu gp, ra, sp	✓	106.69 s	15	2	500
sra tp, sp, a6	✗		15	3	500
srl gp, ra, tp	✓	140.72 s	15	3	500
sub sp, s8, ra	✓	6.83 s	15	2	500
sw sp, 0x2a(ra)	✓	403.87 s	22	2	500
xor ra, gp, tp	✓	8.01 s	15	2	500
xori sp, ra, 0x2a	✓	13.14 s	22	2	500

```
flippy bits: [20, 21, 22, 23, 24, 25]
       31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
        0   0   0   1   0   0   0   0   0   0   0   0                       0   0   0                       0   1   0   0   1   1   1
reg_a                                                                                   4   3   2   1   0
reg_b                                                   4   3   2   1   0
mem_addr = gpr_b
pc = (pc + 0x4)
mem_val_out = Truncate(128 to 8, vec_a)
```

Listing 1: Sample output of InstrSem when reverse-engineering the GhostWrite strided vector-store instruction on the T-Head XuanTie C910. The instruction stores the least significant byte of a vector register to physical memory. `flippy bits` indicates ignored bits in the instruction encoding.

```
       31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
        0   0   1   0   1   1   1   1   1   0
val_a                                          11  10   9   8   7   6   5   4   3   2   1   0
reg_a                                                                                           4   3   2   1   0
reg_b                                                                                                               4   3   2   1   0
mem_addr = (gpr_a + const_a)
pc = (pc + 0x4)
mem_val_out = (gpr_b >> 0x38)
```

Listing 2: Sample output of InstrSem when reverse-engineering an undocumented instruction on the Loongson 3A5000. The instruction stores the most significant byte of a general-purpose register to memory.

```
mul53lo.2d:
       31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
        0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
reg_a                                                                                           4   3   2   1   0
reg_b                                                                                                               4   3   2   1   0
pc = (pc + 0x4)
vec_b = SIMD(128 -> 128, 64, ((vec_b * vec_a) & 0x1fffffffffffff))

mul53hi.2d:
       31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
        0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   1
reg_a                                                                                           4   3   2   1   0
reg_b                                                                                                               4   3   2   1   0
pc = (pc + 0x4)
vec_b = ((vec_b * vec_a) >> 53)
```

Listing 3: Relevant output of InstrSem when reverse-engineering the proprietary Apple Mul53 extension. For `mul53hi.2d`, the vector register bit-width was fixed to 53 bit.

```
       31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
        0   1   0   1   1   1   0                                           0   0   1                       0   0   0   1   0   1   1
reg_a                               4   3   2   1   0
reg_b                                                                                   4   3   2   1   0
reg_c                                                   4   3   2   1   0
pc = (pc + 0x4)
if (not (gpr_a <s gpr_c)):
  gpr_b = gpr_a
else:
  gpr_b = gpr_c
```

Listing 4: Sample output of InstrSem when reversing an undocumented signed maximum instruction on the SiFive P550.