VeriISLE: Verifying Instruction Selection in Cranelift

Monica Pardeshi

July 25, 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Fraser Brown, advisor, chair
Bryan Parno

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Authored and distributed by: Monica Pardeshi
E-mail: mpardesh@andrew.cmu.edu

Keywords: Lightweight verification, SMT solvers, Cranelift, instruction selection, WASM

Abstract

Language-level guarantees—like runtime isolation for WebAssembly (Wasm) modules—are only as strong as the compiler that produces a final, native-machine-specific executable. The process of lowering language-level constructions to ISA-specific instructions can introduce subtle bugs that violate security guarantees. In this paper, we present VeriISLE, a system for lightweight, modular verification of instruction-lowering rules within Cranelift, a production retargetable Wasm code generator. We use VeriISLE to verify lowering rules that cover WebAssembly 1.0 support for integer operations in the ARM `aarch64` backend. We show that VeriISLE can reproduce 3 known bugs (including a 9.9/10 severity CVE), identify 2 previously-unknown bugs and an underspecified compiler invariant, and help analyze the root causes of a new bug.

Acknowledgements

Contents

1 Introduction 1

2 Background 3

3 VeriISLE Design 7

4 Evaluation 14

5 Related Work 21

6 Future Work 23

7 Conclusion 24

Bibliography 25

1 Introduction

WebAssembly [36] (Wasm) is a portable bytecode format originally designed for the browser, with three main goals: safety, speed, and portability. Wasm's machine-independent but low-level semantics make compilation and execution fast on any platform; its type system and bounded memory regions work together to prevent programs from reading or writing data outside of their own heap (their *sandbox*). This isolation guarantee is essential when users interact with the web, because each click leads to untrusted code.

Isolation has made Wasm popular beyond the web, too. *Edge cloud* services from Cloudflare [43], Vercel [71], and Fastly [61], for example, run users' Wasm code on geographically distributed content delivery networks. To improve startup time, these Wasm-based services can co-locate different untrusted code modules *within the same process*; Wasm's lightweight isolation enforcement takes the place of more traditional, costly process- or VM-based isolation.

Unlike a process or VM, however, Wasm's safety guarantee relies on the correctness of the underlying compiler. The compiler inserts dynamic checks that confine a module to its own memory region *before* generating native code for that module. Code generation, then, is a pillar of every Wasm-backed system's trusted compute base: almost *any* miscompilation, however seemingly benign or rare, could be exploited to produce code that bypasses Wasm's security guarantees [31, 24, 22, 23]. Code generation bugs can let malicious Wasm code steal data from—or corrupt the execution of—completely unrelated modules or the host runtime itself.

As one example, a code generation CVE[1] in Cranelift [17], a compiler backend used in several industrial Wasm runtimes, permitted this kind of sandbox escape [25]. The bug was in Cranelift's `x86-64` instruction selection, which uses addressing modes to implement complex address computations with a single instruction. `x86-64` addressing modes can apply small left shifts, so a single `movl` instruction is enough to implement code like the following Wasm snippet:

```
1 (i32.load (i32.shl (local.get x) (i32.const 3)))
```

To lower this code to `x86-64`, Cranelift must convert 32-bit Wasm addresses into offests from an instance's *base address* in the target machine's 64-bit address space. This conversion requires zero-extending the 32-bit Wasm address, computing the 64-bit address as `base+zext(addr)` (where `addr` is the original 32-bit Wasm address, `base` is the base address for the module's memory region, and `zext` is a zero-extension). Unfortunately, the Cranelift

[1] "Common Vulnerabilities and Exposures", a designated list of publicly disclosed security bugs.

instruction selector lowered the above Wasm code to `x86-64` instructions that computed `base+zext(x)<<3` *instead of* `base+zext(x<<3)`. This mistake lets attackers break out of the Wasm sandbox by giving them access to an extra *3 significant bits of native address space*. In Wasmtime [18], a popular Wasm engine that uses Cranelift, this allows a guest Wasm instance to silently read and write memory 6 to 34 GB away from its own sandbox. Clearly, *even simple bugs in instruction selection can create security vulnerabilities*.

Instruction selection is hard to get right because it bridges the (large) semantic gap between the compiler's intermediate representation (IR) and the processor's instruction set architecture (ISA). While some instruction-lowering rules are simple—essentially one-to-one translations from an IR construct to an equivalent ISA instruction—others are not. They perform complex transformations to eke out instruction-level performance improvements; account for operators that exist in *either* the IR or the ISA—not both; and select different ISA instructions based on details of IR operations (e.g., their bit-widths).

To help compiler developers *automatically* reason about the correctness of their instruction-lowering rules, we present VeriISLE. VeriISLE verifies rules written in Cranelift's ISLE domain-specific language (DSL) for specifying how IR terms translate to machine code sequences. To use VeriISLE, developers annotate their ISLE lowering rules with specifications; VeriISLE uses a Satisfiability Modulo Theories (SMT) solver [11] to automatically verify full functional equivalence—i.e., that a rule translates an IR instruction to a native code sequence with equivalent semantics. VeriISLE allows developers to *gradually* annotate new rules, and to quickly update annotations as rules evolve. This modularity is essential because Cranelift is an evolving production compiler: lowering rules—and entire backends!—are subject to change. To our knowledge, our work with VeriISLE is the first formal verification effort for the instruction-lowering phase of an efficiency-focused production compiler.

In sum, in this paper, we:

1. Create VeriISLE, a framework for verifying instruction-lowering rules in ISLE.
2. Verify Cranelift's implementation of all integer operations in the latest major WebAssembly release—1.0 [66]—for the ARM `aarch64` Instruction Set Architecture (ISA).
3. Use VeriISLE to reproduce and detect previously-fixed bugs (§4.0.3) and vulnerabilities (§4.0.3), including the example bug from this section.
4. Use VeriISLE to help Cranelift developers identify (§4.0.4, §4.0.4) and fix (§4.0.4) new bugs and under-specified compiler invariants (§4.0.4).

We begin by introducing background on instruction lowering and the ISLE DSL (§2.0.1). Then, we present VeriISLE's design (§**??**), and evaluate its results on Cranelift (§**??**), a production Wasm compiler backend. Finally, we discuss plans to build on VeriISLE towards fully-verified Wasm compilers (§**??**).

2 Background

This section provides background for understanding VeriISLE verification (§??) by describing the instruction lowering problem (§2.0.1) and Cranelift's ISLE domain-specific language (DSL) for writing lowering rules (§2.0.3). Finally, it introduces SMT solvers [11], the tools that power the VeriISLE verification engine (§2.0.4).

2.0.1 Instruction Lowering

During *instruction lowering*, an *instruction selector* translates the compiler's *intermediate representation* (IR) to machine instructions. The instruction selector's job is to search for a combination of machine instructions that (1) matches the IR's semantics and (2) performs well. A single-pass selector that emits a fixed set of instructions for every IR operator fulfills the first goal but not the second: it allows translations of one IR instruction to N machine instructions, but not more efficient N-to-M translations. This design, for example, precludes compiling a program with addition and multiplication operations to machine code that uses a fast multiply-add (`madd`) instruction.

Most modern instruction selectors *do* support more general N-to-M matching; in fact, a good instruction selector often embodies a good *pattern matcher*. It detects arrangements of multiple operators in the IR that can be translated, together, into machine instructions. In full generality, this is an NP-hard combinatorial search problem; as a result, most production compilers use heuristic shortcuts for practicality (e.g., greedy pattern matching, as in the "maximal munch" scheme [20]).

More complex ISAs and ISA extensions yield more complex matching strategies. For an extreme example, bit-permutation and swizzling instructions vary widely across ISAs, and lowering of a general permutation operator sometimes requires a "solver"—or at least a bevy of heuristic special cases to produce good code [65, 55, 70]. This is part of what makes instruction selection (and instruction selection verification!) interesting: it is *not* simply the task of mapping mostly-equivalent operators, like translating IR addition to the machine's integer addition instruction. The most subtle reasoning—and many bugs—occur when there is a large semantic gap between the IR and ISA, and when producing efficient machine code is a first-order priority [75, 53].

Production compilers today use a mix of hand-written and DSL-based descriptions of their instruction lowering rules: e.g., LLVM [46] has a 46K-line C++ file specifying `x86-64` lowerings, while the Go compiler uses a term-rewriting DSL where developers can specify expression-tree patterns [35]. In this paper, we focus on the Cranelift compiler's lowering

DSL.

2.0.2 The ISLE lowering DSL

The Cranelift compiler project [17] introduced the ISLE (***I**nstruction **S**election **L**owering **E**xpressions*) DSL [32, 3, 33] in 2021 in order to replace handwritten instruction-lowering code with declarative patterns. ISLE is broadly a term-rewriting system [29, 72]. In the next sections, we give a brief overview, and then walk through an example of instruction lowering in ISLE.

ISLE's term rewriting for lowering

The main body of a program in ISLE consists of a series of rules. These rules are written in S-expression syntax and consist of a *left-hand side (LHS)* and *right-hand side (RHS)*. The LHS is a pattern, and can use pattern-matching operators such as wildcards, variable captures, or destructuring (matching a term and then feeding its arguments to sub-patterns). The RHS is an expression consisting of a tree of terms, possibly using variables captured from the LHS. A rule indicates that the RHS expression is produced whenever the instruction selector encounters a term tree matching the LHS.

To express instruction lowering as term rewriting, ISLE introduces a top-level term `lower` that takes an expression tree as its argument. For example, to lower an integer add operator (`iadd`) to the `add` instruction in the ISA (e.g., `x86-64` or `aarch64`), one would write:[1]

```
(rule (lower (iadd ty x y)) (isa_add ty x y))
```

where `iadd` is defined in Cranelift IR and `isa_add` is defined amongst all available machine instructions in the ISA.

ISLE has a strict, static type system that operates on types defined in ISLE (some of which are external, Cranelift-defined types, such as Rust enums for instructions' opcodes). Nested terms on both the left- and right-hand sides must type check (i.e., with return and argument values aligned). In addition, the left- and right-hand side of a rule must have the same type.

Because of the type system's restrictions, Cranelift expresses all lowerings as rewrites from `(lower (IR_operator ...))` to term trees representing machine code expressions, potentially passing through multiple intermediate terms. The term `lower` is necessary because the LHS and RHS of a rule must have the same type—but top-level LHS patterns return IR `Insts`, while top-level RHS expressions return machine `Reg`isters. `lower`, with type signature `(decl lower (Inst)Reg)`,[2] does the `Inst` to `Reg`ister conversion that allows lowerings rules to type check by giving the LHS and RHS the same type.

Finally, ISLE's type system supports *automatic type conversions*. In the `iadd` example, such conversions apply to `x` and `y`, which are variables of type `Value` bound by the left-hand side of the rule. The RHS, in contrast, operates on `x` and `y` `Reg`isters. To reconcile these incompatible types, the ISLE compiler automatically inserts type conversions if a conversion

[1] Slightly simplified for clarity; real rules differentiate on the values' types.

[2] We elide an indirection via another type for clarity.

rule has already been specified for a pair of types. In this case, ISLE wraps the latter uses of `x` and `y` with the user-defined term `put_in_reg`, which converts `Value`s to `Reg`s.[3]

2.0.3 ISLE by example: lowering rotations

In this section, we walk through Cranelift's lowerings for a few specific instructions; this sets us up to *verify* such lowerings in the next section (§**??**).

Consider the Wasm `rotl` and `rotr` ("rotate") binary numeric instructions, which shift the bits of a value left or right with wraparound. Cranelift has corresponding `rotl` and `rotr` IR operations. The ARM `aarch64` ISA has a single implementation of rotate—`ROR`—which has a corresponding ISLE term named `a64_rotr` that includes an additional parameter to specify the 64-bit or 32-bit variants of the instruction.

A simple attempt at lowering `rotr` instructions to the ARM aarch64 backend might look like this:

```
(rule (lower (rotr x y)) (a64_rotr I64 x y))
```

This rule lowers to the 64-bit variant (`I64`) of `a64_rotr`. It works properly for 32- and 64-bit values, but *not* for narrower values (e.g., 8-bit values). This is because Cranelift operates on narrow values of w bits by placing them in 64-bit registers *but considering only their lowest w bits to be meaningful*. To see how the above rule is broken for 8-bit values, imagine it matching in a situation where `x` is `#b00000001`. Placing this value in a 64-bit register and attempting to right-shift it by one moves the right-most 1 bit to the highest bit *of 64—not* the expected result of 64 bits with `#b10000000` as the lowest eight!

Cranelift must instead special-case on narrow values:

```
(rule
  (lower (has_type (fits_in_16 ty) (rotr x y)))
  (small_rotr ty (zext32 x) y))
```

This rule uses external *helper terms* `has_type` and `fits_in_16` to predicate this rule only on narrow types; if `ty` is larger than 16-bits, the rule will not match. The helper terms are defined externally from ISLE, in Rust code that returns the value's type (`has_type`) and checks the type against the integer sixteen (`fits_in_16`), respectively. This rule also abstracts over types (lowering the burden on the compiler engineer): the rule binds a new variable, `ty`, to the type of the return value of `rotr`, and passes `ty` through as an argument to the right-hand side.

The rotate rule also uses an *intermediate term*, `small_rotr`. `small_rotr` only ever exists in ISLE—not in the resulting machine code—and is an intermediate step along the path to a final machine code representation. Intermediate terms like `small_rotr` let developers share logic across many different rules. As one example, Cranelift's `rotl` (rotate *left*) rule for narrow inputs also uses `small_rotr`. The compiler uses a `small_rotr` with a negated rotate amount because ARM does not have a distinct rotate left instruction:

```
(rule
  (lower (has_type (fits_in_16 ty) (rotl x y)))
  (let ((neg_amt Reg (a64_sub I32 (zero) y)))
```

[3] We describe the semantics of `put_in_reg` in §3.0.1.

```
4     (small_rotr ty (zext32 x) neg_amt)))
```

This rule is the same as the previous one with two additions. First, it uses a `let` clause to include another ISA instruction: an ARM `a64_sub` subtraction instruction, negating the value y by computing $0 - y$. Second, the rule wraps x on the right-hand side with a call to `zext32`, which zero-extends (that is, left-pads with zeros) the value of x up to 32 bits. Finally, to lower `small_rotr` to ISA-level operations, the Cranelift ISLE rules specify that narrow rotates can be composed of `aarch64`-native left shift and right shift instructions (not pictured). Thus, these ISLE rules lower a single IR instruction to multiple machine code instructions (`a64_sub` followed by shift and bitwise `or` instructions).

2.0.4 Satisfiability Modulo Theories (SMT)

To verify lowering rules written in ISLE, VeriISLE uses an SMT solver [28]. SMT solvers are tools that determine whether logical formulas are *satisfiable* for some assignment of values to variables.

Unlike SAT formulas [56], SMT formulas allow users to express higher-level statements (e.g., "`x < y[2]`") using a rich set of operators and types (e.g., integers and arrays) that are defined in the SMT-LIB standard [11]. VeriISLE lowers ISLE rules to SMT formulas in the theory of bitvectors and integers; we discuss this further in the next section.

3 VeriISLE Design

VeriISLE is a framework for verifying rewrite rules in the ISLE domain-specific language for instruction selection. VeriISLE uses an SMT solver [28] to show functional equivalence of the left- and right-hand sides of individual rules.[1] An equivalent left and right side mean that the rule has preserved IR semantics at the machine-code level; a differing left and right side indicate a bug in the lowering.

To verify their lowering rules, compiler developers write annotations on ISLE terms in VeriISLE's *annotation language* (§3.0.1). This language makes it simple to express term semantics (e.g., that `fits_in_16` means that a type can losslessly be represented with 16 bits). VeriISLE consumes ISLE's program representation for rules, combines this with the compiled annotations to create its own intermediate representation, and performs *type inference* (§3.0.1). Type inference is necessary for VeriISLE to lower its IR to an SMT formula, a logical formula that asks whether a rule's right and left-hand sides are equivalent. Finally, VeriISLE feeds the resulting formula into the SMT solver. If the right and left-hand sides of a rule differ, the solver returns a counter-example showing a set of inputs that cause the divergence; otherwise, the rule is verified.

In this section, we walk through the verification pipeline, from VeriISLE's annotation language (§3.0.1) to how it constructs and customizes verification conditions (§3.0.2).

3.0.1 The annotation language

It is impossible to verify functional correctness without precise semantics on terms within ISLE. While there are formal semantics for certain ISAs (e.g., ARM [4] and Intel [27]), there are no semantics for Cranelift's intermediate representation—or for ISLE helper terms (e.g., `has_type`) and intermediate terms (e.g., `small_rotr`). The *challenge* in specifying these semantics is that production compilers are living software: engineers change rules, add rules, and occasionally add entire new back-ends. To support modular verification of an evolving codebase, VeriISLE introduces an annotation language that allows rule authors to define specifications *as they go*, introducing a term's semantics inline, next to the term itself.

For example, consider our VeriISLE annotation on the helper term `fits_in_16`:[2]

```
1 (spec (sig (args arg) (ret))
2  (provide (= ret arg))
```

[1] Though VeriISLE supports more general custom verification conditions, as we will describe later in this section.

[2] ISLE terms and specification syntax lightly edited for clarity and brevity.

⟨*annot*⟩ ::= (‘`spec`’ ⟨*sig*⟩ (‘`provide`’⟨*ex*⟩$^+$) (‘`require`’⟨*ex*⟩$^+$))

⟨*sig*⟩ ::= (‘`sig`’ (args ⟨*bound*⟩$^+$) (⟨*bound*⟩))

⟨*bound*⟩ ::= (⟨*ident*⟩ ‘`:`’ ⟨*type*⟩)

⟨*type*⟩ ::= ‘`bv`’ | ‘`bv`’ ⟨*int*⟩ | ‘`Int`’ | ‘`Bool`’

⟨*width*⟩ ::= ⟨*int*⟩ | ⟨*ex*⟩

⟨*const*⟩ ::= ‘`true`’ | ⟨*int*⟩ | ...

⟨*ex*⟩ ::= ⟨*ident*⟩ | ⟨*const*⟩ | ⟨*encoding*⟩ ⟨*ex*⟩$^+$
| (⟨*unop*⟩ ⟨*ex*⟩) | (⟨*binop*⟩ ⟨*ex*⟩ ⟨*ex*⟩)
| (⟨*conv*⟩ ⟨*width*⟩ ⟨*ex*⟩) | (‘`extract`’ ⟨*int*⟩ ⟨*int*⟩ ⟨*ex*⟩)
| (‘`int2bv`’ ⟨*width*⟩ ⟨*ex*⟩) | (‘`bv2int`’ ⟨*ex*⟩)
| (‘`widthof`’ ⟨*ex*⟩) | (‘`concat`’ ⟨*ex*⟩$^+$)
| (‘`if`’ ⟨*ex*⟩ ⟨*ex*⟩ ⟨*ex*⟩) | (‘`switch`’ ⟨*ex*⟩ (⟨*ex*⟩ ⟨*ex*⟩)$^+$)

⟨*unop*⟩ ::= ‘`!`’ | ‘`~`’ | ‘`-`’ | ...

⟨*binop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`=`’ | ‘`<=`’ | ...

⟨*conv*⟩ ::= ‘`sign_ext`’| ‘`zero_ext`’ | ‘`convto`’

⟨*encoding*⟩ ::= ‘`cls`’ | ‘`clz`’ | ‘`rev`’ | ‘`subs`’ | ‘`popcnt`’

Figure 3.1: VeriISLE's annotation language, which combines SMT-LIB constructs with conveniences (e.g., `switch`) and VeriISLE-specific constructs (e.g., `convto` and `widthof`).

```
3  (require (<= arg (16:Int))))
4 (decl fits_in_16 (Type) Type)
```

This specification says that `fits_in_16` is a partial identity function on the argument type `Type`—that is, for the arguments on which `fits_in_16` is defined, it returns the argument itself. The function is specified by the `provide` clause `(= ret arg)`, which sets the return value equal to the first argument; both variables are bound in the `spec` signature. `require` clauses specify a preconditions on the term. This precondition specifies that the rule is a partial function predicated on `(<= arg (16:Int))`—the fact that the argument, which VeriISLE maps to the SMT-LIB theory of integers, is less than or equal to 16. In ISLE, partial functions are used to determine whether a rule matches: if any term on the left-hand side is undefined, the rule does not match. In sum, these three lines of specification are enough to describe the semantics of `fits_in_16`: it is a partial identity function that returns the type argument `arg`, which matches if `arg` is under sixteen bits.

The annotation language grammar and semantics

Figure 3.1 shows the VeriISLE annotation language grammar. Most operations in the annotation grammar map directly to SMT-LIB constructions. For example, `+` applied to a bitvector maps to SMT-LIB's `bvadd` bitvector addition function.

VeriISLE adds conveniences like `switch` and a variadic `concat` operation, both of which desugar to folding SMT-LIB's fixed-argument `ite` (if-then-else) and `concat` (bitvector con-

catenation) operators over any number of arguments. `switch` also adds a verification condition that enforces that its branches are exhaustive, which has helped surface faulty annotations.

VeriISLE provides constructs for introspecting on and modifying bitvector widths. `widthof` returns the width—often only known directly at solving time (§3.0.2)—of a given bitvector value. `convto` changes the width of its bitvector argument with the following semantics: if the destination width is more narrow, `convto` extracts the relevant bits; if the destination width is wider, `convto` leaves the upper bits unspecified by concatenating a fresh SMT variable with unrestricted bits.

VeriISLE also provides higher-level versions of SMT-LIB constructs. For example, SMT-LIB rotates must have statically-provided widths; VeriISLE instead offers symbolic rotates, which it implements with shift and bitvector logic instructions. Finally, VeriISLE includes keywords that map to custom encodings in its backend: (1) `cls` and `clz`, which count the number of leading sign and zero bits, respectively (§4.0.3), (2) `rev`, which reverses the order of bits, (3) `subs`, which performs subtraction-with-flags, and (4) `popcnt`, which counts the number of 1 bits.

`provide` blocks specify the semantics of a term, typically by relating the returned value bound in the specification to one or more of the arguments. `require` blocks specify preconditions, which are assumed when a term is used on the left-hand side of a rule but checked—that is, verified to hold—when a term is used on the right-hand side of a rule. This is analogous to more traditional Hoare-style verification [38, 9], where function preconditions may be assumed within the body of a function but must be checked at function call site.

For example, `small_rotr` `require`s that the amount being rotated has been zero-extended from the narrow starting width to the full 64 bits of the register. This can be specified as:

```
(require (switch (ty)
  ((8:Int) (= (extract 63 8 x) (0:bv)))
  ((16:Int) (= (extract 63 16 x) (0:bv)))))
```

This `require` clause say that the type `ty` is 8 or 16, and that the relevant bits beyond index `ty` have been zero-extended. This must be *proven* true for a term that uses `small_rotr` on the right-hand side, but is assumed true for terms that rewrite from a `small_rotr` on the left-hand side.

The annotation language type system

Types in VeriISLE are integers, booleans, and bitvectors. The VeriISLE annotation language must support polymorphism over bitvector widths, since most of Cranelift's ISLE rules operation on its `Value` type, which is polymorphic over integer values in the Cranelift intermediate representation. (§2.0.3).

For example, during preprocessing, ISLE automatically inserts `put_in_reg` to implicitly convert Cranelift IR `Value`s to machine code `Reg`s—and because `Value`s vary in width, VeriISLE's annotation language must provide a polymorphic type signature to `put_in_reg`. In other words, `put_in_reg` must reconcile the potentially narrow `Value` with the 64-bit `Reg`. VeriISLE's `put_in_reg` annotation uses `convto` to reinterpret the polymorphic bitwidth of the argument as 64 bits:

```
(spec (sig (args arg) (ret))
```

```
  (provide (= (convto (64:Int) arg) ret)))
(decl put_in_reg (Value) Reg)
```

Type inference

The annotation language supports polymorphism over bitvector types, but its target representation does not: all bitvector operations in SMT-LIB operate on fixed-width bitvectors [60]. Therefore, VeriISLE must transform its high-level intermediate representation, which allows polymorphic bitvector types, into several SMT formulas, each over a different set of bitvector widths. VeriISLE uses two passes of *type inference* to determine those widths. The first inference pass produces an assignment of SMT types (e.g., bitvector) for each variable in a term or its specification. The second pass resolves the bitvector widths.

First pass First, VeriISLE runs a variant of classic unification-based type inference [54] in order to rule out type errors between annotations. This first pass yields an SMT type (kind)—either an integer, boolean, or bitvector—for each variable in both the specification and the term it describes. The first pass, however, *does not* necessarily resolve the width of each bitvector.

VeriISLE is not always able to resolve types via the first unification pass because types in ISLE are polymorphic at the time ISLE generates Rust for code generation (e.g., the type `Value` does not have a specific width when ISLE is being processed). For example, the width of the value of `small_rotr` depends on the *value* of an argument passed in, `ty`. Thus, VeriISLE finishes resolving bitwidths in a second typing pass.

Second pass During the second type inference pass, VeriISLE uses an SMT solver to resolve unknown bitvector widths. This pass takes terms and their specifications as input, along with the types that the first inference pass resolved. It models bitvectors as an over-approximation of their width (i.e., with bitwidth 64) and uses integer SMT variables to model the widths of each subexpression.

For each rule, we provide a set of possible type instantiations for the root left-hand side term (that is, a set of possible types for the argument and return values, based on Cranelift semantics). For example, for a simple Cranelift IR type such as `iadd`, the set of type instatiations is $(t, t) \to t$ for t in $\{i8, i16, i32, i64\}$ (e.g., $(i8, i8) \to i8$).

For a more complicated term that involves modifying the Cranelift IR width of the input and output, we consider a wider set of instantiations. For example, for extending values, we consider multiple output types per argument type:

$s \to d$
for s in $\{i8, i16, i32, i64\}$
  for d in $\{i8, i16, i32, i64\}$ if $d \geq s$

Most terms on the right-hand side of Cranelift's ISLE rules operate on types modeling registers, instead of values in the intermediate representation. Cranelift's invariant for narrow types placed in registers is that low bits are defined and high bits are undefined, so we encode registers as 64-bit bitvectors with potentially-unspecified high bits.

For most rules, this second pass produces a single possible type assignment. For some rules, there are multiple valid type assignments—in this case, we continue the verification process until the SMT solver says there are no more unique possible type assignments (similar to counter-example guided inductive synthesis [1]).

3.0.2 Generating verification conditions

Once VeriISLE has run type inference—yielding a low-level, typed intermediate representation—it can lower that representation to an SMT formula(s) that expresses equivalence of the right and left-hand sides of a lowering rule. When VeriISLE invokes the solver on the formula, there are three possible outcomes:

1. `Success`: the rule is verified.
2. `Failure with counterexample`: the rule is broken, and the solver provides a set of inputs that exposes the bug, formatted in ISLE surface syntax.
3. `Rule inapplicable`: for the given type instantiation, the rule does not match. This indicates that the rule contains predicates on the left-hand side—or guarded `if`/`if-let` clauses (see §4.0.4)—such that the rule never matches on this type instantiation.

To produce these 3 outcomes, VeriISLE uses (at least) two additional SMT queries. The first query determines if the rule is applicable by querying the solver to see if there exists a model in which all the necessary preconditions hold; if not, VeriISLE produces a `Rule inapplicable` result. The second query determines whether the lowering rule preserves equivalence; if so, `Success`, and if not, `Failure with counterexample`.

For each query, VeriISLE's formula for a given rule combines the semantics and preconditions of Cranelift IR terms, ISA terms, and external and intermediate terms—all provided by annotations—with the semantics of the ISLE language itself (e.g., `if-let` and other language constructs). VeriISLE combines semantics across term annotations via a recursive descent over the rule's RHS and LHS, equating corresponding arguments and return values.

The first query: applicability

Let $i_0 \dots i_{n-1}$ be input variables in the LHS of a rule, A^{LHS} be the set of SMT variables generated by the recursive descent on the LHS (and analogously RHS), P^{LHS} and R^{LHS} be the set of `provide` and `require` predicates in all annotations on the LHS (and analogously RHS). A rule is applicable if there are some inputs such that the LHS and RHS are both defined:

$$\exists\{i_0, \dots, i_{n-1}\} \quad \cup \quad A^{LHS} \quad \cup \quad A^{RHS} | P^{LHS} \quad \wedge \quad R^{LHS} \quad \wedge \quad P^{RHS} \quad \lfloor 3.1$$

Recall that this query does not ask about equivalence; it asks whether the rule applies at all, to at least one input. Including the RHS SMT variables (A^{RHS}) and `provide` expressions (P^{RHS}) in this initial query helps catch overly restrictive annotations. For instance, a vacuously false assertion in a `provide` annotation on the RHS should make the rule fail the applicability check (otherwise, the next step would be unable to find any counterexamples—because in

first order logic, false implies anything). Including P^{RHS} in the query makes such a rule fail at the applicability check.

The optional model distinctness check The applicability check succeeds as long as at least one assignment of input terms is applicable—*even if* there is just one set of applicable inputs. VeriISLE implements an optional check that looks for distinct input sets (i.e., checks that multiple SMT models are feasible in which every bitvector input term is distinct). VeriISLE creates a formula that asserts that each bitvector input differs from the one in the original model; if the query is unsatisfiable, there is only one set of matching inputs. This check identified a previously unknown bug where an ISLE rule never fired in practice (§4.0.4).

The second query: equivalence

If the first query succeeds, VeriISLE constructs another SMT query to determine equivalence. Let ret^{LHS} be the value returned by the outermost LHS term and ret^{RHS} be the value returned by the outermost RHS term. A rule is correct if *assuming* (1) the semantics of the LHS and RHS terms and (2) preconditions of the LHS *implies* (1) the equivalence of the LHS and RHS and (2) preconditions on the RHS terms:

$$\forall \{i_0, \ldots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} | \\ (P^{LHS} \wedge R^{LHS} \wedge P^{RHS}) \Rightarrow (ret^{LHS} = ret^{RHS}) \wedge R^{RHS} \quad \lfloor 3.2$$

To convert this statement to an SMT query, VeriISLE plays the standard trick of asking if there are counterexample inputs such that the verification conditions do not hold (by switching the quantifier to an existential and negating the implication).

Verification conditions for narrow widths ISLE's type system itself conveys to VeriISLE which bits are demanded to produce the right verification conditions. For many rule and type instantiation pairings, the expression ret^{LHS} (the returned value from the outermost LHS term) has a width narrower than 64 bits. The RHS, however, typically operates on register-width values with 64 bits. In such cases of mis-matched widths, the condition VeriISLE verifies aligns with Cranelift IR's intended invariant: that the lower bits of the register are equivalent to the Cranelift IR semantics on the narrow width. We implement this condition in VeriISLE by adding an annotation on the `output_reg` term, which the ISLE preprocessor inserts as an automatic type conversion:

```
(spec (sig (args arg) (ret))
  (provide (= ret (convto (widthof ret) arg))))
(decl output_reg (Reg) InstOutput)
```

The `convto` in this annotation narrows the bits of `Reg` in consideration to the bit demanded by the width of the `InstOutput` (which models the potentially narrow Cranelift IR type).

Optional custom verification conditions and assumptions Some compiler transformations intentionally break strict equivalence. For example, Cranelift attempts to rewrite comparisons that include a statically-known argument to prefer an even integer immediate:

as a mathematical identity, $A \geq B + 1 \rightarrow A - 1 \geq B \rightarrow A > B$. This rewrite is profitable because even values are more likely to fit in ARM64's 12-bit immediate encodings, improving code size.

The rule that implements this identity is closely tied to how comparisons are emitted to machine code. On ARM, comparisons are done by a subtraction-with-flags and then comparing those flags again the condition code for the specific comparison (in this example, $\geq$ vs $>$). The relevant rule acts on terms that that produce the ISLE type `FlagsAndCC`, rather than a boolean value directly. Since the mathematical identity changes the values of both the flags and the condition code, VeriISLE reports a verification failure on this and similar rules.

Optionally, users can run VeriISLE with custom verification conditions instead of checking strict bitvector equality of the LHS and RHS. In this case, VeriISLE can encode the logic that flattens flags and a condition code into a boolean in order to prove that the *boolean* result of the comparison maintains equivalence. Users can also provide VeriISLE with additional assumptions on input values, which we use to encode cases where a rule would not match due to ISLE's priority semantics.

3.0.3 Implementation and trust model

VeriISLE is implemented 15,825 lines[3] of Rust as a fork of the Wasmtime codebase.[4] We run VeriISLE queries as a Rust test suite in continuous integration on our Wasmtime fork. VeriISLE is designed to be useful to compiler engineers who are not experts in verification tooling; VeriISLE lifts counterexamples from the SMT model back into ISLE syntax to make debugging easier. VeriISLE can also test rules against specific concrete inputs (i.e., run as an interpreter), allowing developers to test their annotations against their expectations (and paving the way for future work in fuzzing VeriISLE's annotations).

Caveats and the trusted code base VeriISLE is limited to reasoning about individual rewrite rules written in ISLE; it reasons about correctness in instruction lowering itself, but trusts other passes in the Cranelift compiler and Wasm runtime. Cranelift and the Wasmtime engine invoke instruction selection *after* WebAssembly safety checks are inserted, but prior to a couple final compiler stages (e.g., register allocation).[5] VeriISLE also trusts the semantics of ISLE terms as written in the annotation language (though our `provide` and `require` distinction and concrete tests help find bad specifications). For example, we found that an old version of VeriISLE did not require condition codes to fall into a valid range. Finally, VeriISLE currently reasons about each rule individually. Support for verifying properties over multiple rules (e.g., reasoning about rule priorities) is future work.

[3] Plus 26,465 lines for our auto-generated annotation language parser.

[4] Forked at commit `9556cb1`.

[5] Cranelift also has a distinct symbolic translation validation checker for register allocation; this shows how engineers can take an ensemble approach to applying formal methods in a production setting.

4 Evaluation

This section answers the following evaluation questions:

Q1 Can VeriISLE be applied to a meaningful set of ISLE rules?

Q2 For test and benchmark suites for WebAssembly and Rust, what proportion of invoked ISLE rules has VeriISLE verified?

Q3 Can VeriISLE reproduce prior, known Cranelift bugs?

Q4 Can VeriISLE help identify and fix new bugs?

We answer **Q1** by verifying a natural subset of rules, those necessary to compile integer computations in the latest major release of WebAssembly ("1.0" [66]). Section 4.0.2 addresses **Q2**—we find that the rules we verify comprise wasm-no-simd-rules of the lowering rules invoked by the WebAssembly reference test suite.

To answer **Q3**, we choose two previously-discovered CVEs in ISLE rules (out of 14 Wasmtime CVEs, 10 of which do not involve ISLE); we also select an ISLE bug that was not assigned a CVE because it affects non-Wasm types. We annotate the buggy rules and present the counterexamples VeriISLE produces in Section 4.0.3.

Finally, in Section 4.0.4 we address **Q4**, outlining 3 new faults (2 patched) that VeriISLE discovered, and 1 *compiler mid-end* bug that VeriISLE helped root-cause and patch. These case studies highlight that instruction-lowering rules are error-prone even for experienced compiler engineers: many of the issues were subtle interactions between constants, sign- and zero- extensions, and tricky bitwidth-specific reasoning. Moreover, to our knowledge, no new bugs have been discovered by any other means (e.g., any Cranelift fuzzers [6]) in rules verified by VeriISLE.

	Total	**Success**	**Timeout**	**Inapplicable**	**Failure**
Rules	rules	rules-succeeded	rules-timedout	rules-inapplicable	rules-failed
Type Instantiations	type-invs	invs-succeeded	invs-timedout	invs-inapplicable	invs-failed

Table 4.1: Verification results for rules and type instantiations (because rules match on multiple possible types, potentially with different verification results) for integer operations from WebAssembly 1.0 to Arm `aarch64`. Note that the failures all succeed with custom (rather than bitvector equivalence) verification conditions.

4.0.1 Is VeriISLE applicable to real rules?

We use VeriISLE to verify the instruction-lowering rules for all integer operations[1] from WebAssembly's 1.0 release to the ARM `aarch64` backend. In addition, we verify most of the new integer operations in WebAssembly's 2.0 version, which is currently in draft status [67]. We choose these rules because WebAssembly uses integers for addressing computations, which means that logical issues in integer codegen can lead to security vulnerabilities. We verify `aarch64` rules because this backend is less well-tested than `x86-64`. The ARM backend rules we *do not* verify fall into four categories: (1) i128 types; (2) floating point; (3) SIMD (vector) instructions; and (4) side effects and control flow. We discuss further in Section **??**.

Verification requires 182 total annotations (1075 LOC). For some ISA terms, we modify or cross-reference formal semantics from SAIL-ISLA [4, 5], a symbolic execution engine for ISAs. For Cranelift IR and external Rust terms, we refer to WebAssembly's specification, Cranelift documentation, and the external Rust definitions.

In total, our verification effort covers rules distinct rules with type-invs type invocations, since each rule is tested against 1 to 10 possible type assignments. For most rules, we consider all Cranelift-supported integers up to 64 bits (i.e., `i8`, `i16`, `u/i32`, and `u/i64`), though we note that WebAssembly 1.0 only supports 32-bit and 64-bit numbers. `rustc_codegen_cranelift`, an alternative backend for the Rust language, uses the narrower types VeriISLE supports [58, 10].

Table 4.1 shows the verification results for all type-invs total type invocations. Recall that the six verification failures do not represent real bugs, since the context in which they are used does not require bitvector equivalence. With custom verification conditions, these rules verify successfully. type-invs-term of the type-invs invocations complete, in sum, within 5 minutes on a laptop.[2] The rules-timedout-number rules that timeout on some type instantiations contain multiplication, division, remainder, and `popcnt` operations on bitvectors, which are difficult for SMT solvers to reason about for wider widths [40].[3] Each of these rules fails with a counterexample within 10 seconds if we inject a flaw in the rule logic.

4.0.2 What proportion of invoked rules has VeriISLE verified?

We instrument Cranelift to determine, on various targets, what proportion of invoked ISLE rules VeriISLE has verified. For the WebAssembly reference test suite, VeriISLE verifies wasm-no-simd-rules (wasm-no-simd-rules-frac) of the unique ISLE rules used during compilation. (We use a version of the WebAssembly specification's test suite that corresponds to the language features in Wasm 1.0, which notably excludes SIMD instructions.) To assess our coverage on integer types narrower than those that Wasm supports, we repeat this experiment on the `rustc_codegen_cranelift` test suite, an alternative backend for the Rust compiler that uses Cranelift as its code generator [58, 10]. Verified rules make up rust-no-simd-rules (rust-no-simd-rules-frac) of the unique ISLE rules used during compilation. These numbers will grow as we enhance VeriISLE to additional memory operations and floating point (§**??**).

[1] All operation defined under section "4.3.2 Integer Operations" of the WebAssembly Specification Release, 1.0

[2] We run experiments on a MacBook Pro Apple M2 Max, 12-core CPU, 32GB RAM, macOS 13.2.1.

[3] Timed out after 6 hours, run in parallel with other tests.

4.0.3 Can VeriISLE detect known bugs?

To answer our third question, we use VeriISLE to detect three known, recent Cranelift bugs. We select these bugs for their severity and because they occur in ISLE rules in scope for the current version of VeriISLE.

`x86-64` addressing mode CVE (9.9/10 severity)

In under one second on a laptop, VeriISLE detects a 2023 CVE in `x86-64` instruction lowering that permitted a WebAssembly sandbox escape (§**??**) [25]. The reproduction requires 13 new annotations to support terms in the `x86-64` backend, which we had not previously covered (§4.0.1).

The bug appeared in this ISLE rule:[4]

```
(rule
  (amode_add (Amode.ImmReg off base)
             (uextend (ishl x (iconst shft))))
  (if (u32_lteq (u8_as_u32 shift) 3))
  (Amode.ImmRegRegShift off base
    (extend_reg x I64 (Extend.Zero)) shft))
```

This rule intends to take advantage of an `x86-64` addressing mode that allows shifts to be computed within the instruction itself, before adding together address components. However, the core problem with this rule (§**??**) is that the LHS performs a shift on a 32-bit value (throwing away any bits that are shifted left beyond 32 bits), while the RHS performs the shift on a 64-bit value (throwing away bits shifted left beyond *64* bits), which lets the emitted shift modify bits beyond WebAssembly's effective address space.

To see how the problem manifests, we walk through the rule. The outermost LHS term, `amode_add`, is an intermediate term that earlier rules construct to model memory address computations that can be folded into addressing modes. The second argument of the match, `(uextend ...)`, is a Cranelift IR value that is a zero-extended (`uxtend`) shift operation (`ishl`) with a statically known, constant shift amount (`shft`) (conceptually `(i64.extend_i32_u ( i32.shl <x> (i32.const <shft>)))`). The rule's `if` clause checks that the shift amount, `shft`, is less than or equal to 3. If all the above conditions hold and the rule matches, it emits a single addressing mode where the value `x` to be shifted is zero-extended, shifted by the static `shft` amount, and added to the other components of the computed address (`base + off`).

VeriISLE provides the following counterexample:[5]

```
(amode_add
  (Amode.ImmReg
    [off|#x30c04100] [base|#x0000000000000000])
  (uextend
    (ishl [x|#xd0000920] (iconst [shft|#x02])))) =>
(Amode.ImmRegRegShift
  [off|#x30c04100]
```

[4] Lightly edited for brevity

[5] Lightly edited for brevity.

```
  (gpr_new [base|#x0000000000000000])
  (extend_to_gpr [x|#xd0000920] I64 Extend.Zero)
  [shft|#x02])

#x0000_0000_70c0_6580 =>
#x0000_0003_70c0_6580
```

In this counterexample, the 32-bit value `x`, `#xd0000920`, has the most significant bit set. When `x` is shifted by the specified 2 bits to the left, the results differ on the LHS and RHS. As expected, the LHS throws away the shifted bits after 32 bits (e.g., the higher 32 bits of `#x0000_0000_70c0_6580` are zero). However, the RHS *does not throw away* the shifted bits after 32 bits, allowing non-zero bits beyond the expected effective address space: `#x0000_0003_70c0_6580`!

The patch for this bug simply removes the rule entirely, so we did not verify the patch with VeriISLE.

aarch64 unsigned divide CVE (moderate severity)

VeriISLE reproduces a 2022 CVE in `aarch64` instruction lowering in which divides with constant divisors were miscompiled. In this case, *trying to write annotations was enough to highlight the root cause of the bug*—that constant values, when used as divisors, were not correctly sign- or zero-extended according to signed or unsigned division.

The ISLE rules that matched on constant divisors for both `udiv` and `sdiv`—unsigned and signed divide—used the term `imm` on the RHS. `imm` models an immediate value that can be encoded in a machine instruction itself, lowering both the number of instructions and register pressure. At the time of this CVE, the ISLE signature for `imm` was:

```
(decl imm (Type u64) Reg)
```

This term's intention was to take the immediate's value as a `u64` and place it in a register. When trying to annotate this term and the terms for signed constant divisors, though, an issue was immediately clear: `imm` provides *no argument* for whether narrow values should be sign- or zero-extended. Annotating zero-extension causes signed division to fail; choosing sign-extension causes unsigned division to fail. In practice, the external Rust implementation sign-extended, so the bug surfaced in `udiv` instructions. The patched version of `imm` takes in an argument for the type of extension, and the rules for `udiv` and `sdiv` now successfully verify.[6]

aarch64 count-leading-sign bug

VeriISLE reproduces a pre-existing bug in the ISLE `aarch64` lowering rule for `cls`, the instruction that counts the number of leading sign bits in a value (excluding the sign bit itself). The rule for narrow `cls` instructions must extend its input values, since Cranelift IR supports operations on narrow types like `i8` and `i16`, while `aarch64` only supports operations on 32- and 64-bit values. Unfortunately, the faulty version of the rule failed to properly extend:

[6] Though as noted previously, VeriISLE times out on some wide divisions.

```
1 (rule
2   (lower (has_type I8 (cls x)))
3   (a64_sub_imm I32 (a64_cls I32 (zext32 x)) 24))
```

This rule matches on `cls` computations over 8-bit values. The RHS extends 8-bit `x` to 32 bits using `zext32`, and then computes `a64_cls` on this wider value. Finally, it subtracts 24 bits $(32 - 8)$ to obtain the leading bit count on the narrow type. VeriISLE reports the following counterexample:

```
1 (lower (has_type I8 (cls [x|#b11111100]))) =>
2 (output_reg
3   (a64_sub_imm I32
4     (a64_cls I32 (zext32 [x|#b11111100])) 24))
5
6 #b00000101 => #b11111111
```

In this counterexample, the LHS correctly computes that the value `#b11111100` has 5 leading sign bits (1), excluding the sign bit itself. The RHS, however, zero-extends this value to 32 bits, then counts the new leading sign (0) to produce 23, and subtracts 24 to produce -1. The amended version of the rule uses a sign-extend instead of a zero-extend, and VeriISLE verifies it successfully.

4.0.4 Can VeriISLE find new bugs?

This section outlines VeriISLE's discoveries in Cranelift so far: two bugs, both patched; a case of imprecise semantics; and a root cause analysis.

Another addressing mode bug

VeriISLE discovered a new correctness bug in an `x86-64` addressing mode rule related to the one discussed in §4.0.3 (which was not identified by Cranelift engineers even in a subsequent close look at addressing mode rules). This rule was identical except that it did not have an explicit `uextend` (line 3 in §4.0.3)—the same bug could surface on a direct load of a 32-bit address. Cranelift developers determined that the bug would not be triggered in practice because on 64-bit targets, all addresses should be 64-bit typed, and frontends generate code in this form. However, nothing in the compiler backend validated this IR invariant and the bug *could* be triggered if frontend implementations changed. Cranelift engineers patched this issue immediately after we notified them of VeriISLE's result.

Flawed negated constant rules

VeriISLE found an issue where 3 rules were unintentionally restricted to never fire in practice. This was a performance issue—optimizations did not apply as often as they should—but not a correctness issue. The three buggy rules all, in various ways, attempted but failed to find small, constant arguments that could be encoded in ARM's `imm12` encoding. This is an optimization because it is an alternative to the more expensive option of using a separate load-immediate instruction.

This is one of the buggy rules VeriISLE discovered:

```
(rule
  (lower (has_type (fits_in_64 ty)
    (isub x (imm12_from_negated_value y))))
  (a64_add_imm ty x y))
```

The `imm12_from_negated_value` term matches when the second argument, after being negated, can be encoded into ARM's 12-bit immediate format. Matching *negated* constants allows a wider range of numbers to be encoded as immediates: around 8,000 constant values can be encoded in ARM's `imm12` (12 bits plus a shift bit)—checking for negated values as well doubles the number of possible constants.

When run on this rule, though, VeriISLE warns that there are `no distinct models`—the rule only matches *one* set of input values. The issue is in the (external Rust) implementation of `imm12_from_negated_value`:

```
Imm12::maybe_from((n as i64).wrapping_neg() as u64)
```

In Cranelift's IR, all constant integers are represented with Rust's `u64` type. This code takes the constant `n`'s underlying `u64` value, negates it, and checks if it fits into an `Imm12` immediate. Unfortunately, for *any* width of integer narrower than 64 bits, the only value this holds true for is zero! This is because Cranelift has an informal invariant that when a negative narrow value is stored as a constant, its value should be zero-extended—not sign-extended—into a `u64` representation. Negating (`wrapping_neg`) a zero-extended constant always produces a 64-bit value with with left-filled *ones*, which will always fail the check `Imm12::maybe_from` because the highest bits on the 64-bit value are set.

VeriISLE discovered that, while not *incorrect*, this rule was useless—it never matched in practice. Our merged fix corrects this rule to negate the narrow constant *and then* zero extend it.

Imprecise semantics for constants in Cranelift IR

VeriISLE also found that Cranelift had under-specified semantics for integer constant representations in IR. While most Cranelift front-ends zero-extend narrow constant values to 64 bits, VeriISLE found that Cranelift's own parser for unit tests sign-extends. The issue we filed is the site of ongoing discussion about enforcing clear semantics; since then, a fuzzer discovered a bug in Cranelift's mid-end optimizations caused by the same imprecise semantics.

A mid-end root cause analysis

While we designed VeriISLE for ISLE's lowering rules, we have found that it can reason about backend-agnostic rewrites—rewrites in the compiler "mid-end"—as well. In this case study, VeriISLE identified the root cause of a new bug—a boolean optimization rewriting false to true—*before* Cranelift engineers identified it.

A Cranelift engineer ran Souper—a superoptimizer for LLVM [57]—on a subset of Cranelift IR and discovered that Cranelift was missing the boolean rewrite `or(and(x, y), not(y) )== or(x, not(y))`. To port this to ISLE, the engineer wrote a new rule with an explicit

guard to check the for a bitwise-not between constants y and z:[7]

```
(rule
  (simplify (bor (band x (iconst y)) (iconst z)))
  (if (u64_eq zk (u64_not y)))
  (bor x z))
```

This rule passed code review and was merged, but broke an integration test with a `wasm trap` error that did not point to a root cause. Before the Cranelift engineers were able to complete a manual investigation, we extended VeriISLE analyze this rule (e.g., added annotations for mid-end terms) in under two hours. VeriISLE produced the following counterexample:[8]

```
(bor (band [x|#b1] [y|#b1]) (iconst [z|#b0])) =>
(bor [x|#b1] [z|#b0])
#b0 => #b1
```

VeriISLE surfaces a subtle bug related to the semantics of ISLE's `if` construct. Recall that terms in ISLE are partial functions. The semantics of ISLE's terms with external Rust implementations are that a match should continue if the return value is `Some(...)` and should not match if any LHS term returns `None`. Deceptively, because the Rust external definition of term `u64_eq` in the prior rule returned `Some(false)` instead of `None` (that is, the boolean was *defined*, just *false*) this guard as written *always* allowed the match to proceed!

To fix this bug, Cranelift engineers re-wrote the guard to actually check for `Some(true)`. VeriISLE's analysis also led Cranelift engineers to propose a longer-term solution—redesigning semantics of `if` to avoid similar mistakes in the future. Finally, after the patch was in, a Cranelift engineer said, "this would have taken me so much longer without the counterexample, really helpful!"

This case study has a another unexpected takeaway: this bug occurred despite the optimization being harvested from *another formal-methods-based tool!* While the Souper superoptimizer is also based on the SMT theory of bitvectors, the subtle interaction between Souper-IR and ISLE semantics could not have been caught by Souper itself. This highlights the benefits of VeriISLE's tight integration with ISLE's own program representation: VeriISLE was able to root-cause this bug because it must reason about core ISLE semantics.

[7] Lightly edited for clarity and brevity.
[8] Example truncated to 1 bit for brevity.

5 Related Work

Compiler verification Compiler verification research falls into two broad categories: lightweight verification of (parts of) existing compilers using solvers (e.g., [45, 48, 47]), and clean-slate, foundational verification using proof assistants [13] (e.g., CompCert [49, 44]). Foundational verification provides end-to-end correctness guarantees at the cost of time and performance: typically, such verification takes experts many years [68], and makes serious optimizations impractical. There are manually verified lowering passes for CompCert [50] and CakeML [69, 34], but not for production compilers that consider performance first-class.

Other works use solver-backed methods to verify portions of industrial compilers. Most closely related to VeriISLE, Alive [52] verifies LLVM [46] peephole optimization rules written in a DSL. Alive's main challenge is undefined behavior; in contrast, VeriISLE need not reason about undefined behavior, but must instead reconcile IR and ISA types. Further afield, Alive2 [51] does translation validation on LLVM IR, and VeRA [15] verifies range analysis in the Firefox JavaScript engine. Finally, Jitterbug [59] verifies lowering from BPF, a setting where instruction selection entails simple "macro expansion" of one instruction at a time.

WebAssembly verification. VeriWasm proves that individual binaries do not violate Wasm's safety guarantees [42]. VeriWasm does not prove *compiler* correctness, though, and places restrictions on how Wasm compilers can emit native code.[1] In [14], the authors present a non-optimizing compiler to `x86-64` that is verified to preserve sandbox safety, and a non-optimizing compiler from Wasm to Rust; in contrast, we verify the correctness of a production, optimizing compiler.

There is also work on mechanizing the Wasm specification [73] and formalizing Wasm in the K framework [37]. Other verification efforts look beyond the language and compiler: WaVE [41] verifies that interactions between the Wasm runtime and the host OS preserve safety guarantees; SecWasm [12] extends Wasm's guarantees using information flow control; [62] bring verified cryptography to Wasm; and CT-Wasm extends Wasm itself with constant-time guarantees [74].

Synthesizing instruction selectors. The complexity of instruction selection has inspired work on automatically generating rules based on machine-language semantics. Because of their focus on portability vs. correctness, many instruction selector generators use *ad hoc*

[1] After discovering the `amode` bug described in the introduction, Cranelift engineers tried to update VeriWasm to operate on the current version of the backend, but determined it would be too large of an undertaking.

search procedures instead of solver-aided techniques [39, 19, 21, 30]. Others use solver-aided synthesis: LibFIRM [16], for example, uses SMT to synthesize new rules that cover about 75% of input instructions, while using an existing, handwritten rule set for the rest. [26] uses a solver to generate high-coverage selection simple rules for diverse target architectures. Rake [2] synthesizes lowering rules from Halide [63] to digital signal processor ISAs, but its focus is on capturing complex data movement mechanics within vector registers instead of general-purpose instruction semantics. Though many compilers use a DSL to express instruction selection rules, to our knowledge VeriISLE is the first tool for verifying existing rules by modeling DSL semantics.

Formal semantics for ISAs. Several efforts formalize ISA semantics, including the SAIL language [4] and the K Framework [27]. In the future, we will extend VeriISLE to exploit these existing semantic models.

6 Future Work

VeriISLE annotations are currently trusted. We can address this issue by deriving certain annotation from existing formal models. For example, VeriISLE can integrate SAIL semantics for `aarch64` [4] and K framework semantics for `x86-64` [27]. While neither Cranelift IR nor external Rust term definitions have formal semantics, we can raise assurance in our specifications by, for example, verifying them against their external Rust implementations [7, 8, 64].

Future work can extend VeriISLE to reason about floating point, more operations with side effects, some SIMD vector instructions, and wider integers. VeriISLE already incorporates annotations for *some* 128-bit vector instructions, because the implementation of `popcnt` on `aarch64` uses them. VeriISLE can also be extended to automatically reason about rule priorities and to cover other backends and the mid-end optimizer.

VeriISLE is meant to be used. We are working to upstream it into mainline Cranelift, which raises research questions around usability: how can a formal methods tool best support engineers who are experts in their domain, but not necessarily in verification? We hope to explore these questions as we improve VeriISLE, and as we build on VeriISLE to create more comprehensive verification infrastructure for other parts of the compiler.

7 Conclusion

Language-based technologies such as WebAssembly promise a more secure computing environment, where hosts can safely sandbox untrusted code to limited segments of memory. This software-level isolation, though, fundamentally places an incredibly high burden (full functional correctness!) on the compiler that produces the final executable in a machine-specific ISA. VeriISLE is a tool for verifying instruction-lowering rules in one such safety-critical compiler: the Cranelift code generator. VeriISLE's key selling point is its modularity—VeriISLE's annotation language allows concise semantics of individual terms to be added alongside definitions in ISLE, a feature-rich instruction-lowering DSL. With VeriISLE, compiler developers can eliminate instruction lowering logic as a potential source security-critical vulnerabilities such as sandbox escapes. VeriISLE builds toward a future where heavily optimized, production compilers can integrate advanced formal methods to produce fast *and* correct machine code.

Bibliography

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. 2018.

[2] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. 2022.

[3] Bytecode Alliance. ISLE language reference. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/isle/docs/language-reference.md, 2023.

[4] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. 2019.

[5] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. 2021.

[6] Javier Cabrera Arteaga, Nicholas Fitzgerald, Martin Monperrus, and Benoit Baudry. Wasm-mutate: Fuzzing WebAssembly compilers with e-graphs. In *E-Graph Research, Applications, Practices, and Human-factors Symposium*, 2022.

[7] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. 2019.

[8] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. Verifying Rust programs with SMACK. 2018.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2005.

[10] Björn Roy Baron et al. Cranelift codegen backend for Rust, 2023.

[11] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[12] Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. Secwasm: Information flow control for WebAssembly. In *Static Analysis*, 2022.

[13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[14] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using WebAssembly. 2022.

[15] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. 2020.

[16] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. 2018.

[17] Bytecode Alliance. The Cranelift compiler. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift, 2023.

[18] Bytecode Alliance. Wasmtime: A fast and secure runtime for WebAssembly. https://wasmtime.dev, 2023.

[19] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 1980.

[20] R G G Cattell. *Formalization and Automatic Derivation of Code Generators*. PhD thesis, Carnegie Mellon University, 1978. https://apps.dtic.mil/sti/pdfs/ADA058872.pdf.

[21] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C compiler retargeting based on instruction semantics models. 2005.

[22] Alex Crichton. Data leakage between instances in the pooling allocator. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-wh6w-3828-g9qf, November 2022.

[23] Alex Crichton. Miscompilation of constant values in division on aarch64. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-7f6x-jwh5-m9r4, July 2022.

[24] Alex Crichton. Miscompilation of 'i8x16.swizzle' and 'select' with v128 inputs. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-jqwc-c49r-4w2x, 2022.

[25] Alex Crichton. Guest-controlled out-of-bounds read/write on x8664. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8, 2023.

[26] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from RTL using SMT. 2022.

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

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008.

[29] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 1991.

[30] João Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. 2010.

[31] Chris Fallin. Memory access due to code generation flaw in Cranelift module. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-hpqh-2wqx-7qp5, May 2021.

[32] Chris Fallin. RFC: Design of the ISLE instruction-selector DSL. https://github.com/bytecodealliance/rfcs/pull/15, August 2021.

[33] Chris Fallin. Cranelift's instruction selector DSL, ISLE: Term-rewriting made practical. https://cfallin.org/blog/2023/01/20/cranelift-isle/, January 2023.

[34] Anthony Fox, Magnus O Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. 2017.

[35] Go Authors. Go compiler backend lowering rules. https://github.com/golang/go/tree/master/src/cmd/compile/internal/ssa/_gen, 2023.

[36] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. 2017.

[37] Rikard Hjort. Formally verifying WebAssembly with KWasm, 2020.

[38] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, 1969.

[39] Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. 1996.

[40] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Computer Aided Verification*, 2009.

[41] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. 2023.

[42] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Trust but verify: SFI safety for native-compiled Wasm. 2021.

[43] Kenton Varda. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

[44] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. 2014.

[45] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. 2009.

[46] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. 2004.

[47] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. 2003.

[48] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. 2005.

[49] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[50] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[51] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. 2021.

[52] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. 2015.

[53] Nuno P. Lopes and John Regehr. Future directions for optimizing compilers. 2018.

[54] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. In *ACM Transactions on Programming Languages and Systems*, 1982.

[55] Charith Mendis and Saman Amarasinghe. GoSLP: Globally optimized superword level parallelism framework. 2018.

[56] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, 2001.

[57] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. Dataflow-based pruning for speeding up superoptimization. 2020.

[58] Joshua Nelson. Using rustc_codegen_cranelift for debug builds. https://blog.rust-lang.org/inside-rust/2020/11/15/Using-rustc_codegen_cranelift.html, November 2020.

[59] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. 2020.

[60] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. Towards bit-width-independent proofs in SMT solvers. 2019.

[61] Pat Hickey. Lucet takes WebAssembly beyond the browser — Fastly. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[62] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[63] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. 2013.

[64] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are. 2020.

[65] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for SIMD devices. page 118–131, 2006.

[66] Andreas Rossberg. WebAssembly Specification Release 1.0. https://webassembly.github.io/JS-BigInt-integration/core/_download/WebAssembly.pdf, 2019.

[67] Andreas Rossberg. WebAssembly Specification Release 2.0 Draft Draft 2023-04-08. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, 2023.

[68] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. 2015.

[69] Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.

[70] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. 2021.

[71] Vercel Inc. Using WebAssembly (Wasm) at the edge. https://vercel.com/docs/concepts/functions/edge-functions/wasm, 2023.

[72] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. 1998.

[73] Conrad Watt. Mechanising and verifying the WebAssembly specification. 2018.

[74] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the web ecosystem. 2019.

[75] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. 2011.