Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection

Alexa VanHattum¹, Monica Pardeshi², Chris Fallin³, Adrian Sampson⁴, and Fraser Brown²

¹Wellesley College ²Carnegie Mellon University ³Fastly ⁴Cornell University

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 Crocus, a system for lightweight, modular verification of instruction-lowering rules within Cranelift, a production retargetable Wasm code generator. We use Crocus to verify lowering rules that cover WebAssembly 1.0 support for integer operations in the ARM aarch64 backend. We show that Crocus 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.

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 CVE¹ 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:

\[
(\text{i32.load} \ (\text{i32.shl} \ \text{\{local.get x\}} \ (\text{i32.const 3})))
\]

To lower this code to x86-64, Cranelift must convert 32-bit Wasm addresses into offsets 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 \(\text{base} + \text{zext}(\text{addr})\) (where \(\text{addr}\) is the original 32-bit Wasm address, \(\text{base}\) is the base address for the module’s memory region, and \(\text{zext}\) is a zero-extension). Unfortunately, the Cranelift instruction selector lowered the above Wasm code to x86-64 instructions that computed \(\text{base} + \text{zext}(\text{x} << 3)\) instead of \(\text{base} + \text{zext}(\text{x} << 3)\). 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

¹“Common Vulnerabilities and Exposures”, a designated list of publicly disclosed security bugs.
To help compiler developers automatically reason about the correctness of their instruction-lowering rules, we present Crocus. Crocus verifies rules written in Cranelift’s ISLE domain-specific language (DSL) for specifying how IR terms translate to machine code sequences. To use Crocus, developers annotate their ISLE lowering rules with specifications; Crocus 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. Crocus 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. The annotation language has been designed in collaboration with Cranelift engineers, to ensure that annotations can be co-maintained without undue burden. To our knowledge, our work with Crocus 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 Crocus, a framework for verifying instruction-lowering rules in the ISLE domain-specific language.
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 Crocus to reproduce and detect previously-fixed bugs (§4.3.3) and vulnerabilities (§4.3.1), including the example bug from this section.
4. Use Crocus to help Cranelift developers identify (§4.4.1, §4.4.2) and fix (§4.4.4) new bugs and under-specified compiler invariants (§4.4.3).

We begin by introducing background on instruction lowering and the ISLE DSL (§2.1). Then, we present Crocus’s design (§3), and evaluate its results on Cranelift (§4), a production Wasm compiler backend. Finally, we discuss plans to build on Crocus towards fully-verified Wasm compilers (§6).

2. Background

This section provides background for understanding Crocus verification (§3) by describing the instruction lowering problem (§2.1) and Cranelift’s ISLE domain-specific language (DSL) for writing lowering rules (§2.3). Finally, it introduces SMT solvers [11], the tools that power the Crocus verification engine (§2.4).

2.1. 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.

2.2. The ISLE lowering DSL

The Cranelift compiler project [17] introduced the ISLE (Instruction Selection Lowering Expressions) 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.

2.2.1. 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:

\[ \text{lower (iadd \( ty \) \( x \) \( y \))} \]

where iadd is defined in Cranelift IR and iadd_ty 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 Registers. lower, with type signature \((\text{decl} \text{lower (Inst)Reg)}\), \(^3\) does the Inst to Register 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 \( y \) Registers. To reconcile these incompatible types, the ISLE compiler automatically inserts type conversions if a conversion 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 Values to Regs. \(^4\)

### 2.3. 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 (§3).

Consider the Wasm rotrl and rotr (“rotate”) binary numeric instructions, which shift the bits of a value left or right with wraparound. Cranelift has corresponding rotrl 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:

\[ \text{lower (rotr x y)} \]

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:

\[ \text{lower (has_type (fits_in_16 ty) (rotr x y))} \]

This rule uses external helper terms has_type and fits_in_16 to predicate this rule only on narrow types; if the number of bits (\( ty \)) 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 rotr1 (rotate left) rule for narrow inputs also uses the small_rotr term. The compiler uses a small_rotr with a negated rotate amount because ARM does not have a distinct rotate left instruction:

\[ \text{lower (has_type (fits_in_16 ty) (rotl x y))} \]

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).

\(^2\)Slightly simplified for clarity; real rules differentiate on the values’ types.

\(^3\)We elide an indirection via another type for clarity.

\(^4\)We describe the semantics of put_in_reg in §3.1.2.
2.4. Satisfiability Modulo Theories (SMT)

To verify lowering rules written in ISLE, Crocus uses an SMT solver [28]. SMT solvers are tools that determine whether logical formulas are satisfiable for some assignment of values to all variables in the formula. Unlike SAT formulas [56], SMT formulas allow users to express higher-level statements (e.g., \( x < y \)) using a rich set of operators and types (e.g., integers and arrays) that are defined in the SMT-LIB standard [11]. Crocus lowers ISLE rules to SMT formulas in the theory of bitvectors and integers; we discuss this further in the next section.

3. Crocus Design

Crocus is a framework for verifying rewrite rules in the ISLE domain-specific language for instruction selection. Crocus uses an SMT solver [28] to show functional equivalence of the left- and right-hand sides of individual rules. 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.

The initial version of Crocus supports pure functions that model the computations on SSA-style values. This is in part because Cranelift’s instruction selection pass comes before register allocation, so it operates on abstract, immutable SSA values rather than on concrete, mutable machine registers (see Section 3). In practice, Crocus is able to find nuanced bugs and raise the level of assurance in critical code even with this restriction (see Section 4).

To verify their lowering rules, compiler developers write annotations on ISLE terms in Crocus’s annotation language (§3.1). This language makes it simple to express term semantics (e.g., that \( \text{fits}_{\text{in}}_{16} \) means that a type can losslessly be represented with 16 bits). Crocus consumes ISLE’s program representation for rules, combines this with the compiled annotations to create its own intermediate representation, and performs type inference (§3.1.3). Type inference is necessary for Crocus 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, Crocus 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.

The annotation language has been designed in collaboration with Cranelift engineers so that it fits well into the ISLE ecosystem and can be co-maintained with the lowering rules. This constraint led us to co-locate annotations in the main ISLE source files. These decisions make it more feasible for production compiler engineers to engage with the verification effort.

In this section, we walk through the verification pipeline, from Crocus’s annotation language (§3.1) to how it constructs and customizes verification conditions (§3.2).

3.1. The annotation language

It is impossible to verify functional correctness without precise semantics on terms within ISLE. While there are formal semantics for parts of 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, Crocus 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 Crocus annotation on the helper term \( \text{fits}_{\text{in}}_{16} \):\(^6\)

\[
\begin{align*}
&\text{(spec (fits\textunderscore in\textunderscore 16 arg)} \\
&\text{(provide (= result arg)} \\
&\text{(require (<= arg 16)))} \\
&\text{(decl fits\textunderscore in\textunderscore 16 (Type) Type)}
\end{align*}
\]

This specification says that \( \text{fits}_{\text{in}}_{16} \) is a partial identity function on the argument type Type—that is, for the arguments on which \( \text{fits}_{\text{in}}_{16} \) is defined, it returns the argument itself. The function is specified by the provide clause (= result arg), which sets the return value equal to the first argument; both variables are bound in the spec signature. require clauses specify a precondition on the term. This precondition specifies that the rule is a partial function predicated on (<= arg 16)—the fact that the argument, which Crocus 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 \( \text{fits}_{\text{in}}_{16} \): it is a partial identity function that returns the type argument arg, which matches if arg is under sixteen bits.

3.1.1. The annotation language grammar and semantics

Figure 1 shows the Crocus annotation language grammar. Figure 2 provides judgements that specify the typing and semantics of Crocus’s annotation language. Most operations in the annotation grammar map directly to SMT-LIB constructions. For example, \(+\) applied to a bitvector maps to SMT-LIB’s \(\text{bvadd}\) bitvector addition function. Crocus provides a special `result` keyword expression which models the value produced by the annotated term.

Crocus adds conveniences like \texttt{switch} and a variadic \texttt{concat} operation, both of which desugar to folding SMT-LIB’s fixed-argument \texttt{ite} (if-then-else) and \texttt{concat} (bitvector concatenation) operators. \texttt{switch} also adds a verification

\(^5\) Though Crocus supports more general custom verification conditions, as we will describe later in this section.

\(^6\) ISLE terms and specification syntax lightly edited for clarity and brevity.
where function preconditions may be assumed within the body 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 requires 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:

1 (require (switch ty
2 (8 (= (extract 63 8 x) (0:bv)))
3 (16 (= (extract 63 16 x) (0: bv))))

This require clause specifies 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.

3.1.2. The annotation language type system

Types in Crocus are integers, booleans, and bitvectors. The Crocus annotation language must support polymorphism over bitvector widths, since most of Cranelift’s ISLE rules operate on its Value type, which is polymorphic over integer values in the Cranelift intermediate representation. (§2.3). For example, during preprocessing, ISLE automatically inserts put_in_reg to implicitly convert Cranelift IR Values to machine code Regs—and because Values vary in width, Crocus’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. Crocus’s put_in_reg annotation uses convto to reinterpret the polymorphic bitwidth of the argument as 64 bits:

1 (spec (put_in_reg arg)
2 (provide (= result (convto 64 arg))))
3 (decl put_in_reg (Value) Reg)

3.1.3. Type inference and monomorphization

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, Crocus must transform its high-level intermediate representation, which allows polymorphic bitvector types, into several SMT formulas, each over a different set of bitvector widths. Crocus uses two passes of type inference to fully re-solve all bitvector widths and to monomorphize from each rule into a set of SMT formulas, each with a specific concrete type.

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 instantiations is:

\{ (t, t) \rightarrow t \mid t \in \{ i8, i16, i32, i64 \}\}

For a more complicated term that involves modifying the Cranelift IR width of the input and output, we consider a wider

Figure 1: Crocus’s annotation language, which combines SMT-LIB constructs with Crocus-specific constructs (e.g., convto and widthof), conveniences such as switch, and custom encodings such as cls (count leading sign). Figure 2 provides the semantics for key terms in this annotation language.

condition that enforces that its branches are exhaustive, which has helped surface faulty annotations.

Crocus provides constructs for introspecting on and modifying bitvector widths. widthof returns the width—often only known directly at solving time (§3.2)—of a given bitvector value. convto changes the width of its bitvector argument based on the first, integer argument.

Crocus also provides higher-level versions of SMT-LIB constructs. For example, SMT-LIB rotates must have statically-provided widths; Crocus instead offers symbolic rotates, which it implements with shift and bitvector logic instructions. Finally, Crocus 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 (§4.3.3), (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
We elide the second component of the tuple when the assertions are solely the union of the assertions on the expression's subterms—that is, we consider multiple output types per argument type: is the typing context, \( \Gamma \)

\[
\frac{\Gamma \vdash e_1 : \text{Int} \leadsto N \quad \Gamma \vdash e_2 : \text{bv}(N) \leadsto e'_2}{\Gamma \vdash (\text{convto } e_1 e_2) : \text{bv}(N) \leadsto e'_2}
\]

**CONVTO-WIDE**

\[
\frac{\Gamma \vdash e_1 : \text{Int} \leadsto N \quad \Gamma \vdash e_2 : \text{bv}(M) \leadsto e'_2 \quad N > M \quad e'_1 = (\text{declare-fun fresh } (_{-\text{BitVec}} N-M)) ; \text{fresh}} {\Gamma \vdash (\text{convto } e_1 e_2) : \text{bv}(N) \leadsto (\text{concat } e'_1 e'_2)}
\]

**CONC**

\[
\frac{\Gamma \vdash e_1 : \text{bv}(N_1) \leadsto e'_1 \ldots \Gamma \vdash e_n : \text{bv}(N_n) \leadsto e'_n}{\Gamma \vdash (\text{concat } e_1 \ldots e_n) : \text{bv}(\Sigma N_1 \ldots N_n) \leadsto (\text{concat } e'_1 (\text{concat } e'_2 (\ldots (\text{concat } e'_n))))}
\]

**WIDTH-OF**

\[
\frac{\Gamma \vdash e : \text{bv}(N) \leadsto e'} {\Gamma \vdash (\text{widthof } e) : \text{Int} \leadsto N}
\]

**INT2BV**

\[
\frac{\Gamma \vdash e : \text{Int} \leadsto e'} {\Gamma \vdash (\text{int2bv } N \ e) : \text{bv}(N) \leadsto (\text{nat2bv } N \ e')}
\]

**ZEROEXT**

\[
\frac{\Gamma \vdash e_1 : \text{Int} \leadsto (M, A_{e_1}) \quad \Gamma \vdash e_2 : \text{bv}(N) \leadsto (e'_2, A_{e_2})}{\Gamma \vdash (\text{zeroext } e_1 e_2) : \text{bv}(M) \leadsto (((\_ \text{ zero\_extend } (M-N)) e'_2), A_{e_1} \cup A_{e_2} \cup \{N < M\})}
\]

**SIGNEXT**

\[
\frac{\Gamma \vdash e_1 : \text{Int} \leadsto (M, A_{e_1}) \quad \Gamma \vdash e_2 : \text{bv}(N) \leadsto (e'_2, A_{e_2})}{\Gamma \vdash (\text{signext } e_1 e_2) : \text{bv}(M) \leadsto (((\_ \text{ sign\_extend } (M-N)) e'_2), A_{e_1} \cup A_{e_2} \cup \{N < M\})}
\]

**ROTL**

\[
\frac{\Gamma \vdash e_1 : \text{bv}(N) \leadsto e'_1 \quad \Gamma \vdash e_2 : \text{bv}(N) \leadsto e'_2 \quad e'_3 = (\text{bvrem } e'_2 (\text{nat2bv } N \ N))}{\Gamma \vdash (\text{rotr } e_1 e_2) : \text{bv}(N) \leadsto (\text{bvor } (\text{bvlshr } e'_1 e'_3) (\text{bvsub } (\text{nat2bv } N \ N) e'_3))}
\]

**ROTR**

\[
\frac{\Gamma \vdash e_1 : \text{bv}(N) \leadsto e'_1 \quad \Gamma \vdash e_2 : \text{bv}(N) \leadsto e'_2 \quad e'_3 = (\text{bvrem } e'_2 (\text{nat2bv } N \ N))}{\Gamma \vdash (\text{rotr } e_1 e_2) : \text{bv}(N) \leadsto (\text{bvor } (\text{bvlshr } e'_1 e'_3) (\text{bvsub } (\text{nat2bv } N \ N) e'_3))}
\]

**SWITCH**

\[
\frac{\Gamma \vdash m_1 : t_1 \leadsto <m'_1, A_{m_1}> \ldots \Gamma \vdash m_n : t_1 \leadsto <m'_n, A_{m_n}> \quad \Gamma \vdash e_1 : t_2 \leadsto <e'_1, A_{e_1}> \ldots \Gamma \vdash e_n : t_2 \leadsto <e'_n, A_{e_n}>}{\Gamma \vdash (\text{switch } c (m_1 e_1) \ldots (m_n e_n)) : t_2 \leadsto ((\text{ite } (= c \ m'_1) e'_1 (\text{ite } (= c \ m'_2) e'_2 (\ldots (\text{ite } e'_n))))), A_{e_1} \cup A_{m_1} \ldots A_{m_n} \cup A_{e_2} \ldots A_{e_n} \cup \{\text{bvor } (= c \ m'_1) (\text{bvor } (= c \ m'_2) \ldots (= c \ m'_n))\})}
\]

\[
\begin{array}{c}
e_1, e'_1, m_1 \in \text{expr} \\
e'_1, e'_2, m'_1 \in \text{QF_BVUNTS} \\
M, N \in \text{N}
\end{array}
\]

Figure 2: Typing and elaboration judgements for key terms in Crocus’s annotation language. Judgements take the form \( \Gamma \vdash e : t \rightarrow \langle e', A \rangle \), where \( \Gamma \) is the typing context, \( e \) is an expression, \( \langle e' \rangle \) in the grammar given in Figure 1), \( t \) is a type, \( \langle e' \rangle \) is the expression’s translation into the SMT-LIB theories of bitvectors (QF_BV) and integers (INTS), and \( A \) is a set of additional assertions that we add to the verification conditions. We elide the second component of the tuple when the assertions are solely the union of the assertions on the expression’s subterms—that is, we write the shorthand judgement as \( \Gamma \vdash (f \ e_1 \ldots e_n) : t \rightarrow \langle e' \rangle \) in place of the full judgement \( \Gamma \vdash (f \ e_1 \ldots e_n) : t \rightarrow \langle e', A_{e_1} \cup A_{e_2} \ldots A_{e_n} \rangle \).

A set of instantiations. For example, for extending values, we consider multiple output types per argument type:

\[
\{(s) \rightarrow d \mid s, d \in \{i8, i16, i32, i64\}, d \geq s\}
\]

Crocus then runs the two passes of type inference and monomorphize for each type instantiation of a given rule. The first inference pass then produces an assignment of SMT types (int, boolean, bitvector) for each variable in a term or its specification given the specific type instantiation. The first pass is also able to resolve some bitvector-widths to concrete widths (e.g., bv32) using an implementation of unification-based type inference. However, in some cases (such as rules that use intermediate terms on the LHS), the first pass is unable to resolve all bitvector-widths. In that case, we run a second, solver-based type inference pass to iteratively resolve all pos-
fn monomorphize():
  for ty_instantiation in ty_instantiations:
    G ← ty_instantiation
    // Unification-based type inference
    G ← type_inference_pass_1(G)
    // Solver-based type inference
    if type_set = type_inference_pass_2(G):
      return type_set
    type_set ← resolve_unknown_tys(solver, ts)
    if type_set.empty():
      return VerificationResult::InapplicableRule;
  return run_correctness_queries(type_set)

fn type_inference_pass_2(G):
  type_set ← initialize_with(G)
  if undetermined bitvector types ts:
    solver ← initialize_solver(G)
  type_set ← resolve_unknown_tys(solver, ts)
  return type_set

fn resolve_unknown_tys(solver, ts):
  // Solve for undetermined bitvector types
  match solver.check_sat():
  SAT ⇒
    new_types = resolved_types(solver.model())
  // Check whether another model with some distinct type is possible
  solver.assert(or_many(new_types.map(
    | (ty_var, concrete)| not(eq(ty_var, concrete)))))
  return [new_types]+resolve_unknown_tys(solver, ts)
  UNSAT ⇒ return []
  UNKNOWN ⇒ ERROR()

Figure 3: High-level algorithm for Crocus’s monomorphization and type inference, which produce a set of precisely-typed formulas for each potentially-polymorphic rule. type_inference_pass_1 has a standard unification-based implementation that we omit here.

Possible assignments of widths to bitvectors. Figure 3 provides high-level pseudocode for Crocus’s combined type inference algorithm.

First pass. The first pass Crocus runs is 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.

Crocus is not always able to resolve precise bitvector 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, Crocus finishes resolving all bitwidths in a second typing pass when necessary.

Second pass. During the second type inference pass, Crocus 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.

Most terms on the right-hand side of Cranefl’s ISLE rules operate on types modeling registers, instead of values in the intermediate representation. Cranefl’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. We iteratively call the SMT solver to check if there are multiple distinct type assignments that are possible for a given rule and type instantiation (lines 19-20 of Figure 3), similar to counter-example guided inductive synthesis [1].

3.2. Generating verification conditions

Once Crocus has run type inference and monomorphization—yielding one or more low-level, precisely typed rule representations—it lowers those representations to sets of SMT formulas that expresses equivalence of the right and left-hand sides of a lowering rule. While the left-hand side of a rule frequently has a narrow width, such as 32 bits, the right-hand side typically has the full register width of 64 bits. In discussion with Cranefl engineers, we learned that Cranefl’s intended invariant is that the low bits of a register corresponding to a given type must match the computed value, while the higher bits (outside the value’s type) are unspecified. Thus, Crocus’s correctness check performs an equality comparison by comparing any narrow values (typically on the LHS) with an extraction of the same number of low bits on register-width values (typically on the RHS).

At a high level, when Crocus performs a correctness check, 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-let clauses (see §4.4.4)—such that the rule never matches on this type instantiation.

To produce these 3 outcomes, Crocus uses (at least) two 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, Crocus 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, Crocus’s formula for a given rule combines the semantics and preconditions of Cranefl 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). Crocus combines semantics across term annotations via a recursive descent over the rule’s RHS and LHS, equating corresponding arguments and return values.
3.2.1. The first query: applicability

Let \( i_0 \ldots i_{n-1} \) 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 \texttt{provide} and \texttt{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:

\[
\exists \{i_0, \ldots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} | p^{LHS} \land R^{LHS} \land p^{RHS} \quad (1)
\]

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 \texttt{provide} expressions \( (p^{RHS}) \) in this initial query helps catch overly restrictive annotations. For instance, a vacuously false assertion in a \texttt{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. Crocus 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). Crocus 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 (§4.4.2).

3.2.2. The second query: equivalence

If the first query succeeds, Crocus constructs another SMT query to determine equivalence. Let \( \text{result}^{LHS} \) be the value returned by the outermost LHS term and \( \text{result}^{RHS} \) be the value returned by the outermost RHS term. A rule is correct if \textit{assuming} (1) the semantics of the LHS and RHS terms and (2) preconditions of the LHS \textit{implies} (1) the equivalence of the LHS and (possibly extracted bits from) the RHS and (2) preconditions on the RHS terms:

\[
\forall \{i_0, \ldots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} | (p^{LHS} \land R^{LHS} \land p^{RHS}) \Rightarrow (\text{result}^{LHS} = \text{result}^{RHS}) \land R^{RHS} \quad (2)
\]

To convert this statement to an SMT query, Crocus 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 Crocus which bits are demanded to produce the right verification conditions. For many rule and type instantiation pairings, the expression \( \text{result}^{LHS} \) (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 Crocus verifies aligns with Cranefit IR’s intended invariant: that the lower bits of the register are equivalent to the Cranefit IR semantics on the narrow width. We implement this condition in Crocus by adding an annotation on the \texttt{output_reg} term, which the ISLE preprocessor inserts as an automatic type conversion:

1. \texttt{(spec \{output_reg \ arg\})}
2. \texttt{(provide (= \ result \ \ (convto (widthof \ result) \ arg))))}
3. \texttt{(decl \ output_reg \ (Reg) \ InstOutput)}

The \texttt{convto} in this annotation narrows the bits of \texttt{Reg} in consideration to the bit demanded by the width of the \texttt{InstOutput} (which models the potentially narrow Cranefit IR type). In practice, this often produces an \texttt{extract} of the low bits of the RHS ISA term before comparing to the LHS IR term.

Optional custom verification conditions and assumptions. Some compiler transformations intentionally break strict equivalence. For example, Cranefit 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, \( \geq \text{vs} > \)). The relevant rule acts on terms that that produce the ISLE type \texttt{FlagsAndCC}, rather than a boolean value directly. Since the mathematical identity changes the values of both the flags and the condition code, Crocus reports a verification failure on this and similar rules.

Optionally, users can run Crocus with custom verification conditions instead of checking strict bitvector equality of the LHS and RHS (this may be required because Crocus currently reasons about rules in isolation). In this case, Crocus can encode the logic that flattens flags and a condition code into a boolean in order to prove that the \texttt{boolean} result of the comparison maintains equivalence. Users can also provide Crocus with additional assumptions on input values, which we use to encode cases where a rule would not match due to ISLE’s priority semantics. With addition assumptions \( A_n \) and custom verification conditions \( C_m \), the correctness statement becomes:
Figure 4: The cumulative distribution function of verification times for each rule run in isolation. Rules where some type instantiations time out are split into two separate tests (one that terminates and one that times out) for this figure.

\[
\forall \{i_0, \ldots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} \cup A_n

(P^{LHS} \land R^{LHS} \land p^{RHS}) \Rightarrow C_m \land R^{RHS}
\]  

(3)

3.3. Implementation and trust model

Crocus is implemented 15,825 lines\(^7\) of Rust as a fork of the Wasmtime codebase.\(^8\) We run Crocus queries as a Rust test suite in continuous integration on our Wasmtime fork. Crocus is designed to be useful to compiler engineers who are not experts in verification tooling; Crocus lifts counterexamples from the SMT model back into ISLE syntax to make debugging easier. Crocus can also test rules against specific concrete inputs (i.e., run as an interpreter), allowing developers to test their annotations against their expectations.

Caveats and the trusted code base. Crocus 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).\(^9\) Crocus 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 Crocus did not require condition codes to fall into a valid range. Finally, Crocus currently reasons about each rule individually. Support for verifying properties over multiple rules (e.g., reasoning about rule priorities) is future work.

4. Evaluation

This section answers the following evaluation questions:

Q1 Can Crocus 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 Crocus verified?

Q3 Can Crocus reproduce prior, known Cranelift bugs?

Q4 Can Crocus help identify and fix new bugs?

We answer Q1 by verifying a natural subset of rules, those necessary to compile (to aarch64) integer computations in the latest major release of WebAssembly (“1.0” [66]). Crocus has preliminary support for some x86-64 instructions (see Section 4.3.1). Section 4.2 addresses Q2—we find that the rules we verify comprise 19.8% 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 Crocus produces in Section 4.3.

Finally, in Section 4.4 we address Q4, outlining 3 new faults (2 patched) that Crocus discovered, and 1 compiler mid-end bug that Crocus 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 Crocus.

4.1. Is Crocus applicable to real rules?

We use Crocus to verify the instruction-lowering rules for all integer operations\(^10\) 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 serious 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 6.

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 96 distinct rules with 388 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., \(i_8, i_{16}, u/i_{32},\) and \(u_{164}\)), though we note that WebAssembly 1.0 only supports 32-bit and 64-bit numbers.

\(^7\) Plus 26,465 lines for our auto-generated annotation language parser.

\(^8\) Forked at commit 9556cb1.

\(^9\) 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.

\(^10\) All operation defined under section “4.3.2 Integer Operations” of the WebAssembly Specification Release, 1.0
Table 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.

<table>
<thead>
<tr>
<th>Rules</th>
<th>Total</th>
<th>Success</th>
<th>Timeout</th>
<th>Inapplicable</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>388</td>
<td>217</td>
<td>28</td>
<td>N/A</td>
<td>4 (0)</td>
</tr>
<tr>
<td>Type Instantiations</td>
<td>84 (all types) / 93 (any type)</td>
<td>10 (any type) / 1 (all types)</td>
<td>139</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>84 (all types) / 93 (any type)</td>
<td>10 (any type) / 1 (all types)</td>
<td>N/A</td>
<td>2 (0)</td>
</tr>
</tbody>
</table>

rustc_codegen_cranelift, an alternative backend for the Rust language, uses the narrower types Crocus supports [58, 10].

Table 1 shows the verification results for all 388 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. 360 of the 388 invocations complete, in sum, within 5 minutes on a laptop.\textsuperscript{11} The 10 rules that timeout on some type instantiations contain multiplication, division, remainder, and \texttt{popcnt} operations on bitvectors, which are difficult for SMT solvers to reason about for wider widths [40].\textsuperscript{12} Each of these rules fails with a counterexample within 10 seconds if we inject a flaw in the rule logic. Figure 4 presents the cumulative distribution function of verification times for each rule run in isolation as a Rust unit test (including the time for Rust test initialization).

4.2. What proportion of invoked rules has Crocus verified?

We instrument Cranelift to determine, on various targets, what proportion of invoked ISLE rules Crocus has verified. For the WebAssembly reference test suite, Crocus verifies 19.8\% (50/253) 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 15.8\% (24/152) of the unique ISLE rules used during compilation. These numbers will grow as we enhance Crocus to additional memory operations and floating point (§6).

4.3. Can Crocus detect known bugs?

To answer our third question, we use Crocus 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 Crocus.

4.3.1. x86–64 addressing mode CVE (9.9/10 severity)

In under one second on a laptop, Crocus detects a 2023 CVE in x86–64 instruction lowering that permitted a WebAssembly sandbox escape (§1) [25]. The reproduction requires 13 new annotations to support terms in the x86–64 backend, which we had not previously covered (§4.1).

The bug appeared in this ISLE rule:\textsuperscript{13}

\begin{verbatim}
1 (rule
2   (amode_add (Amode.ImmReg off base)
3     (uextend (ishl x (iconst shft))))
4   (if (u32_lteq (u8_as_u32 shift) 3))
5     (Amode.ImmRegRegShift off base
6     (extend_reg x I64 (Extend.Zero) shift))

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 (§1) 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, \texttt{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, \texttt{(uextend \ldots)}, is a Cranelift IR value that is a zero-extended (\texttt{uextend}) shift operation \texttt{(ishl)} with a statically known, constant shift amount \texttt{shft} (conceptually \texttt{i64.extend_i32_u (i32.shl <x> (i32.const < shft>>>>))). The rule's if clause checks that the shift amount, \texttt{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 \texttt{x} to be shifted is zero-extended, shifted by the static \texttt{shft} amount, and added to the other components of the computed address \texttt{(base + off)}.

Crocus provides the following counterexample:\textsuperscript{14}
\begin{verbatim}
1 (amode_add
2   (Amode.ImmReg
3     [off]|x30c04100| [base]|x0000000000000000)|
4     (uextend
5     (ishl [x]|x0d000920| (iconst |shft|\#x02)))) ->
6   (Amode.ImmRegRegShift
7     [off]|x30c04100
8     (gpr_new [base]|x0000000000000000)
9     (extend_to_gpr [x]|x0d000920] I64 Extend.Zero)
10   |shft|\#x02)
\end{verbatim}

\textsuperscript{11} We run experiments on a MacBook Pro Apple M2 Max, 12-core CPU, 32GB RAM, macOS 13.2.1.
\textsuperscript{12} Timed out after 6 hours.

\textsuperscript{13} Lightly edited for brevity
\textsuperscript{14} Lightly edited for brevity.
In this counterexample, the 32-bit value \( x \), \( \#x00000000_70c0_6580 \), 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 Crocus.

### 4.3.2. aarch64 unsigned divide CVE (moderate severity)

Crocus 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 \( \text{imm} \) on the RHS. \( \text{imm} \) 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 \( \text{imm} \) was:

\[
\text{(decl } \text{imm } \text{(Type u64) Reg})
\]

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: \( \text{imm} \) 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 \( \text{imm} \) takes in an argument for the type of extension, and the rules for \( udiv \) and \( sdiv \) now successfully verify.\(^{13}\)

### 4.3.3. aarch64 count-leading-sign bug

Crocus reproduces a pre-existing bug in the ISLE aarch64 lowering rule for \( \text{cls} \), the instruction that counts the number of leading sign bits in a value (excluding the sign bit itself). The rule for narrow \( \text{cls} \) 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:

\[
\text{(rule}
\begin{align*}
1 & \quad \text{(lower } \text{(has_type I8 } \text{(cls } x\text{) })}) \\
2 & \quad \text{(a64_sub_imm I32 } \text{(a64_cls I32 } \text{(zext32 } x\text{) } 24) \text{))}
\end{align*}
\]

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

\[
\begin{align*}
1 & \quad \text{(lower } \text{(has_type I8 } \text{(cls } [x\text{] } \#b11111100))}) \quad \Rightarrow \\
2 & \quad \text{(output_reg} \\
3 & \quad \text{(a64_sub_imm I32} \\
4 & \quad \text{(a64_cls I32 } \text{(zext32 } [x\text{] } \#b11111100)) 24)) \\
5 & \quad \text{(a64_add_imm ty x y)})
\end{align*}
\]

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 Crocus verifies it successfully.

### 4.4. Can Crocus find new bugs?

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

#### 4.4.1. Another addressing mode bug

Crocus discovered a new correctness bug in an \( x86-64 \) addressing mode rule related to the one discussed in §4.3.1 (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 §4.3.1)—the same bug could surface on a direct load of a 32-bit address. Crocus discovered 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 Crocus’s result.

#### 4.4.2. Flawed negated constant rules

Crocus 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 \( \text{imm12} \) 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 Crocus discovered:

\[
\text{(rule}
\begin{align*}
1 & \quad \text{(lower } \text{(has_type (fits_in_64 ty)} \\
2 & \quad \text{(isub x (imm12_from_negated_value y))}) \\
3 & \quad \text{(a64_add_imm ty x y)})
\end{align*}
\]

\(^{13}\)Though as noted previously, Crocus times out on some wide divisions.
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, Crocus 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`:

```rust
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.

Crocus 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.

### 4.4.3. Imprecise semantics for constants in Cranelift IR

Crocus 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, Crocus 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.

### 4.4.4. A mid-end root cause analysis

While we designed Crocus 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, Crocus 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`:\(^{16}\)

\[^{16}\text{Lightly edited for clarity and brevity.}\]

```rust
<table>
<thead>
<tr>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(simplify (bor (band x (iconst y)) (iconst z)))</td>
</tr>
<tr>
<td>(if (u64_eq zk (u64_not y)))</td>
</tr>
<tr>
<td>(bor x z)</td>
</tr>
</tbody>
</table>
```

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 Crocus analyze this rule (e.g., added annotations for mid-end terms) in under two hours. Crocus produced the following counterexample:17

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

Crocus 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)`. Crocus’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 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 Crocus’s tight integration with ISLE’s own program representation: Crocus was able to root-cause this bug because it must reason about core ISLE semantics.

### 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.

\[^{17}\text{Example truncated to 1 bit for brevity.}\]
Other works use solver-backed methods to verify portions of industrial compilers. Most closely related to Crocus, Alive [52] verifies LLVM [46] peephole optimization rules written in a DSL. Alive’s main challenge is undefined behavior; in contrast, Crocus need not reason about undefined behavior, but must instead reconcile IR and ISA types. Crocus also must contend with a language for instruction selection where engineers can (and do) build new operators within the instruction lowering language itself, whereas Alive reasons about a subset of LLVM’s IR plus a small set of built-in predicates (e.g., isPowerOf2) for conditioning on program values. 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 a Just-In-Time (JIT) compiler from BPF to native code, in a restricted setting where instruction selection entails simple “macro expansion” of one instruction at a time. While Jitterbug requires a substantially smaller TCB than Crocus, Crocus considers more complex backend instruction selection, with potentially M-to-N instead of just 1-to-N lowerings. For example, Cranelift’s Arm64 backend is around 3,600 lines of ISLE and 10,000 lines of Rust compared to Jitterbug’s Arm64 JIT’s 653 lines in their DSL (or 1,025 lines of equivalent C). While Crocus does not verify the entirety of the backend, our setting within the context of the larger, frequently-changing Cranelift project motivates our distinct design decisions.

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. 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 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 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 instruc-

\[\text{tions; using an existing, handwritten rule set for the rest.} \] [26] uses a solver to generate high-coverage selection 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 Crocus 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 Crocus to exploit these existing semantic models.

6. Future work

Crocus annotations are currently trusted. We can address this issue by deriving certain annotation from existing formal models. For example, Crocus 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 Crocus to reason about floating point, more operations with side effects, some SIMD vector instructions, and wider integers. Crocus already incorporates annotations for some 128-bit vector instructions, because the implementation of popcnt on aarch64 uses them. Crocus can also be extended to automatically reason about rule priorities and to cover other backends and the mid-end optimizer.

We are working to upstream Crocus 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 Crocus and as we build on Crocus 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 fundamentally places a high burden on the compiler that produces the final executable in a machine-specific ISA. Crocus is a tool for verifying instruction-lowering rules in one such safety-critical compiler: the Cranelift code generator. Crocus’s key selling point is its modularity—Crocus’s annotation language allows concise semantics of individual terms to be added alongside definitions in ISLE, a feature-rich instruction-lowering DSL. With Crocus, compiler developers can reduce the risk of security-critical vulnerabilities in instruction lowering logic.

---

18 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.
8. Acknowledgements

Thank you to Jamey Sharp, Nick Fitzgerald, Trevor Elliott, Björn Roy Baron, Till Schneideret, the members of the Bytecode Alliance, and participants in the Foundations of WebAssembly Dagstuhl seminar for their feedback. The first author was partially supported by an NSF GRFP under DGE-1650441. The second author was in part supported by the AAUW Selected Professions Fellowship. This research was also supported by NSF grants 2124045 and CNS-2120642.

References


[34] Anthony Fox, Magnus O Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. 2017. URL: https://doi.org/10.1145/3018610.3018621.


[37] Rikard Hjort. Formally verifying WebAssembly with KWasm, 2020. URL: https://github.com/sergei/apirfc/bitstreams/a06be182-a1e2-46ce-9d3f-c7ifa5d2ca2b/content.