Scaling Instruction-Selection Verification against Authoritative ISA Semantics

MICHAEL MCLOUGHLIN, Carnegie Mellon University, USA
ASHLEY SHENG, Wellesley College, USA
CHRIS FALLIN, F5, USA
BRYAN PARNO, Carnegie Mellon University, USA
FRASER BROWN, Carnegie Mellon University, USA
ALEXA VANHATTUM, Wellesley College, USA

Secure, performant execution of untrusted code—as promised by WebAssembly (Wasm)—requires correct compilation to native code that enforces a sandbox. Errors in instruction selection can undermine the sandbox's guarantees, but prior verification work struggles to scale to the complexity of realistic industrial compilers.

We present Arrival, an instruction-selection verifier for the Cranelift production Wasm-to-native compiler. Arrival enables end-to-end, high-assurance verification while reducing developer effort. Arrival **(1)** automatically reasons about *chains* of instruction-selection rules, thereby reducing the need for developer-supplied intermediate specifications, **(2)** introduces a lightweight, efficient method for reasoning about stateful instruction-selection rules, and **(3)** automatically derives high-assurance machine code specifications.

Our work verifies nearly all AArch64 instruction-selection rules reachable from Wasm core. Furthermore, Arrival reduces the developer effort required: 60% of all specifications benefit from our automation, thereby requiring 2.6× fewer hand-written specifications than prior approaches. Arrival finds new bugs in Cranelift's instruction selection, and it is viable for integration into production workflows.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal software verification**.

Additional Key Words and Phrases: Compiler Verification, Instruction Selection, WebAssembly, ISA Semantics

ACM Reference Format:
Michael McLoughlin, Ashley Sheng, Chris Fallin, Bryan Parno, Fraser Brown, and Alexa VanHattum. 2025. Scaling Instruction-Selection Verification against Authoritative ISA Semantics. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 418 (October 2025), 27 pages. https://doi.org/10.1145/3764383

1 Introduction

Web browsers, OS kernels, cloud services, and other pillars of the modern computing world contain compilers—and typically rely on compiler correctness for their own security guarantees. These compiler-embedding applications demand performance that can only be achieved with native code, and so, for security reasons, isolate untrusted code in a (compiler-enforced) sandbox [64]. WebAssembly (Wasm) [29] is a popular sandboxing mechanism, as it provides lightweight software isolation between different Wasm modules by design. However, Wasm's theoretical guarantees (and thus the guarantees of the embedding applications) rely in practice on a bug-free compiler.

Authors' Contact Information: Michael McLoughlin, Carnegie Mellon University, Pittsburgh, PA, USA, mcloughlin@cmu.edu; Ashley Sheng, Wellesley College, Wellesley, MA, USA, as126@wellesley.edu; Chris Fallin, F5, San Jose, CA, USA, chris@cfallin.org; Bryan Parno, Carnegie Mellon University, Pittsburgh, PA, USA, parno@cmu.edu; Fraser Brown, Carnegie Mellon University, Pittsburgh, PA, USA, fraserb@andrew.cmu.edu; Alexa VanHattum, Wellesley College, Wellesley, MA, USA, av111@wellesley.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART418
https://doi.org/10.1145/3764383

Formal verification can prove the absence of compiler bugs, but despite CompCert [41] and CakeML [37] demonstrating clean-slate, fully-verified compilers over a decade ago, most widely deployed production compilers are unverified. Industrial compilers are huge feats of engineering, including many thousands of lines of code and hundreds of optimizations on diverse representations. Hence, they cannot be re-written for clean-slate verification, nor verified automatically with existing tools. Correct Wasm compilation requires new tools that are practical in the context of realistic industrial compilers.

The Cranelift compiler [49] is one such tricky-to-verify (yet security-critical) industrial compiler backend. Cranelift serves as a faster replacement for LLVM in the Rust compiler and, most notably, backs the Wasm-to-native compiler in Wasmtime [51], a leading Wasm engine. Some of Wasmtime's worst vulnerabilities—e.g., the Wasm sandbox escape in CVE-2023-26489 [65]—resulted from errors in the Cranelift compiler's *instruction selection* pass. Instruction selection maps intermediate representation (IR) terms to equivalent ISA instructions, while also applying a complex suite of target-specific peephole optimizations. Cranelift's ISLE domain-specific language expresses instruction selectors as the composition of multiple intermediate rules; instruction selection is essentially a traversal from IR to ISA nodes in a selection graph, where edges represent ISLE rule applications and nodes are intermediate steps toward the final instruction choice (Figure 1). Cranelift's instruction selector has deep graphs, where many paths visit ten or more rules, and the total number of possible paths through the selection graph is huge.

Prior work struggles to scale to industrial instruction selectors like Cranelift. The first challenge is reaching end-to-end coverage of the selection graph. Existing verification approaches either force engineers to specify and verify every single edge [63], or confine themselves to relatively shallow graphs [46]. Second, high coverage requires the ability to specify the behavior of all selection graph nodes, including *stateful* operations (how instructions act on memory, set implicit flags, and raise possible exceptions). Prior work was either incapable of modeling state [63], or used complex state models that can hurt verification performance and hinder usability. The third challenge is that verification requires an accurate specification of the many possible output machine instructions. Defining instruction set semantics has typically been done by hand [13, 14, 46], which is not only tedious but also error prone. Lastly, industrial projects raise practical considerations: verification tools must be accessible to compiler engineers, and responsive enough for continuous integration.

To resolve these challenges, we present Arrival, a verifier that enables end-to-end, high-assurance instruction-selection verification in the industrial Cranelift compiler. Arrival achieves broader, more correct verification coverage with reduced developer effort.

To address the first challenge—scaling to end-to-end coverage of the instruction selector—Arrival uses a new *rule chaining* technique that analyzes the composed effect of a chain of rules (i.e., a path through the graph). Verifying every rule one-by-one demands too much of the developer, while the naive approach of enumerating every possible path through the instruction selector is computationally infeasible. Instead, rule chaining allows developers to specify the boundaries of the graph, and a few points in-between, and Arrival automates the rest. This makes it possible to divide a larger, intractable verification problem into a small number of components—each of which is tractable to automatically verify with rule chaining, and requires less of the developer.

To address the second challenge—modeling state—Arrival relies on one key insight: just a handful of variables suffices to model state in the instruction selection setting. Arrival provides language support for these custom state variable declarations, and specifications for how instructions modify them; these features are enough to efficiently verify stateful Cranelift to AArch64 instruction selection. By using such a minimal state representation, Arrival avoids the punishing solver times associated with modeling large addressable memories in their full fidelity.

To address the third challenge–verifying against precise and accurate ISA semantics—Arrival derives succinct verification specifications from vendor-provided, machine-readable ISA specifications. Prior projects were unable to replace hand-written instruction specifications with vendor-provided ones for a few reasons. First, vendor-provided semantics are verbose—with e.g., irrelevant microarchitectural detail—and are not directly equivalent to most compilers' ISA intermediate representations semantics. Second, while prior work [39] has derived succinct semantics for *concrete* machine instructions, compiler verification demands reasoning about *partially symbolic* instructions—where registers, immediate values, and more are not determined. We observe that compiler ISA representation semantics can be described by sets of symbolic instruction encodings, and build a pipeline that produces succinct specifications for them.

We evaluate Arrival's ability to verify Cranelift instruction selection for Wasm compilation on the AArch64 backend, since Arm provides authoritative ISA specs. With limited exceptions, our work verifies AArch64 instruction selection rules reachable from WebAssembly core [1] opcodes. We find that Arrival's coverage surpasses that of prior work. Arrival verifies memory and floating-point instructions that prior work [63] could not handle. Arrival's deeper coverage revealed a miscompilation of the `sdiv` instruction that prior work had erroneously verified, a discovery made possible via analysis of a chain of 12 ISLE rules, trap modeling, and accurate ISA flag specifications.

In sum, we make the following contributions:

- Arrival introduces rule chaining, an approach for automatically reasoning over multiple instruction mapping rules, thereby requiring 2.6× fewer developer-written specifications than prior work.
- Arrival introduces a novel lightweight mechanism for state modeling, which is performant and sufficient for instruction selection.
- Arrival is the first instance of compiler verification against auto-generated semantics from authoritative, vendor-provided hardware specifications. By automatically deriving 93% of its machine code specifications from ISA semantics, Arrival both reduces developer burden and raises assurance.
- Our evaluation demonstrates that Arrival can verify nearly all instructions required to compile WebAssembly 1.0 on AArch64, including memory operations, and integer and floating point numerics. Arrival proves the absence of miscompilations in 10,807 unique paths through the instruction selector, and finds a new bug missed by prior work.
- Our evaluation also demonstrates that Arrival is a step towards verification-in-production. Arrival derives 60% of all specifications automatically—reducing developer burden—and within 11 minutes on commodity hardware, Arrival verifies 97.8% of the in-scope instruction lowering paths—making continuous integration possible.

2 Overview

2.1 Cranelift's Instruction Lowering and the ISLE Language

Cranelift handles the complexity of instruction selection with a custom domain-specific language, ISLE (Instruction Selection Lowering Expressions) [21]. ISLE programs replace hand-written instruction selectors, which are typically thousands of lines of unwieldy nested conditional code. ISLE expresses instruction lowering with *term-rewriting* rules that eventually map *terms* representing IR instructions to machine-instruction terms, often via a series of intermediate terms. ISLE's meta-compiler translates ISLE terms and rules into Rust functions and optimized decision trees, which are integrated into the Cranelift compiler backend.

Rules consist of a left-hand-side *pattern* match describing when a rule applies, and a right-hand-side result expression. ISLE lowers IR terms to machine code through a series—or *chain*—of several rule applications via intermediate steps, visualized in Figure 1. Consider the initial rules for lowering iabs (an IR node), Cranelift's integer absolute value instructions, to AArch64 machine code:

```
(rule 1 (lower (has_type (fits_in_32 ty) (iabs x)))
  (abs (OperandSize.Size32) (put_in_reg_sext32 x)))
(rule 2 (lower (has_type $I64 (iabs x)))
  (abs (OperandSize.Size64) x))
```

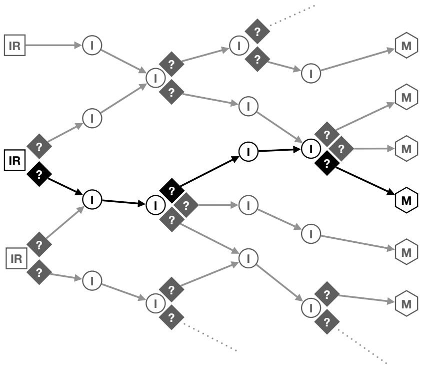


Fig. 1. Conceptual graph visualization of ISLE programs mapping IR terms IR via intermediate terms Ⓘ to machine code (ISA) terms Ⓜ. Rule match conditions use ◆ predicates, often via a bridge to external Rust.

The lower outermost term is the entry-point to instruction lowering. Terms in the rule's pattern (iabs, fits_in_32, ...) are *extractors* which conditionally match and bind additional values when successful. Extractors are often *predicate* terms (◆ nodes) that bridge to external Rust code to perform conditional pattern matching logic. The rules above match CLIF iabs instructions on bitwidths 32-bits-or-smaller, or exactly 64-bits, respectively. The integers 1 and 2 indicate the priority between the rules, i.e., the specific 64-bit case takes higher priority if it applies to a given CLIF subtree. In the narrow case, the input x is sign-extended to 64 bits. Both rules delegate to a helper, *intermediate* term abs (an Ⓘ node), which has its own rule:

```
(rule (abs size rn) (with_flags (cmp_imm size rn (u8_into_imm12 0))
                                (csneg (Cond.Gt) rn rn)))
```

This rule lowers the abs intermediate helper term into a comparison with zero (cmp_imm) followed by conditional negation (csneg). ISLE has a static type system that is used to enforce instruction lowering requirements, such as the constraint that an instruction that consumes flags must be directly after an instruction that produces flags. In this ISLE rule, this logic is encapsulated in the with_flags combinator, which expresses the transfer of implicit CPU flags between instructions.

Terms cmp_imm and csneg have one more level of rules to rewrite into Cranelift's lowest-level representation of AArch64 machine-instructions called MInst (Ⓜ nodes). For instance, for cmp_imm:

```
(rule (cmp_imm size src1 src2)
  (ProducesFlags.ProducesFlagsSideEffect
    (MInst.AluRRImm12 (ALUOp.SubS) size (writable_zero_reg) src1 src2)))
```

This rule implements comparison to an immediate value using a subtraction that sets flags (SubS opcode), and discards its result (writable_zero_reg destination). The ProducesFlagsSideEffect wrapper communicates the flag-producing behavior of the instruction to the with_flags combinator. Similarly, the csneg rule lowers to an MInst.CSNeg instruction. These rules complete the lowering phase within ISLE, after which the final compilation stages operate on produced MInst instructions.

Thus, ISLE lowers iabs instructions from IR term IR to machine code Ⓜ via a chain of rule applications. Notably, intermediate terms such as abs are never materialized in the resulting *MInst* IR or the eventual AArch64 instructions—they only exist during compilation, as helpers that enable compiler engineers to abstract over common logic.

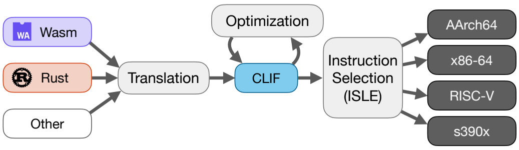


Fig. 2. Cranelift compilation phases.

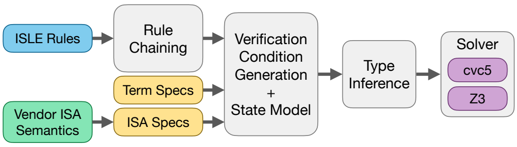


Fig. 3. Arrival's verification pipeline.

ISLE is strongly typed at the term level. Types representing the IR program being compiled (i.e., `Inst`, `Value`, and value types `Type`), and the machine instructions targeted (an `MInst`), are *statically known* and checked. However, ISLE rules are *polymorphic* with respect to the value types the input IR program might have (e.g., integer bitwidths).

2.2 ISLE's Specification Features

Arrival adopts and extends specification annotations recently added to the ISLE core language [20, 21, 63]. Compiler engineers can specify term pre- and post-conditions with `require` and `provide` constraints, expressed in a syntax modeled on SMTLIB. For example, the integer addition IR term `iadd` has the following annotation, expressing that the term's `result` (a dedicated keyword) is the bit-vector addition of its inputs:

```
(spec (iadd x y) (provide (= result (bvadd x y))))
```

To scale to high coverage of production instruction selection, Arrival extends ISLE's specification language and implements an entirely new verification backend. Arrival's rule chaining technique (§3) saves developers from adding pre- and post-conditions to every intermediate term, and Arrival automatically generates specifications for the lowest-level machine code `MInst` terms, from vendor-provided semantics (§5). To ensure high coverage, Arrival's extensions include support for stateful operations (§4) and floating-point. Arrival further extends ISLE's specification language with ergonomic features (struct types, macros, and enum pattern matching) which make Arrival expressive and productive at the scale of Cranelift's instruction selectors.

2.3 Arrival's Architecture

Arrival is built on top of Cranelift/Wasmtime. Figure 2 shows Cranelift's compilation phases, beginning with the CLIF intermediate representation produced by compiler front-ends such as Wasmtime and the Rust compiler. Cranelift's mid-end optimizes CLIF code, before lowering to machine code in target-specific instruction selectors (AArch64, x86, ...). Cranelift's instruction-selection backends, and some mid-end optimizations, are implemented as programs of ISLE rules.

Figure 3 shows how Arrival verifies ISLE instruction selection. Arrival's verification operates on *expansions*, each of which represents the result of combining (*chaining*) multiple ISLE rules together (often, for multiple distinct sub-terms). Arrival derives all possible expansions from an ISLE program, starting from the ISLE meta-compiler's low-level rule representation (called the *trie*). For each expansion, Arrival forms verification conditions (VCs) to express the semantic equivalence of the expansion's input and output terms. At this point expansions are still polymorphic—they typically apply to multiple possible compile-time value types. A final type inference pass expands terms into all possible *type instantiations* (monomorphizations) of input-program types. Finally, concrete VCs for every expansion and type instantiation are lowered to SMTLIB and dispatched to a backend SMT solver.

Arrival derives high-assurance instruction set semantics from vendor-provided specifications. Arrival's ISA specification pipeline is *independent* from the verifier: the verifier consumes the resulting ISA specifications in ISLE syntax. Using Arrival's Rust library, developers configure the ISLE machine instruction terms to generate specifications for the lowest-level machine code IR (`MInst`), including any symbolic parameters (e.g., not-yet-fixed immediate values or registers) that exist in the IR but not the final concrete instruction encoding. Arrival assembles each instruction configuration into symbolic instruction encodings, executes them according to the vendor specification to derive semantics, and finally compiles semantics into ISLE's spec language.

2.4 Bug Discovery with Arrival

This section shows Arrival's capabilities by walking through a subtle Cranelift bug uncovered by Arrival but *missed* by prior work. This discovery highlights Arrival's advances in industrial-scale instruction-selection verification: reasoning over multiple rule applications with rule chaining, precise and automated ISA specifications, and lightweight state modeling.

2.4.1 The Bug. The bug was a failed overflow check in lowering for 8- and 16-bit signed division on AArch64, causing code that should *trap* to instead be miscompiled to assembly that silently computes the wrong result. While not security-critical for Wasm (which does not support integer types narrower than 32 bits), the bug could affect other Cranelift-based compilers.

Cranelift IR (CLIF) semantics—inherited from Wasm—say that signed division (`sdiv`) must trap when the result is unrepresentable. Beyond division by zero, the more subtle edge case is that for n-bit values, the signed division $-2^{n-1}/-1$ overflows (since the integer maximum is $2^{n-1} - 1$). A bitwise logic flaw in `trap_if_div_overflow`—one of *twelve* rules chained in the lowering of `sdiv`—caused the attempted overflow check to fail for narrow bitwidths.

The `trap_if_div_overflow` helper term takes a type `ty` (an n-bit integer type), and two registers `x` and `y`. The rule intends to work by checking if `y` is -1, checking if `x` is the minimum representable value of type `ty`, and trapping if both are true. Concretely, it did so with the following ISLE lowering rule to machine instructions (some detailed elided with `...`):

```
1 (rule (trap_if_div_overflow ty x y) (let (
2     ;; Check if y is -1, set flags if so.
3     (y_is_neg1 ... (MInst.AluRRImm12 (ALUOp.AddS) ... y (u8_into_imm12 1)))
4     ;; Check if x is min_value, set flags if so.
5     (x_is_min_val ... (MInst.CCmpImm (size_from_ty ty) x (u8_into_uimm5 1) (nzcv ...) (Cond.Eq)))
6     ;; Consume flags from previous instructions and trap if both are set (i.e., on overflow)
7     (trap_if ... (MInst.TrapIf (cond_br_cond (Cond.Vs)) (trap_int_overflow))))
8     ;; Return the sequence of instructions above
9     ...))
```

The issue is line 5, which attempts to check whether the numerator `x` has minimum value using a `CCmpImm` instruction (conditional compare with immediate). Comparisons in AArch64 (as with most ISAs) are functionally a subtraction that sets flags. Here, the input `x` is compared to the constant 1. If `x` is the minimum representable value, subtracting 1 should overflow. The following instruction `TrapIf` (line 7) checks flags for the overflow condition (`Cond.Vs`) and traps if so.

However, the bug arises from the *operand size* of the conditional comparison, computed by the `size_from_ty` helper. `size_from_ty` returns `Size64` if its input is a 64-bit type, and `Size32` if its input is a 32-bit type *or smaller*. As a result, this line incorrectly checks for 32-bit overflow, when it should check for 8- or 16-bit overflow. Subtracting one from 32-bit -128 or from 32-bit -32,768 won't overflow; it will simply yield -129 and -32,769! Therefore, the buggy rule *will not* properly set flags and *will not* trap if the result of the division is unrepresentable.

2.4.2 How Arrival finds the bug. Arrival uncovered the bug in under 9 seconds in attempting to verify `sdiv` against CLIF semantics, expressed in the specification:

```
(spec (sdiv x y) (modifies clif_trap)
  (provide ; Division by zero causes trap.
           (if (bv_is_zero! y) clif_trap
           ; Integer overflow causes trap.
           (if (and (= x (bv_s_int_min! (widthof x)))
                    (bv_is_zero! (bvnot y))) clif_trap
           ; Else, compute the result.
           (and (not clif_trap) (= result (bvsdiv x y)))))))
```

This specification uses Arrival's state modeling (§4) to express which edge cases should cause a trap (`clif_trap`), and otherwise specify that the `result` is signed division of the inputs (`bvsdiv`). The `sdiv` lowering involves 12 ISLE rules, including the buggy `trap_if_div_overflow`. Arrival applies rule chaining (§3) to *expand* those rules into the resulting end-to-end lowering of `sdiv` IR to *nine* inter-dependent machine instructions. Arrival forms verification conditions to check whether the ISA semantics (derived from vendor specifications for AArch64—§5) do correctly refine CLIF semantics. Arrival's internal type-inference solver determines which bit-widths the rule chain could apply to, leading to *type instantiations* for 8-, 16- and 32-bit cases. For each instantiation, Arrival formulates SMT queries to check internal consistency of the specifications, and to check the verification conditions. In this case, the narrow 8- and 16-bit cases fail, and Arrival reports a counter-examples exhibiting the semantics mismatch.

This bug was *missed* by prior work, but found due to Arrival's three core contributions.

The flawed `sdiv` rule resided in a many-thousand line file, relied on 12 ISLE rules, and had a buggy helper rule that lived in a *separate* many-thousand line file. With *rule chaining*, Arrival automatically constructs an *expansion* that describes the entire (buggy) path from top-level CLIF semantics (`sdiv`) to ISA instructions. Therefore, Arrival minimizes the specifications the developer has to provide in order to find the bug in this intricate combination.

Arrival's *lightweight state modeling* enables both the description of CLIF trap semantics expected in `sdiv` (side effects), and the transfer of processor flags between ISA instructions (implicit state). Arrival models state effects with just a few variables for IR and ISA, and thus *efficiently* finds bugs in stateful instruction selection.

The bug involved seven machine instructions from five *instruction families*: `AluRRImm12` addition setting flags, `CCmpImm` conditional compare, `Extend` for sign extension, `AluRRR` for division, and the `TrapIf` Cranelift-internal pseudo-instruction. Moreover, it manifested in low-level details of these instructions' behavior—setting and conditionally acting on flags. By *automatically deriving ISA specifications* from vendor-provided, authoritative semantics, Arrival provides the fidelity required to catch the `sdiv` bug (and those like it), as well as the ISA coverage needed for industrial-scale compilers.

While prior work on the Crocus tool [63] claimed to have verified the `sdiv` instruction, Crocus in fact missed this bug. First, Crocus specs can only express purely functional semantics; stateful trapping behavior was out of scope. Second, lacking our rule chaining technique, Crocus required that every intermediate rule be verified independently. Crocus seemingly side-stepped that daunting workload by settling for *non-end-to-end* verification. In particular, Crocus stopped short of verifying the internals of the buggy `trap_if_div_overflow`. Third, the hand-written trusted spec of `trap_if_div_overflow` did not account for the buggy behavior (while Arrival's spec derived from the vendor semantics *does*). In the next three sections, we detail how Arrival advances the state-of-the-art in automatically verifying industrial-scale instruction selection.

3 Rule Chaining

Achieving high-coverage verification of instruction selectors requires proving all paths from the IR to ISA instructions are semantics preserving. One of ISLE's strengths is in factoring instruction selection into multiple intermediate terms and rules. We saw this capability used to express `sdiv` lowering with a combination of 12 ISLE rules. Many lowering paths invoke even more. The simplest approach to verification—pursued by prior work [63]—is to specify every term and verify every rule one-by-one. However, this approach only gets you so far: the annotation burden for industrial instruction selectors is extremely high. Moreover, Cranelift (and other production compilers) change constantly; asking developers to change hundreds of annotations *in addition* to hundreds of lines of code is a non-starter. We need an approach that can scale with reasonable developer effort.

Arrival aims to minimize required annotations. Specifications are *unavoidable* at the boundaries of the instruction selector: we need semantics for the input IR, predicate terms that bridge to external Rust, and the target ISA (although Arrival derives these automatically—§5). Within ISLE, can we make specifications for intermediate terms optional? Arrival's rule chaining technique shows that yes, we can eliminate the need for almost all intermediate-term specifications.

Arrival's approach is to **(1)** compute the effect of applying (*chaining*) multiple rules together, resulting in an *expansion*, and **(2)** use a *rule chaining* algorithm that generates all feasible expansions within an ISLE program. The main challenge is *path explosion*: an unrestricted rule-chaining algorithm in realistic instruction selectors would produce a computationally infeasible—or even infinite—number of expansions. Arrival solves this by dividing the instruction-selection graph into components, within which the total number of expansions is tractable. Developers control this division via targeted specification and attribute annotations. Overall, Arrival's rule chaining minimizes the annotation effort required to verify instruction selection end-to-end from IR to ISA.

In this section, we give the intuition behind our rule chaining approach (§3.1), explain the challenges of applying it to production compilers (§3.2), and describe the algorithm in detail (§3.3).

3.1 Expansions: Chained Rule Applications

To see what it means to *chain* rules together, consider the initial lowering of bitwise-and (`band`) on AArch64:

```
(rule (lower (has_type (fits_in_64 ty) (band x y)))
      (alu_rs_imm_logic_commutative (ALUOp.And) ty x y))
```

This rule lowers a `band` instruction on 64-bit-or-smaller integers to an intermediate helper term, `alu_rs_imm_logic_commutative`. Other bitwise operators use the same helper; the helper has five rules of its own to further select the final machine code. One such rule optimizes the case where the right operand is a value shifted by a constant amount (`ishl` with `iconst`):

```
(rule (alu_rs_imm_logic_commutative op ty x (ishl y (iconst k)))
      (if-let amt (lshl_from_imm64 ty k))
      (alu_rrr_shift op ty x y amt))
```

When this rule matches, it produces the term `alu_rrr_shift` representing an AArch64 shifted-register instruction (which in turn has its own rules). To verify such a lowering, we could specify the behavior of the intermediate helper and confirm both rules are semantics preserving. Instead, we could compute the combined effect of both rules (as shown below), and verify the overall result.

```
(rule (lower (has_type (fits_in_64 ty) (band x (ishl y (iconst k)))))
      (if-let amt (lshl_from_imm64 ty k))
      (alu_rrr_shift (ALUOp.And) ty x y amt))
```

We call this operation *chaining* rules together, resulting in an *expansion*. Note how the helper term is no longer present: its pattern match conditions (`ishl`, `iconst`, ...) and rule body have been *inlined* into the expansion. We can repeat this process for all five of the helper term's rules, producing five possible expansions for the `band` lowering (during compilation, only one of the five expansions would apply to a given IR snippet based on which match predicates are true). We can also *iterate* on terms remaining in the expansion (e.g., the rules for alu_rrr_shift can also be chained), continuing until no more rules can be applied, or we otherwise choose to stop.

Note this operation presents a trade-off: when we apply rule chaining to a term, it no longer needs a specification (saving developer effort), but it creates a cross product in the number of total expansions to be processed (increasing verifier work). This trade-off is the core challenge of rule chaining, among others we'll see in the next section.

3.2 Challenges of Rule Chaining

In this section, we discuss how Arrival addresses the practical challenges of applying rule chaining: **(1)** path explosion, **(2)** rule priorities, and **(3)** expansion representation.

Path Explosion. At first, one might consider applying rule chaining *maximally*, generating expansions for every possible path through an entire ISLE instruction selector. Unfortunately, this is impractical for industrial instruction selectors.

First, the major ISLE backends in Cranelift have a theoretically unbounded number of paths due to *cyclic* rewrite rules. For example, consider this lower_fmla rule in the AArch64 backend:

```
(rule 5 (lower_fmla op (fneg x) y z size)
        (lower_fmla (neg_fmla op) x y z size))
```

This rule lowers a fused-multiply-add operation in a special case where one operand is negative, turning it into a fused-multiply-subtract (or vice versa). The rule could apply repeatedly to a nested series of fneg operators. Any fixed input program has finitely many negations, but from the instruction selector's perspective, there is no natural bound on the number of times it could apply.

Second, even after discarding cyclic terms, the number of paths through the selector is enormous: unrestricted rule chaining would yield over 487 *million* possible rule combinations in the AArch64 backend alone (many of which have multiple type instantiations). There are two main causes for this blowup: widely-used terms that contain just a few rules and terms that are less widely used but have many distinct rules. For example, the operand_size term contains two rules and appears pervasively, while the with_flags_chained term contains 26 rules.

Critically, these "explosive terms" are not the common case: 94% of terms in the AArch64 backend have four or fewer rules. Cyclic terms are also infrequent. This is encouraging: rule chaining *can* be practical, if we can mitigate the explosive effect of a small number of outliers.

Arrival addresses path explosion by making rule chaining configurable per-term. If a developer provides a specification for a term, it will not be chained. To avoid unbounded recursion, cyclic terms are always excluded. Otherwise, developers enable chaining by annotating the term with a (veri chain) attribute (Arrival also identifies single-rule terms that are essentially trivial wrappers that can be chained by default). When verifying an expansion, if a term has neither a specification nor an annotation, the developer will be prompted to add one of the two. Explicit opt-in for rule chaining provides a more predictable developer experience. This configurability allows most terms to be chained (reducing specification burden) while excluding explosive terms.

The rule chaining algorithm accounts for this configurability. Arrival eagerly applies rule chaining until it finds terms that *already have specifications*. Completed expansions are handed to the verifier. Next, terms with specifications become a new origin point for rule chaining expansion. The effect of this approach is to break the instruction selector graph into components. Terms at the boundaries have specs and rule chaining automatically handles reasoning within components.

Rule Priorities. Rules in ISLE have priorities to break ties when rules have overlapping match conditions. Most often, developers use priorities for efficiency rather than correctness: the higher-priority case produces faster code, but both would be correct. There are, however, certain cases where a lower priority rule is only correct *assuming* that a higher-priority rule did not match. The `operand_size` rules are a concise example:

```
(rule 1 (operand_size (fits_in_32 _)) (OperandSize.Size32))
(rule 0 (operand_size (fits_in_64 _)) (OperandSize.Size64))
```

Here, the first rule with higher priority 1 handles the 32-bit or smaller case. The second, lower-priority rule is for the 64-bit case; its correctness relies on only matching the 64-bit type, which is guaranteed because the higher-priority rule did not match. While this style is not common in ISLE, it does appear in important cases like the lowering of integer-comparison operations.

Arrival can automatically reason about prioritized rules (unlike prior work [63]). Logically, accounting for priority means that when chaining a rule, we must assert the negation of higher-priority rules' match conditions. The practical challenge is that expressing negated match conditions can significantly bloat verification conditions. Negated conditions from higher-priority rules bring additional variables into scope, potentially many when a rule is the lowest priority in a long list. To avoid unnecessary bloat when priority doesn't matter for correctness, Arrival's rule negation is *opt-in* through an a `(veri priority)` attribute. Without the attribute, Arrival effectively assumes a conservative over-approximation of ISLE semantics. If a developer omits the attribute where it's required, Arrival will catch the resulting verification failure and present a counter-example.

Expansion Representation. Our rule chaining algorithm requires an efficient implementation for rule composition. For ease of exposition in §3.1, we showed the result of chaining two rules in ISLE rule syntax. This is an over-simplification: expansions resulting from rule chaining cannot always be represented in ISLE's surface syntax. Instead, Arrival implements chaining on a lower-level representation of ISLE terms and rules. Arrival achieves this by building its expansion data structure upon the ISLE meta-compiler's *trie* representation, which we describe in the next section.

3.3 The Rule Chaining Algorithm

This section outlines Arrival's rule chaining algorithm. First, it describes the trie representation on which chaining operates; then, it describes chaining itself. Finally, it describes how Arrival prunes infeasible paths and accommodates rule priorities.

The trie representation. Arrival's rule chaining pass consumes the ISLE meta-compiler's *trie* intermediate representation, which the compiler produces after performing semantic analysis on the AST. While ISLE's AST has syntactic sugar, the trie is distilled to minimal primitives. Chaining is efficient at this level of abstraction. The trie representation for a given term contains:

- *Bindings*, which are expressions with identifiers. The two most relevant binding types for chaining are term *arguments* and *calls*—the results of invoking other terms.
- A *rule set* of all the rules for a term. Rules all reference the same set of bindings; each is composed of a list of *constraints* under which the rule matches, a priority, and a *result* binding to be used when the constraints match.

When the ISLE rules are compiled to the Rust instruction selector, terms become functions, bindings become variables, rules and their constraints describe a control-flow path through the function, and the result is the return value on that path.

Building Expansions by Rule Chaining. The rule chaining algorithm takes ISLE terms and rules, a designated *root term* (typically the `lower` entry point), and term attributes configuring which terms can be chained. The output is a list of all feasible expansions, each of which represents the result of applying some set of rules.

Rule chaining is a worklist algorithm. We begin with a single expansion that only contains a call to the root term. Given an expansion popped from the worklist, we look for any term calls. If

Arrival finds a call to a term that's eligible for chaining, Arrival forks the expansion once for each of the rules for the called term. In each fork, it will chain that rule into the expansion—that is, inline the rule's body in place of the call and collect the rule's constraints. Conceptually, each fork represents the effect of taking one possible control-flow path (a rule) through the called term. Forked expansions are pushed onto the worklist. When an expansion no longer has any term calls eligible for chaining, it is complete. The completed expansion is then checked for calls to terms that have specs: instead of forking, Arrival initiates a new rule chaining process rooted at that term.

Mechanically, the resulting expansion structure holds all the information Arrival needs to build the verification query for a given chain of rules. It contains a root term, a set of *all* the relevant bindings for rules included in the chain, *all* the constraints brought into scope by applied rules, and a result binding (i.e., the result when the entire chain of rules matches).

Pruning on feasibility. Some rules are statically incompatible with one another, and thus chaining a rule into an expansion may produce an expansion with constraints that are trivially unsatisfiable. Take for example the logic for the `smul_overflow` instruction, which utilizes the intermediate helper `(lower_extend_op ty (ArgumentExtension.Sext))` to emit a *signed* extension (`Sext`). One of the rules for this helper matches on 8-bit *unsigned* extension `(lower_extend_op $I8 (ArgumentExtension.Uext))`, and an attempt to chain this rule into the prior call would trivially fail to match on the extension type (signed vs. unsigned). To reduce path exploration, Arrival tries to statically evaluate constraints during expansion and prunes unsatisfiable paths.

Accounting for priorities. When Arrival chains a rule into an expansion, it may need to account for priority. Arrival considers all overlapping higher-priority rules that are annotated with the ISLE `priority` attribute. Limiting to explicitly annotated rules prevents verification condition bloat. For each selected higher-priority rule rule, Arrival expresses the negation of the rule's constraints in the expansion structure, importing bindings as necessary.

4 State Modeling: Pragmatic, Efficient Effects Representation

Scaling to high-coverage verification of industrial instruction selectors requires reasoning about all IR and ISA instruction behaviors. Beyond pure computations, we must be able to specify operations with side effects. IR semantics may define loads/stores to memory or raise a trap on illegal operations. Machine instructions can act on implicit processor state, such as memory and flag bits. Our goal is *lightweight* state reasoning that is pragmatic for compiler developers, and expressive enough to cover realistic instruction selectors. Furthermore, modeling state should impose low performance overhead on the verifier: the core trade-off of rule chaining (§3.1) means the more expansions the verifier can process, the less specification effort is required by developers.

Prior work either lacks the required expressivity or uses a state model that is too complex for our goals. The Crocus instruction-selection verifier [63] could only express purely functional specifications, leaving stateful verification out of scope. On the other hand, traditional approaches [42, 43, 46] to state modeling are heavyweight. Common approaches are big- or small-step operational semantics; however, it is costly to instantiate a full representation of machine state at every time step. Memory representation presents another challenge: large byte-addressable arrays bloat state and introduce a mix of theories that hurts underlying SMT solver performance. Arrival's approach is expressive while avoiding these pitfalls.

In this section, we motivate our lightweight approach (§4.1), define Arrival's language support for state management (§4.2), and demonstrate its use for AArch64 verification (§4.3).

4.1 Modeling Effects on State

Arrival's approach is to model *effects*, not complete state. Arrival takes advantage of the specifics of the instruction-selection domain to define a simpler model. While production instruction selectors are complex in the number of rewrites they define, each individual *m*-to-*n* IR-to-ISA mapping tends to have limited effects on state. For instance, instruction lowerings that interact with memory typically only involve one load or store. Similarly, a lowering either traps once or it does not. We find it is sufficient to capture these effects as global specification variables. To describe a possible load we do not need an entire representation of memory. Instead we can capture the *load parameters* on both IR and ISA—its address and size—as state variables, and verify that *both sides have the same effect*. Likewise, we can model the presence of a trap on each side with two boolean state variables. Since IR and ISA behaviors are diverse, developers use Arrival to define *custom* effect variables.

Given variables for possible state effects (loads, stores, traps, . . .), Arrival enables developers to specify **(1)** how terms modify effect variables, and **(2)** relations that must hold between effect variables after lowering. When a term has a state effect—for example a CLIF `load` or an AArch64 `ULoad32`—the specs for those terms declare that they *modify* the corresponding effect variable and provide post-conditions that specify the effect's parameters. State relations that must be true of any lowering, such as that both sides perform the same load, are expressed as post-conditions on the top-level `lower` entry point.

Since any term in an expansion can modify state variables, we also have to consider the cases where either *zero* or *more than one term* modifies a state variable. The zero case is common: for example, most instructions do not load from memory. Arrival handles this by allowing developers to specify a default value for an effect—for example, expressing that a load did not happen. More than one modification is an important case, though less common. For example, some lowerings (e.g., `sdiv`) can trap in more than one way (e.g., division by zero and overflow). Arrival handles this with *conditional modification* in the specs, which expresses the case where only one of multiple possible state modifications occurs (e.g., only one of the possible traps will actually happen).

4.2 Specification Language Support for State Effects

We extend the ISLE specification language with support for state effects.

Effect Variables. Developers declare a named state effect variable, with its type and default specification, as follows:

```
(state <name> (type <type>) (default <default>))
```

This declaration introduces a global variable `name` of the given type. This variable will be present in the verification conditions for *any* expansion, including those where no term modifies the variable (in which case the default clause provides a spec for the effect variable's value).

For example, consider the declaration:

```
(state clif_trap (type Bool) (default (not clif_trap)))
```

It creates a state variable to model whether the input IR sequence `traps` in CLIF semantics. The default `(not clif_trap)` applies when none of the terms in an expansion modify `clif_trap`.

Specifying Term Effects. To describe a term's effect on state, developers use `modifies` clause in the spec:

```
(spec (<term> <args...>) (modifies <state> <cond>?) ...)
```

The `modifies` clause declares that the associated term modifies `state`. If the modification is conditional, the clause should also include a condition variable `cond`. When a spec declares it modifies a given state effect, the spec body must provide constraints that define when `cond` holds and the implied constraints on `state` when it does.

When verifying an expansion, the modifies clauses and their condition variables determine when the default spec for a state effect applies. That is, Arrival collects all modifies clauses for a given `state` and assumes the default spec when all conditional variables are false.

4.3 Case Study: CLIF-to-AArch64 State Modeling

Next, we walk through an example of Arrival's state modeling for CLIF to AArch64 lowering.

Loads. In Arrival's approach, we do not model all of memory. Instead, we model the *parameters* of a possible *load effect* by a CLIF instruction with a state variable.

```
(state clif_load
    (type (struct (active Bool)
                  (size_bits Int)
                  (addr (bv 64))))
    (default (not (:active clif_load))))
```

The state variable expresses the fact that a load may or may not happen (via the `active` flag) and its size and address when it does. The default case is an inactive load. We define a parallel state variable `isa_load` for a possible load operation on the ISA side of the lowering. We also define a bit-vector `loaded_value` state variable for the symbolic value produced by a load.

```
(state loaded_value (type (bv 64)) (default true))
```

The specification of the CLIF `load` instruction indicates that it modifies the `clif_load` effect and how the load parameters relate to instruction parameters. In addition, the result of `load` is the `loaded_value` state variable, truncated to the load size. This symbolic state variable enables Arrival to correctly find errors where, for example, the IR specifies that a loaded value should be sign-extended but the ISA performs a zero-extend.

```
(spec (load flags p offset)
    (modifies clif_load) (modifies loaded_value)
    (provide
        ; Activate the CLIF load effect
        (:active clif_load)
        ; Load size is the width of the loaded value.
        (= (:size_bits clif_load) (widthof result))
        ; Address calculation.
        (= (:addr clif_load) (bvadd p (sign_ext 64 offset)))
        ; Result of the load is represented by low bits of the loaded value state register.
        (= result (conv_to (widthof result) loaded_value))
    )
)
```

Likewise, Arrival's ISA specification pipeline (§5) automatically detects when instructions access memory and modifies the `isa_load` effect accordingly within the generated spec. The result register of the load instruction is also set equal to the `loaded_value` state variable.

Finally, at the top-level, we must ensure that lowerings from CLIF to ISA preserve load semantics. We do so with a specification on the top-level `lower` entry point (§2):

```
(spec (lower arg)
  (provide
    ; Either both active, or both not.
    (= (:active clif_load) (:active isa_load))
    ; If active, their parameters must match.
    (=> (:active clif_load) (= clif_load isa_load))
    ; ...
```

This specification states that `clif_load` and the `isa_load` must both be inactive or both active, and, if both are active, their parameters must be equivalent.

Traps. To describe trapping behavior, we introduce variables for whether CLIF semantics require a trap, and whether one happens in execution.

```
(state clif_trap (type Bool) (default (not clif_trap))
(state exec_trap (type Bool) (default (not exec_trap))))
```

These effect variables are boolean flags—either they fire or they don't—and they default to not firing. The top-level `lower` specification asserts that for any possible lowering, `clif_traps` if and only if `exec_traps`.

CLIF instruction specifications describe trap semantics by specifying when `clif_trap` is true, as we saw for `sdiv` (§2.4.2). Likewise, the specification for Cranelift's `TrapIf` pseudo-instruction defines the conditions under which it triggers a trap.

```
(spec (MInst.TrapIf kind trap_code)
  (modifies exec_trap this_inst_traps)
  (provide
    ; Conditions under which this instruction traps.
    (= this_inst_traps (match kind
        ((Zero r) (bv_is_zero! r))
        ((Cond cc) (cond_holds! cc (:flags_in result))) ...))
    ; If this instruction traps, set the global trap state.
    (=> this_inst_traps exec_trap)))
```

Crucially, the use of the condition variable `this_inst_traps` allows lowerings to instruction sequences that can potentially trap in multiple ways.

Flags. Lastly, we note that while we could use Arrival's state modeling for AArch64 flags, it is natural to build on ISLE's existing conventions for flags handling. Cranelift uses a `with_flags` combinator to wrap instruction sequences that pass flags between them, ensuring they are kept consecutive when machine code is emitted. Given a producer and consumer instruction, ISLE rules construct a `(with_flags producer consumer)` term when flags must be passed from one to the next. This affords an elegant way to *specify* the transfer of flags between instructions.

We specify that the representation of machine instructions during verification have input and output flags (AArch64's NZCV flag bits):

```
(model MInst (type (struct (flags_in  (named NZCV))
                           (flags_out (named NZCV)))))
```

Given this, the specification of the `with_flags` term asserts that the `flags_out` of the producer equals the `flags_in` of the consumer.

4.3.1 Full State.

The full set of state variables defined for Arrival's current AArch64 verification scope are:

- *Loads:* Load parameters on both sides (`clif_load` and `isa_load`), and the `loaded_value` when a load is active.
- *Stores:* Store parameters on both sides (`clif_store` and `isa_store`), handled similarly to loads.
- *Traps:* Trap flags on both sides, `clif_trap` and `exec_trap`.
- *Floating-point:* The floating-point control register must exist for ISA floating point instructions. We consider it to have a fixed configuration, with state variable `fpcr` as a placeholder.

5 Auto-Generating High-Assurance Machine Instruction Semantics

Verifying an instruction selector relies on a foundation of trusted machine-instruction specifications; a flaw in these specs can undermine the verification guarantees. Arrival scales to industrial compilers by automatically deriving verification-friendly, high-assurance specifications for families of ISA instructions. Arrival aims for specifications that are **(1)** *high-assurance*, with trust backed by authoritative, vendor-provided semantics; **(2)** *integrated* with the compiler's backend; **(3)** *symbolic*, enabling reasoning over families of instructions; **(4)** *succinct*, covering only behavior required for user-space verification; and **(5)** produced with *automated* assistance.

High Assurance. Processor vendors are best-placed to develop and validate high-trust instruction specifications; our specifications should derive trust from them. Hardware manufacturers—led by Arm—have begun to release formal, machine-readable semantics. Arm's A-class and M-class architecture specifications [54] are encoded with XML structured data and a custom Architecture Specification Language (ASL). RISC-V uses Sail for its formal specification [4, 5], while Intel researchers plan to release an ASL x86 specification in the near future [55].

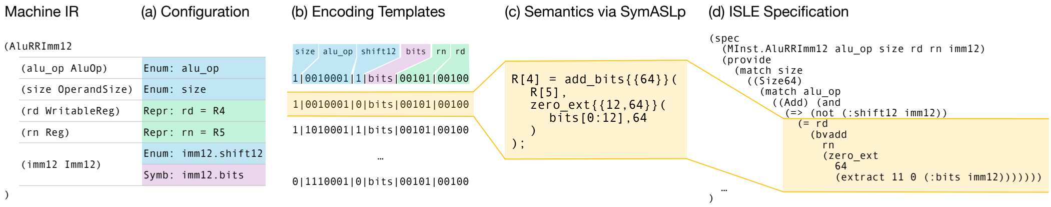
Integrated. Production compilers must have an in-memory machine instruction IR. Cranelift uses backend-specific `MInst` types, while LLVM's Machine Code layer [40] has an `MCInst` representation. Machine instruction IRs are *almost assembly*, but with differences that serve critical functions in the compiler. For example, Cranelift's `MInst` provides an SSA interface to machine code with virtual registers (prior to register allocation). `MInst`'s variants share common properties (e.g., the `AluRRR` variant covers arithmetic operations that involve three registers), but differ from vendor categorizations. As another example, LLVM's `MCInst` is designed to facilitate further peephole optimizations. In contrast, vendor semantics are at a lower abstraction level (i.e., actual bit-by-bit instruction encodings), so they must be *lifted* to the machine code IR. Conversely, our validation must also cover the compiler's assembler, since it ultimately maps the machine IR down to concrete encodings, and bugs in this process risk invalidating our specifications.

Symbolic. Compiler verification must reason about machine instructions that are partially *symbolic*. Consider Cranelift's instruction family `MInst.AluRRImmShift`, representing a register shifted by an immediate (constant) shift amount. An example instance of `AluRRImmShift` is the assembly `lsl w8,w9,#24`, which encodes a left-shift of register `w9` by the immediate amount of 24, where 24 is encoded directly into specific bits of the instruction. Cranelift has ISLE lowering rules mapping shifts by constant amounts to `AluRRImmShift` instructions. When verifying such a rule, *considered over the space of all input programs*, these immediate fields in the instruction encoding must be treated as variables. Therefore, we need specifications for instruction encodings with symbolic ranges of bits.

Succinct. When verifying a compiler, machine instruction specifications should *succinctly* capture user-space behavior. For a Cranelift `(MInst.AluRRR (ALUOp.Add) ...)` instruction, the specification should just be a bit-vector add (`bvadd`), while for `(MInst.ULoad16 ...)`, it should capture a 16-bit load effect at a computed address (§4). While the processor might in fact perform page-table lookups, increment performance counters, or perform many other micro-architectural actions, none of these details are required for instruction-selection verification.

In contrast, while vendor specifications provide high assurance, they are not succinct as-is. This makes sense, as full processor specifications serve many levels of the software stack—from validating hardware designs and operating system kernels up to reasoning about user-space software. Moreover, Arm's specification covers every possible combination of processor features. We must distill these specifications down to user-space essentials; concise specifications improve the efficiency of underlying solvers and aid debugging when verification fails.

Automated. Modern ISAs contain thousands of instructions—each with intricate details—and production compilers target a large subset. Historically, most software verification projects have



```
(AluRRImm12
    (alu_op AluOp)         Enum: alu_op
    (size OperandSize)     Enum: size
    (rd WritableReg)       Repr: rd = R4
    (rn Reg)               Repr: rn = R5
                           Enum: imm12.shift12
    (imm12 Imm12)
                           Symb: imm12.bits
)
```

```
1|0010001|1|bits|00101|00100
1|0010001|0|bits|00101|00100
1|1010001|1|bits|00101|00100
              …
0|1110001|0|bits|00101|00100
```

```
R[4] = add_bits{{64}}(
   R[5],
   zero_ext{{12,64}}(
      bits[0:12],64
   )
);
```

```
(spec
  (MInst.AluRRImm12 alu_op size rd rn imm12)
  (provide
    (match size
      ((Size64)
        (match alu_op
          ((Add) (and
            (=> (not (:shift12 imm12))
              (= rd
                (bvadd
                  rn
                  (zero_ext
                    64
                    (extract 11 0 (:bits imm12)))))))
  …
)
```

Fig. 4. Arrival's specification generation pipeline for `AluRRImm12` instructions. **(a)** Developer-provided configuration determines how to parameterize the instruction family. **(b)** Arrival produces encoding templates using the Cranelift assembler, one for each possible value of the enumerated fields (`alu_op`, `size` and `imm12.shift12`), and splices in the symbolic bit range `bits`. **(c)** SymASLp generates semantics for each encoding template. **(d)** Arrival translates each semantics block into one case of the full ISLE specification.

written their own ISA semantics [13, 14, 46]. Custom semantics provide succinct specs tightly integrated with the target IR. However, hand-written specs require massive developer effort and cannot provide high assurance. Transcribing from ISA manuals is error-prone, and such documentation can itself be buggy [22]. Instead, machine-instruction specifications should be produced with the *assistance of automation*. Some input from the developer is necessary, especially to configure the specifics of the compiler's in-memory machine-code IR, but configuration should be modest.

In this section we show how Arrival achieves these goals. We illustrate Arrival's approach by example (§5.1) before describing the ISA specification pipeline in detail (§5.2).

5.1 Overview of Arrival's Approach: Semantics from Symbolic Instruction Encodings

We illustrate Arrival's approach with a running example throughout the section. The `AluRRImm12` instruction family represents arithmetic on a register and a constant expressed in AArch64's `Imm12` immediate format. Cranelift's machine code IR represents the `AluRRImm12` family in ISLE as:

```
(AluRRImm12 (alu_op ALUOp) (size OperandSize) (rd WritableReg) (rn Reg) (imm12 Imm12))
```

Note the type of the last parameter, `Imm12`, is itself a structured type with two fields: a 12-bit `bits` constant and a flag `shift12` that indicates whether to shift `bits` by 12 before the operation.

Considered as an ISLE term, our desired specification for `AluRRImm12` would treat its inputs as *symbolic variables*: `rn` represents the value contained in the input register, and `imm12` is the value of the instruction immediate. The specification should describe how the instruction performs an arithmetic operation based on these symbolic inputs.

However, vendor specifications capture the semantics of concrete *instruction encodings*. Overall, `AluRRImm12` represents a massive family of encodings: four possible arithmetic operations, two possible operand sizes, 32 possible output registers, 32 possible input registers, two possible immediate shift flags, and 4,096 possible immediate bits. In total: 67,108,864 concrete instruction encodings! Generating semantics for all of them is out of the question (let alone feeding those results to an off-the-shelf SMT solver).

Arrival's approach bridges the gap between symbolic values and instruction encodings, enabling all 67,108,864 encodings to be reduced to a tractable state space. At a high level, the approach partitions the entire `AluRRImm12` family into subsets, each represented by an *encoding template* (an instruction encoding with symbolic bit ranges), and computes succinct semantics for each one.

Arrival builds upon recent work by the ASLp project [2, 39], which showed how to reduce the verbosity of Arm's specifications via *partial evaluation*. Their tooling maps a *fully concrete* 32-bit encoded instruction to a (usually) short program describing that instruction's behavior. For example, given the concrete instruction `0x91048ca4`, ASLp produces a "reduced ASL" program that roughly says:

```
R[4] = add_bits{{64}}(R[5], 291);
```

This corresponds to the assembly code: `add x4,x5,#291`, i.e., adding the immediate constant value 291 to the value of register `x5` and storing the result in register `x4`.

To derive semantics for instruction encoding templates, we built SymASLp, a fork of ASLp extended to support partial evaluation on symbolic instruction encodings. For example, SymASLp can derive semantics for the set of instructions `add x4,x5,#<bits>`, with encoding template `1|0010001|0|bits|00101|00100`. The template describes: 64-bit size (`1`), `add` opcode (`0010001`), an immediate that is not shifted (`0`) with a value given by a 12-bit variable (`bits`), and register indices (`00101` and `00100`). Given this template, SymASLp produces a short semantics program (Figure 4) for all 2^{12} instruction encodings in the template, in terms of the `bits` variable. With templates like this, SymASLp can derive concise semantics for entire subsets of the `AluRRImm12` family at once.

Using SymASLp, Arrival's pipeline generates the `AluRRImm12` specification as illustrated in Figure 4 and described below (with further details in §5.2):

(a) *Configuration.* Developers use Arrival's Rust library for ISA-specification generation to configure how the `AluRRImm12` family should be parameterized, or partitioned, into encoding templates. In this case, we *enumerate* (`Enum`) over the 4 × 2 × 2 possible values for the opcode (`alu_op`), operation size (`size`), and immediate shift flag (`imm12.shift12`). We introduce a *symbolic* (`Symb`) bit range (`bits`) into the encoding templates for the immediate value (`imm12.bits`). We reduce the possible registers to one by picking a fixed *representative* (`Repr`) register for each of `rd` and `rn`, as we'll see in §5.2.

(b) *Encoding Templates.* The developer-provided configuration partitions the seemingly intractable state space of `AluRRImm12` encodings into 16 encoding templates. Arrival uses Cranelift's assembler to compute the encoding template bit patterns and to splice in the symbolic bit range (`bits`).

(c) *Semantics via SymASLp.* For each encoding template, SymASLp derives imperative semantics in reduced ASL.

(d) *ISLE Specification.* Arrival translates SymASLp's output semantics into ISLE's functional specification language. Arrival's translator maps processor state accesses in vendor specifications to ISLE variables and Arrival's effect model (§4). The result is ISLE specification source code, consumed by Arrival to verify lowerings to `AluRRImm12` instructions.

Finally, Arrival validates that the instruction encoding templates are consistent with the Cranelift assembler (§5.2.3). That is, Arrival provides assurance that the 16 cases we defined for `AluRRImm12` are identical to the encodings the assembler produces.

5.2 Details of Arrival's ISA-Specification Generation Pipeline

Arrival takes as input authoritative, vendor-validated ISA specifications written in ASL. Developers use Arrival's Rust library to configure instruction families (§5.2.1). Arrival automatically generates ISLE specifications for each configured machine instruction (§5.2.2). The results are succinct enough to be human-readable and checked into the Arrival codebase, decoupling specifications from the pipeline and its dependencies. Arrival also automatically validates the configured instruction encodings against those produced by Cranelift's internal assembler (§5.2.3). This moves more of Cranelift's code (e.g., much of its assembler) out of our trusted computing base.

5.2.1 Configuration. For each machine IR instruction family (e.g., `MInst.AluRRImm12` from §5.1), the Arrival generator requires a configuration to determine how to partition the family into instruction encodings. The configuration describes how each field should be handled:

- *Enumeration.* Instruction fields can be handled by enumerating over all possible values, for example, `AluRRImm12`'s opcode and size fields. The resulting specification will contain branches over these case splits. This strategy makes sense when the field has relatively few possibilities and the instruction semantics have a complex dependency on the field's value (such that symbolic execution might not produce a succinct result).
- *Symbolic Fields.* Fields such as immediate values may be preserved as symbolic fields in the encoding template. Developers configure which range of bits the field corresponds to; for example, the `AluRRImm12` configuration places a 12-bit `bits` variable at offset 10. The rest of the encoding template is computed by the Cranelift assembler. (In future work, we hope to remove this obligation by generating both the assembler and specifications from shared data sources.) This strategy makes sense when the field has many possible values and the instruction semantics utilize the field in a simple way (for example, immediate constants).
- *Concrete Representative.* In the case of registers, for example, we can recognize that all registers of the same class have equivalent semantics (with the limited exception of `R31` on AArch64). Therefore, we can *specialize* to a fixed, general representative register, and generalize the resulting specification to arbitrary register values.

We have configured 559 AArch64 instructions[1] (across 39 families). The configuration totals 1.9K tedious but mostly straightforward lines of Rust and ultimately generates 12.6K lines of specs.

5.2.2 Generation. Arrival processes the developer-provided configuration into a list of encoding templates. Next, it uses SymASLp to generate semantics for each encoding template in a reduced subset of the ASL language. SymASLp extends ASLp to make the type of an input instruction symbolic rather than a concrete bit-vector. Therefore, SymASLp accepts encoding templates as inputs, which are represented as concatenations of symbolic and concrete bit-vectors. In order to produce succinct resulting semantics, SymASLp adds further symbolic expression simplification and optimization logic to ASLp.

Given the imperative semantics output by SymASLp, Arrival translates them by symbolic execution, building up ISLE constraints as it goes. It introduces temporary variables for versioned mutable state and compiles conditional blocks into if-then-else expressions. Arrival converts operations on processor state to ISLE variables and state effects. Memory operations appear as `Mem.read` and `Mem.set` function calls in the semantics; Arrival translates them into the constraints on the load/store-effect state variables described in §4.

Reduced ASL and ISLE's spec language are at a similar abstraction level, meaning that most ASL operators have natural equivalents. For example, ASL's `cvt_bits_uint` (convert bits to unsigned integer) can be translated directly to ISLE's (and SMTLIB's) `bv2nat`. In a few complex cases, floating-point numerics are implemented as macro calls to an ISLE primitives library, which consists of 15 numeric routines carefully hand-written by line-by-line transcription from Arm's manual. This approach follows ASLp—which also treats floating-point numerics as primitives—and importantly allows us to use SMTLIB's FloatingPoint theory [15, 62] without the theory of Reals [61].

Finally, once Arrival has generated a specification for each encoding template in the configured instruction family, it unifies them into a complete specification. Arrival uses nested `switch` and pattern `match` expressions to codify the case splits in the provided configuration. The resulting specifications are printed as ISLE source code and committed to the codebase.

[1] Precisely, 559 encoding templates, which roughly correspond to the colloquial understanding of instructions

			Result			Solver	
Category	Expansions	Type Inst.	Verified	Timeout	Inapplicable	cvc5	Z3
Memory	5,832	10,368	10,368	0	0	10,368	0
Float	97	85	66	0	19	75	10
Rest (Integer etc.)	283	563	373	27	163	535	28
Total	6,212	11,016	10,807	27	182	10,978	38

Table 1. Verification results for selected Wasm-critical lowerings (§6.1) by category. Rule chaining produces the given number of expansions, and type inference derives potential type instantiations for each expansion. For each type instantiation, the verifier either deems it verified, inapplicable, or times out. Type instantiations are processed by either cvc5 or Z3 backend SMT solvers.

5.2.3 Validation. Assemblers are difficult to get right: Cranelift's assembler is implemented in 3.5K lines of intricate logic. In addition, configuring instruction encoding templates in Arrival's specification pipeline is potentially subject to error. A bug in the assembler or configuration would cause us to generate specifications for the wrong instruction encodings. Arrival rules out this possibility with a final validation step. This step enumerates all possible concrete encodings from the configured templates and confirms they match the result of assembling the compiler's machine IR version of the same instruction. The key distinction is that while generating (and committing the code for) semantics of 2^n concrete variations (where $n \approx 16$) is impractical, doing a basic validation step in Rust is quick. In practice, this check helped us immediately identify a case where we placed a symbolic immediate at the wrong offset. Passing this validation step moves (parts of) Cranelift's assembler out of our trusted computing base.

6 Evaluation

We evaluate Arrival on three research questions.

(1) What coverage can Arrival achieve in Cranelift's WebAssembly-critical lowerings?
(2) Does Arrival reduce specification burden?
(3) Is Arrival performant enough to be incorporated into developer workflows?

We use Arrival to verify a large subset of Cranelift's Wasm-relevant instruction set, with near-complete coverage of loads/stores and the core integer and floating-point numerics (§6.1). Furthermore, Arrival substantially reduces specification burden: 60% of 410 terms in the verified subset have automation-assisted specs. (§6.2). Lastly, we show that verification runs could be done in scheduled CI environments, and local development on specific rules is typically fast (§6.3).

6.1 What verification coverage can Arrival achieve?

We use Arrival to verify Cranelift instruction lowerings for CLIF instructions emitted by Wasmtime's Wasm compiler. We focus on Wasm's core instruction set [1], where verification is required to provide assurance of its sandboxing security guarantees. Our work verifies, with minor exceptions: all integer numerics, (non-atomic) memory loads/stores, floating-point numerics, and conversions.

We verify Cranelift's AArch64 backend, because it is actively-maintained, widely-used, and benefits from Arm's machine-readable semantics. RISC-V has a formal specification in Sail [4], but RISC-V is less widely used and a Tier 3 target [66]. Intel's promised machine-readable specification [55] is not yet available; applying it to Cranelift would be exciting future work.

Arrival supports tagging ISLE rules and terms by their categories, which we use to select subsets of expansions for verification. Tags include categories like `wasm_proposal_mvp` for Wasm 1.0 core functionality—the Minimum Viable Product (MVP) release—and `float` for floating-point numerics and conversions to/from integers.

Our verification scope encompasses 6,212 total expansions (distinct chains of rules). As Arrival's type inference and applicability checks process these expansions, some are deemed impossible due to type or pattern-matching predicate conflicts. Others have multiple possible applicable type instantiations (i.e., the same rule chain could apply to both 32- and 64- bit inputs). As a result, a total of 10,834 type instantiations are deemed applicable and proceed to verification.

We measure performance on a desktop workstation with Intel i7-13700K processor with 8 performance-cores and 8 efficiency-cores, totaling 24 threads with hyper-threading.

Table 1 shows the full break-down of expansion verification results by category. Across all expansions, 10,807 are verified, while 27 (0.2%) time out with a solver timeout of 4 hours. The overwhelming majority of queries complete in a matter of a few seconds (§6.3). We use two backend SMT solvers (cvc5 [8] and Z3 [23]) and enable developers to select a default and configure an override by applying tags. We found the best performance with cvc5 for most expansions and Z3 for certain floating-point and bitwise operator terms (38 expansions). Expansions that timeout are: some divisions and remainders, and population count. Such cases have always been a challenge for the underlying SMT solvers [27, 31, 43]. We do find that although proving `UNSAT` in these cases is difficult, our anecdotal experience shows that Arrival can quickly provide counter-examples when bugs are purposefully introduced.

Our verification coverage has limited caveats:

- We do not model Cranelift's constant pool, which deduplicates constant accesses using a small cache. This prevents verification of one lowering of `f64const` instructions.
- In a few cases, we refactored Cranelift's ISLE rules to avoid recursion. This allowed us to benefit from rule chaining and avoid writing specifications for those terms. These changes were made in collaboration with Cranelift engineers, who confirmed they would accept the changes upstream.
- Cranelift's mid-end optimizations are not considered in the classification of WebAssembly-relevant expansions. The set of IR terms for verification is based on those reachable from Cranelift's Wasm-to-CLIF translator. After the mid-end optimizations, the set of reachable IR terms might be larger, and hence visit expansions not covered by our verification.

Arrival also covers some Cranelift extensions to Wasm, for example 8- and 16-bit integer operations (revealing the `sdiv` bug). Further work could expand our coverage to all of Wasm core (e.g., stack and control flow), and go beyond Wasm 1.0 (e.g., SIMD).

Comparison to Prior Work. In both breadth and depth of coverage, Arrival goes beyond prior efforts to verify Cranelift's instruction selector [63]. Arrival **(1)** verifies memory and floating-point lowerings; **(2)** achieves deeper *end-to-end* coverage of the same lowerings, down to machine code IR; **(3)** verifies more behaviors (e.g., trap semantics); and **(4)** uncovers previously missed bugs (§2.4.1). Arrival's rule chaining makes direct comparisons difficult, but to convey a sense of scale: Arrival covers 3.3× more rules and 29.2× more type instantiations.

6.2 Does Arrival reduce specification burden?

We evaluate specification burden by considering the specs required to verify the large subset of the WebAssembly-critical backend described in §6.1. We consider how many specs were required, how many of those were provided with the assistance of Arrival's automation, and how much manual effort it saved. Overall, we find that of the 410 term specs required, 60% benefit from some form

		Hand-Written			Automated			
#	Category	Terms		Lines	Terms		Cases	Lines
(1)	ISA	3	7%	173	39	93%	559	12,608
(2)	Internal	13	5%	116	211	95%		-
(3)	Cranelift IR	60	100%	734				
(4)	External	84	100%	529				

Table 2. Specs required for Wasm-critical verification (§6.1) organized by category and automation. Categories are: **(1)** ISA machine-code terms, **(2)** Internal terms (e.g., intermediate terms and ISLE types), **(3)** Cranelift IR, and **(4)** External terms (e.g., Rust predicates). For each category, we show the number and proportion of terms that use hand-written vs. automated specs, and the number of lines of ISLE source code required. Given the absence of formal specifications for Cranelift IR or External terms, automated specifications are not possible.

of Arrival's automation, including almost all of the most complex ISA specs. Indeed, of the 266 terms where automation was feasible, we automated 93%.

Table 2 shows how specs benefit from automation, broken down by the major categories below.

ISA. Arrival's ISA specification pipeline (§5) automatically derives the vast majority of instruction family specifications. ISLE's machine instruction IR groups many instructions into a small set of top-level terms. For example, 19 distinct arithmetic instructions use `AluRRR` (which is then further specialized based on operand size). In total, the generated specifications cover 559 encoding templates across 39 ISLE terms, leading to 12,608 lines of ISLE specification language. The configuration required to parameterize all instruction families is 1.9K lines (6.5× less than the generated specifications). This code is largely boilerplate, and the ISA specification pipeline provides guard-rails to detect mistakes (§5.2.3). Future work could express this configuration in a more concise DSL.

Three specific ISA terms still require hand-written specifications. Two immediate encoding cases—`AluRRImmLogic` and `AluRRImmShift`—present substantial challenges for the ISA specification pipeline. Both cases use an unusually complex part of the ASL specification for the encoding of logical immediates in AArch64 [25]. While it might have been possible to proceed with code-generation, we opted for simpler hand-written specifications instead. The other exception to automation here are pseudo-instructions, e.g., `TrapIf` (mentioned in §4). Pseudo-instructions do not directly correspond to AArch64 instructions and have tight integration with the Cranelift engine; therefore specifications cannot be derived purely from vendor-provided semantics.

Internal. Internal terms are defined within ISLE itself: mostly intermediate terms used to express instruction selection logic, or terms introduced by ISLE type definitions (e.g., enum variants). Arrival's automation means that very few internal terms require manual specification. With rule chaining, most intermediate terms (116) are automatically inlined (§3); therefore developers do not have to provide specifications. In addition, Arrival synthesizes specifications for most ISLE-defined types (95 terms), allowing authors to reason about them without additional specification effort.

Cranelift IR. Since Cranelift IR has no formal semantics, we wrote our own. Where possible, we used line-by-line transcriptions of the WebAssembly specification associated with the IR term, or specifications already added to ISLE in prior work [63]. In other cases, we relied on inspection of the implementation and discussions with developers. In the long run, we expect that semantics for a large subset of CLIF could be automatically derived from Wasm's semantics once the migration to SpecTec [68] is complete, with human input for the non-one-to-one operators.

External. External terms are bridges to external Rust functions, mostly for pattern-matching predicate logic. Their behavior must be specified by hand for Arrival to reason about them. External specifications are trusted, as well as some trusted helpers (e.g., the `with_flags` combinator).

Comparison to Prior Work. The Crocus verifier [63] would have required hand-written specifications for all 410 terms, or 2.6× more than Arrival. The initial Crocus verification effort required 136 annotations [63, §4.1]. At the cost of 24 more hand-written specifications Arrival achieved far broader coverage (§6.1).

6.3 Is Arrival performant enough to be incorporated into developer workflows?

In this section, we evaluate whether Arrival's performance is sufficient to meet our usability goals. Can it produce results fast enough to be used in Cranelift development, both in CI and in local iteration on instruction-selection rules?

We envision that continuous-integration might execute full-coverage verification runs on a schedule (e.g., nightly or weekly). In this context, developers are not blocked on feedback, so execution times of up to a few hours could be acceptable. In addition, runs should be possible on commodity hardware readily available from cloud providers. Arrival is multi-threaded and will parallelize expansions over available cores. For high-assurance, some verification runs should use large solver timeouts to maximize coverage. In this case, the overall time is dictated by the selected timeout. There is also value in more frequent lower-timeout runs that bring most of the value with far less compute resources. With a solver timeout of 60 seconds and using cvc5 only, the verification run in §6.1 takes 10.06 minutes to complete. This run achieves almost the same total coverage, with 55 expansions timing out instead of 27. While a 16-core machine is powerful, it could be a reasonable expense for a job lasting a few minutes once a day. With Arrival's parallelization, the job could scale down to smaller hardware and still take an acceptable amount of time in CI. With this performance, it could also be possible to run verification for a selected subset of rules on pull-requests affecting ISLE files.

For local iteration, developers seek faster feedback, typically on a laptop rather than server-class hardware. In this context, limited runs over in-development rules are more common, and developers will typically benefit from the fact that the common-case verification time for a single type instantiation is a few seconds. Figure 5 shows the distribution of type instantiation verification times across the full verification run, showing that the vast majority complete in 5 seconds or less. For a developer utilizing Arrival's filtering mechanisms to narrow down to an in-development rule, each run will hit a modest number of type instantiations and produce feedback in seconds or minutes.

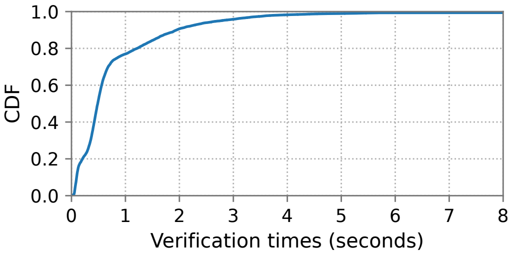


Fig. 5. Cumulative distribution function (CDF) of verification times for type instantiations in the full verification run from §6.1.

7 Related Work

Compiler Verification. The CompCert [41] and CakeML [37] compilers are verified end-to-end with proof assistants. However these efforts came at great cost [60], and the resulting compilers compromise on performance. Therefore, their use tends to be limited to safety-critical domains. Recent work [9] has built upon CompCert's backend in the JIT setting.

Jitterbug [46] provides a verified JIT framework targeted at BPF JITs in the Linux kernel. While Jitterbug proves a more complete correctness property than Arrival, its techniques apply to comparatively simple "template" instruction selection. Jitterbug would not scale to instruction selectors in more complex compilers like Cranelift.

Like Arrival, some prior projects introduce formal methods to improve practical compiler assurance short of a full end-to-end guarantee. Alive2 [42] applies translation validation to LLVM IR to find bugs in LLVM's mid-end optimization passes, while TurboTV [38] extended these techniques to JIT compilation in V8's JavaScript engine. VeRa [16] verifies the range-analysis rules in a browser JIT. The Icarus [59] JIT DSL verifies inline-cache implementations for the Firefox engine. Arrival's approach emphasizes the use of automation to reduce specification burden, and therefore verify a large component of a production compiler.

Instruction-Selection Correctness. ISLE was designed with automated testing and verification in mind [26]. The Crocus [63] tool individually verified a more restricted subset of ISLE rules. Compared to Arrival, Crocus required hand-written ISA specifications, could not reason about side effects, mandated one-by-one rule verification (limiting scaling), and did not achieve end-to-end verification (missing bugs as a result). RGFuzz [48] applies rule-guided fuzzing to Cranelift's ISLE rules, finding bugs as a result [47]; however, fuzzing lacks the assurance of verification.

Beyond ISLE, DSLs for term-rewriting systems are common in optimizing compilers, for example Go's SSA rewrite rules [7]. Alive [43] verifies LLVM optimization rules in a custom DSL. The PyPy JIT engine uses a Peephole Rule DSL with SMT-based checks [11], though the rules are single-step rewrites (lacking ISLE's composition), and the ruleset is tiny relative to Cranelift's [12].

Synthesis attempts to ensure rules are correct by construction. There is a long history of work on automatic derivation of code generators [17, 18, 24, 30]. Recently, Hydra [45] generates peephole optimizations for LLVM by generalizing results from superoptimization.

Formal ISA Semantics. Arm's machine-readable specification [54] led the way in formal ISA semantics from major processor vendors. Their specifications introduced the Architecture Specification Language DSL and have since supported many applications, for example: formally validating Arm's processor implementations [56], finding bugs in ISA emulators [32], and synthesizing SIMD logic [36].

Our ISA semantics generation builds on ASLp [2, 19, 39], a partial evaluator for ASL that produces succinct semantics for concrete instructions. ASLp is suited to applications such as binary lifting and translation validation, where the required reasoning is over concrete encoded instructions. However, ASLp's output is not sufficient to reason about symbolic instructions as required by compiler verification. For instance, it cannot provide semantics for instruction families, such as `add <r1>,<r2>,#<imm>` with symbolic registers and immediate value `imm`. Furthermore, ASLp operates at the level of encoded instructions, leaving a gap to integrate with a compiler's machine code IR.

The Sail [5] language and ecosystem is a rich platform for processor specification, notably RISC-V's official formal specification [4]. Sail supports translation from ASL [10]. The ISLA [6] symbolic executor for Sail produces SMTLIB-like traces for instruction executions and backs the Islaris [58] machine-code verifier. We considered ISLA as an alternative to ASLp, but while ISLA does have some support for symbolic instructions, its traces lack the succinctness of ASLp's semantics.

Intel has not yet released formal semantics for x86-64. The most complete alternative is the semantics in K for the Haswell architecture [22]; however this is tightly bound to the K framework and has been archived [3]. The (incomplete) x86 semantics for ACL2 [28] are used by the Sail project [50]. We look forward to building upon Intel's expected ASL specification [55].

The CakeML compiler uses hand-written L3-based ISA specifications, which have been formally validated against Arm vendor semantics [35]. In contrast, Arrival removes the need to write specifications by hand.

Hydride [36] demonstrates another exciting use case for vendor-provided ISA specifications in compiler development. Their work extracts semantics from multiple ISAs and synthesizes a shared IR and code generator backend. Hydride focuses on synthesis rather than verification, and vector instruction sets important for compute DSLs such as Halide [52]. In contrast, Arrival lifts formal ISA semantics to verify an existing production compiler backend.

WebAssembly Verification. WebAssembly is an exciting domain for formal verification given its formal specification [1, 53, 67]. The SpecTec [68] project—soon to be officially adopted by Wasm [57]—is a DSL and toolchain that unifies Wasm's specification artifacts (prose, formal specification, reference interpreter, ...). SpecTec, once integrated and mature, could help provide formal IR-level semantics for tools such as Arrival.

vWasm [14] is a Wasm-to-x86 compiler with end-to-end verification of the sandboxing property. However, vWasm does not prove functional correctness, and its performance falls short of production compilers. The same work proposes rWasm, which achieves sandboxing assurance via an embedding of Wasm semantics into Rust; however this approach accepts LLVM in its TCB. VeriWasm [34] does not verify a compiler, but instead verifies that individual binaries compiled from Wasm do preserve the sandbox. Beyond pure Wasm, WaVe [33] verifies a runtime system.

8 Conclusion

We have presented Arrival, a verifier for the security-critical instruction-selection phase of the Cranelift industrial Wasm compiler. Arrival scales to this complex code base via automatic rule chaining, lightweight state modeling, and automatically derived, high assurance specifications for machine instructions. Compared to prior work, our results show that Arrival achieves higher coverage with less developer effort. We are actively working with Cranelift developers to integrate Arrival into their production workflows.

Data-Availability Statement

Arrival is open source at github.com/mmcloughlin/arrival. Our verification results (§6) are documented in our artifact [44]. We are working with the Cranelift developers to upstream Arrival into their open-source repository.

Acknowledgments

We thank Vaishu Chintam for her help verifying floating-point rules; Jamey Sharp, Nick Fitzgerald, Alex Crichton, and the members of the Bytecode Alliance for their advice and support extending the Cranelift codebase; Adrian Sampson, John Regehr, and the anonymous OOPSLA reviewers for their constructive feedback on earlier drafts this paper; and Pratap Singh for his help testing and improving our artifact. This work was supported in part by National Science Foundation (NSF) Grant No. 2154964, and Seed Funding from Carnegie Mellon University's CyLab.

References

[1] 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/
[2] 2024. aslp: Partial evaluator for Arm's Architecture Specification Language. https://github.com/UQ-PAC/aslp
[3] 2024. Projects using K. https://kframework.org/projects/
[4] 2024. Sail RISC-V model. https://github.com/riscv/sail-riscv
[5] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell.

2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3290384

[6] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *Proceedings of the Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-030-81685-8_14

[7] Go Authors. 2024. Introduction to the Go compiler's SSA backend. https://go.dev/src/cmd/compile/internal/ssa/README

[8] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS) International Conference*. doi:10.1007/978-3-030-99524-9_24

[9] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3571202

[10] Thomas Bauereiss, Brian Campbell, and Peter Sewell. 2024. ASL to Sail translation tool. https://github.com/rems-project/asl_to_sail

[11] CF Bolz-Tereick. 2024. JIT Integer Optimization Peephole Rule DSL. https://github.com/pypy/pypy/blob/main/rpython/doc/jit/ruleopt.rst

[12] CF Bolz-Tereick. 2024. JIT Integer Optimization Peephole Rules. https://github.com/pypy/pypy/blob/main/rpython/jit/metainterp/ruleopt/real.rules

[13] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[14] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[15] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. 2015. An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. In *Proceedings of the IEEE International Symposium on Computer Arithmetic (ARITH)*. doi:10.1109/ARITH.2015.26

[16] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3385968

[17] R. G. Cattell. 1980. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (April 1980). doi:10.1145/357094.357097

[18] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. 2005. C Compiler Retargeting Based on Instruction Semantics Models. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*. doi:10.1109/DATE.2005.88

[19] Nicholas Coughlin, A. Michael, and Kait Lam. 2025. Lift-Offline: Instruction Lifter Generators. In *Proceedings of the Static Analysis Symposium (SAS)*. doi:10.1007/978-3-031-74776-2_4

[20] Cranelift Project. 2024. Crocus: An SMT-based ISLE verification tool. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/isle/veri/README.md

[21] Cranelift Project. 2024. ISLE Language Reference. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/isle/docs/language-reference.md

[22] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3314221.3314601

[23] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS) International Conference*. doi:10.1007/978-3-540-78800-3_24

[24] João Dias and Norman Ramsey. 2010. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/1706299.1706346

[25] Dominik Inführ. 2017. Encoding of immediate values on AArch64. https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64

[26] Chris Fallin. 2021. RFC: design of ISLE instruction-selector DSL. https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-isel-isle-peepmatic.md

[27] Zhoulai Fu and Zhendong Su. 2016. XSat: A Fast Floating-Point Satisfiability Solver. In *Proceedings of the Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-319-41540-6_11

[28] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. 2017. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. doi:10.1007/978-3-319-48628-4_8

[29] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3140587.3062363

[30] Roger Hoover and Kenneth Zadeck. 1996. Generating machine specific optimizing compilers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/237721.237779

[31] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. 2009. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Proceedings of the Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-642-02658-4_53

[32] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3503222.3507736

[33] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. doi:10.1109/SP46215.2023.10179357

[34] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. Trust but verify: SFI safety for native-compiled Wasm. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2021.24078

[35] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. 2022. Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification. In *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. doi:10.4230/LIPIcs.ITP.2022.20

[36] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3620665.3640385

[37] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2535838.2535841

[38] Seungwan Kwon, Jaeseong Kwon, Wooseok Kang, Juneyoung Lee, and Kihong Heo. 2024. Translation Validation for JIT Compiler in the V8 JavaScript Engine. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639189

[39] Kait Lam and Nicholas Coughlin. 2023. Lift-off: Trustworthy ARMv8 semantics from formal specifications. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*. doi:10.34727/2023/ISBN.978-3-85448-060-0_36

[40] Chris Lattner. 2010. Intro to the LLVM MC Project. https://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html

[41] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM (CACM)* (July 2009). doi:10.1145/1538788.1538814

[42] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3453483.3454030

[43] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2737965

[44] Michael McLoughlin, Ashley Sheng, Chris Fallin, Bryan Parno, Fraser Brown, and Alexa VanHattum. 2025. *Scaling Instruction-Selection Verification Against Authoritative ISA Semantics*. doi:10.5281/zenodo.16929954

[45] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/3649837

[46] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi20/presentation/nelson

[47] Junyoung Park. 2024. X64 Fixed select + load floating point wrong lowering. https://github.com/bytecodealliance/wasmtime/issues/8112

[48] Junyoung Park, Yunho Kim, and Insu Yun. 2025. RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00003

[49] Cranelift Project. 2025. Cranelift. https://cranelift.dev
[50] REMS Project. 2024. ACL2-to-Sail translator and the resulting Sail x86 ISA model. https://github.com/rems-project/sail-x86-from-acl2
[51] Wasmtime Project. 2025. Wasmtime: A fast and secure runtime for WebAssembly. https://wasmtime.dev
[52] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2491956.2462176
[53] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3591265
[54] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*. doi:10.1109/FMCAD.2016.7886675
[55] Alastair Reid. 2024. Engineering large, multipurpose microprocessor specifications (using the x86-64 architecture as a case study). https://www.youtube.com/watch?v=tVdUFM3_cCM
[56] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of ARM Processors with ISA-Formal. In *Proceedings of the Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-319-41540-6_3
[57] Andreas Rossberg. 2025. SpecTec Update and Poll. WebAssembly Community Group Meeting (March 11, 2025). https://github.com/WebAssembly/meetings/blob/main/main/2025/presentations/2025-03-11-rossberg-spectec.pdf
[58] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3519939.3523434
[59] Naomi Smith, Abhishek Sharma, John Renner, David Thien, Fraser Brown, Hovav Shacham, Ranjit Jhala, and Deian Stefan. 2024. Icarus: Trustworthy Just-In-Time Compilers with Symbolic Meta-Execution. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3694715.3695949
[60] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2676726.2676985
[61] Cesare Tinelli. 2024. SMT-LIB Reals Theory. https://smt-lib.org/theories-Reals.shtml
[62] Cesare Tinelli and Martin Brain. 2024. SMT-LIB FloatingPoint Theory. https://smt-lib.org/theories-FloatingPoint.shtml
[63] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. 2024. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3617232.3624862
[64] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/168619.168635
[65] Wasmtime Project. 2023. Guest-controlled out-of-bounds read/write on x86_64. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8
[66] Wasmtime Project. 2024. Tiers of Support in Wasmtime. https://docs.wasmtime.dev/stability-tiers.html
[67] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. doi:10.1145/3167082
[68] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3656440

Received 2025-03-24; accepted 2025-08-12