LIGHTWEIGHT FORMAL METHODS FOR CORRECT, EFFICIENT SYSTEMS PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Alexa VanHattum

August 2023

LIGHTWEIGHT FORMAL METHODS FOR CORRECT, EFFICIENT SYSTEMS PROGRAMMING

Alexa VanHattum

Cornell University 2023

Compilers are foundational to everything we ask our computers to do—applications can only be as efficient and reliable as the underlying compiler stack that translates their logic to machine code. But compiler expertise is a finite resource, and engineers may have to choose whether to prioritize adding optimizations for efficiency or validating their existing features for reliability. This dissertation presents three systems that use lightweight, practical formal methods to push past this tension between performance and correctness. The Diospyros compiler combines an efficient term-rewriting strategy, equality saturation, with translation validation to find correct, fast vectorizations for specialized linear algebra tasks on digital signal processors. The Kani verifier for Rust leverages compiler invariants to improve the performance of dynamically dispatched methods in a satisfiability-solver-based model checker for low-level systems code. Finally, the VeriISLE engine uses annotations to automatically verify machine code generation in Cranelift, a popular production compiler infrastructure for WebAssembly where miscompilation bugs can cause serious security vulnerabilities. In sum, these projects point to a future where formal methods help us build compilers for fast *and* reliable computer systems.

BIOGRAPHICAL SKETCH

Alexa VanHattum was born and raised in Grand Rapids, Michigan, where she graduated from Kentwood Public Schools. She received her Bachelor of Science in Computer Science from Brown University in 2016 and her Master of Science in Computer Science from Cornell University in 2021. Prior to graduate school, she worked on health software at Apple. She is a recipient of the 2020 National Science Foundation Graduate Research Fellowship. She currently lives in Brooklyn, New York, with her dog, Lyra. She will begin as an Assistant Professor of Computer Science at Wellesley College in the fall of 2023.

ACKNOWLEDGEMENTS

First, an unbounded thank you to my collaborators, without whom none of this work would have been nearly as fun (if possible at all). Rachit Nigam brought together the motley crew that would become the Diospyros co-authors: Vincent T. Lee brought us the industry perspective, James Bornholt shared his synthesis expertise and impressive ability to rapidly typeset formal grammars, and Adrian Sampson helped me see that I could be the type of person who understood computer architecture, after all. Rachit himself showed me just how fun writing papers could be with labmates as co-conspirators by your side.

The Kani paper could only have happened after Daniel Schwartz-Narbonne was the best internship host—those countless hours of virtual pair programming to understand tricky Rust compiler internals were the highlight of my summer—and Nathan Chong was the best possible internship manager. I am grateful to them both for letting me stick around to uncover the greatest technical insight of the project weeks after the internship theoretically ended.

The final chapter of this dissertation was only feasible because Chris Fallin made Cranelift such a welcoming open-source project. I cannot wait to see how Chris' (and the other Bytecode Alliance contributors') brilliant technical work to design Cranelift to be amenable to formal reasoning will pave the way to years of groundbreaking work on verification of production systems. I am so thankful to Fraser Brown for her eye toward imparting real-world security impact and her five-dimensional-chess writing insights. An enormous thank you to Monica Pardeshi, who implemented the entirety of the initial annotation language and who bravely kept the project alive when I absconded to teach data structures for two months.

I want to thank my brilliant undergraduate and masters student mentees, who were patient with me as I learned how to be a researcher myself—Katy Voor,

Jacob M. Delgado-López, Jonathan Tran, Jasper Liang, and Alaia Solko-Breslin. I cannot wait to see what you all continue to accomplish. Thank you to the entire CAPRA lab and Gates 407 occupants (including the shared Aeropress) for the years of feedback, snack breaks, and choice Slack memes.

When I was deciding between graduate programs, I received the advice to focus on choosing an advisor who is *kind* above all else; Adrian Sampson is this advice personified. It is almost unfair that he is also absolutely technically brilliant and an unusually fantastic communicator. A huge thank you to Adrian for taking me on as a student when I had functionally zero computer architecture knowledge; and for imparting so many lessons in collaboration, effective communication, and research taste. I am sorry that the experimental peanuts I planted on Adrian's farm did not live to see the completion of this dissertation.

I am endlessly grateful to my other committee members and letter-writers. Andrew Myers was one of the primary reasons I chose Cornell; his courses and feedback have been influential to this work. Nicki Dell was gracious in helping me navigate an Information Science minor from 200 miles away. James Bornholt helped me find my niche and connected me with so many other researchers that impacted the trajectory of this dissertation. Michael Clarkson was the best teaching mentor—the experience of teaching CS 2110, with Michael in my corner, gave me the confidence to know what I want to do with my life (for now, at least). I could not have survived the academic job markets without advice and solidarity from my letter writers and so many others—especially Fraser Brown, Maria Antoniak, Molly Q. Feldman, and my CS-Teaching student job market cohort—thank you all for making such a grueling process also a place of connection and shared accomplishment.

I am grateful to the entire Gates Hall community. Thank you especially to the

Programming Languages Discussion Group for being a captive audience as I honed the ideas in this dissertation (Dexter Kozen, in particular, was the only human to hear my job talk in person before the real deal; his questions are always clarifying forces to be reckoned with). Hakim Weatherspoon helped me see myself as a *systems* person. Thank you to the CIS staff—including Becky Stewart, LeeAnn Roberts, Lacy Jordaens, and Randy Hess—for their herculean efforts in service of the department community.

Thank you to Brown Computer Science—especially Andy van Dam, Tim Nelson, Shriram Krishnamurthi, and the undergraduate TA program—without having stumbled into your community sophomore year, I would never have discovered this field that I adore.

To my friends—your support has kept me well-fed, smiling, and (mostly) sane throughout this half-decade. Kara Fikrig, Phoebe Koenig, Benjamin Hoffman, and Gregory Yauney—cohabitating with you all made Ithaca feel like home, even through a global pandemic. Marianne Aubin Le Quéré, Oliver Richardson, Varsha Kishore, Kate Donahue, Rachit Nigam, Shreyas Malpathak, Soham Sankaran, Maria Antoniak, Gloire Rubambiza, Katie Van Koevering, Tegan Wilson, Kevin Negy, and all of my cohort-mates—navigating the dual mazes of Cornell and academia was a little easier with you beside me. Thank you for the ∞ dinner parties. Molly Q. Feldman was the best programming-languages-to-primarily-undergraduate-institution mentor. Thank you to all the other co-founders and volunteers for Graduate Students for Gender Inclusion in Computing (GSGIC)—the work we did mattered. Kate Storey-Fisher—I am so insanely lucky to have had a friend as extraordinary as you on such a comically parallel yet distinct path over this past decade. Dash, Paige, Noel, Dylan, Katrina, Alisa, Maddie, Marley, Megan, Alina and so many other friends—thank you for encouraging me to go

back to graduate school, after all; for making the trek to Ithaca to see me; and for hearing me out when I occasionally doubted everything. I would be nothing without the community I found in all of you.

Thank you to my parents, Mary and Jeff VanHattum, and my brother Andrew: your enthusiastic support of me pursuing this weird science and math thing—and your endless love and generosity—has made it all worth it.

This material was based upon work in part supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-1650441.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Compilers are foundational to all of computing, but especially to low-level systems software. Compilers must consume an ever-increasing range of new high-level programming languages yet still produce high-performance machine code tailored to the specific target hardware. The resource constraints in the domain of systems programming mean that programmers must be especially aware of how their code interacts with the underlying compiler stack. A compiler must generate machine executables that meet performance demands—it must produce, for example, fast machine code that does not use excess memory. The machine executables must also be correct—they should exactly capture the intention of the high-level program, without introducing any new bugs or unexpected behavior. Conventional wisdom posits a tension between these two goals of performance and correctness.

This tension is typically not explicit—engineers are unlikely to consciously choose slower or more buggy, incorrect systems for their software. Instead, within a given software system, developers must choose priorities: should they focus their effort on validating the implementation they already have, or on designing new optimizations for performance improvements? The implicit choice between correctness and efficiency is also present when engineers choose an implementation language and compiler. How much a given language focuses on specialized performance characteristics is often inversely proportional to how easy it is to write correct code within that language. The choice between prioritizing efficiency or correctness is one of the core challenges within systems programming.

This dissertation pushes on this performance/correctness dichotomy by using practical formal methods. My thesis is that compiler stacks can use lightweight (i.e., mostly automated) formal methods to improve the development process for low-level systems programming—enabling software that is both efficient and correct. The work in this dissertation is designed to be feasible for real-world, production deployments where good performance is a first-order priority.

1.2 Background and Context

Systems programming languages enable efficiency in resource-constrained settings, but the low-level details they expose can make it hard to produce correct, bug-free software. Systems programming consists of writing software that interfaces between application-level programs that users directly touch (e.g., a photo-editing application, a word-processor, a website) and the underlying hardware (e.g., a laptop computer, a higher-powered sever, an embedded systems chip on a mobile device). Systems programming is closely tied to resource consumption within the underlying hardware: typical goals include minimizing run time (the time for a program to run on a specific hardware platform), memory (the storage memory a program needs to execute beyond the input data size), or energy consumption (the amount of electricity needed to run a program on a specific hardware instantiation).

Compiler engineering is an instance of systems programming. Compilers that translate programs to machine code must be intimately tied to the details of the underlying hardware instruction set architecture, or ISA. Because the vast majority of programs are written in high-level languages, the performance characteristics achieved by compilers impact nearly every piece of running software.

Compilers typically translate programs along a series of data representations, starting with formats closer to the source code and moving to increasingly low-level formats closer to the machine code. The data format used for the core analysis of the compiler is called an intermediate representation, or IR. The component of the compiler that translates between the source code and the core intermediate representation is called the front end; the analyses and optimizations that occur on the IR itself make up the mid end; and the final stage of the compiler that translates to machine code is the backend. This dissertation primarily focuses on the mid end and backend of compilers used in production.

Formal methods—rigorous mathematical techniques applied to model real world systems—have a long history in aiding compiler construction and low-level programming. Some of the earliest foundational results in computer science are built on reasoning about program and compiler correctness [McCarthy and Painter(1966), Milner and Weyhrauc(1972), Hoare(1969), Morrisett et al.(1999)]. In 2009, the CompCert compiler for a large subset of the C systems programming language demonstrated that it was possible to fully mechanize (in the proof assistant Coq) a realistic compiler [Leroy(2009a)]. Ongoing research on compiler correctness falls into two primary thrusts: similar efforts for foundational verification of compilers using interactive proof assistants [Leroy(2009b), Tan et al.(2019), Fox et al.(2017), Watt(2018), Ringer et al.(2020)], and more lightweight verification and synthesis of specific compiler components using more *automated* formal methods [Lopes et al.(2015), Brown et al.(2020), Newcomb et al.(2020)]. While foundational verification with interactive proof assistants typically yields a smaller trusted compute base, such efforts often take many person-years of effort from academic experts [Stewart et al.(2015)] and require the compiler to be co-designed

from the ground up with verification in mind. Lightweight techniques trade off the level of assurance with the promise of more feasible applications to *existing* production compiler infrastructures. This dissertation follows in this second vein, with a focus on compilers for systems programming applications where performance is especially critical.

1.3 Contributions and Organization

This dissertation proceeds with three distinct technical chapters, each covering a different software system that leverages lightweight formal methods. Finally, we conclude and discuss future directions.

1.3.1 A Vectorizing Compiler with Equality Saturation

Chapter 2 addresses the question of how lightweight formal methods can be used in embedded systems to produce optimized, correct code without tuning from expert programmers. Formal methods are especially powerful for reasoning about embedded systems, where correctness is difficult to achieve in the presence of complex, domain-specific hardware. Energy-constrained hardware, such as mobile devices, employ specialized processors called *digital signal processors* (DSPs). To maintain low latency with low power usage, DSPs rely on vector instruction set architectures (ISAs) that amortize the cost of a machine instruction (e.g., an `add`) across multiple lanes of distinct values—but require either the programmer or the compiler to group independent instructions. To target a vector ISA, either the program itself or a *vectorizing* compiler needs to group together independent instruction

uses. Traditional vectorizing compilers use heuristics to find such groupings, e.g., by analyzing loops over large matrices. However, some DSP applications require processing small, fixed-size matrices (such as 3×5) that do not match up with the number of vector lanes (e.g., a small power of 2). When existing compilers fail to group these instructions, expert systems programmers manually specify vectorization. The key insight of our compiler, called Diospyros, is to use an automated *equality graph* or *e-graph* solver to find these irregular groups without explicit programmer direction.

In our DSP-vectorization problem, an algorithm that greedily grouped instructions would preclude more optimal groupings. Our solution is instead a set of vectorizing rewrite rules that are fed into an e-graph solver to compare many equivalent solutions. These rules are written once for a specific hardware target, removing the dependency on an expert programmer for each new program and matrix configuration. Users write new programs at a high level without any vector instructions, and Diospyros emits size-specific vectorized code. Diospyros has a translation validation mode that uses an additional solver to show input and output program equivalence. Our empirical evaluation demonstrates that Diospyros produces code that is on average $3.1\times$ faster than existing DSP libraries.

1.3.2 Using Compiler Invariants to Improve Verification of Systems Code

Chapter 3 reverses the relationship between formal methods and systems compilers, asking: can we use compiler invariants to help scale formal methods? Kani is a verifier to check safety properties—like the absence of run-time crashes or assump-

tion violations—in components of production Rust code, including pieces of the Firecracker hypervisor [Agache et al.(2020)]. The key insight of this chapter is that Kani can leverage invariants from the Rust compiler's intermediate representation to improve the correctness and verification latency for code that uses *dynamic dispatch*. Dynamic dispatch is a powerful language design strategy that determines at run time which of several possible functions to invoke—but its use complicates formal reasoning about the program. For example, in most object-oriented languages, dynamic dispatch is used to determine the mostpecific subclass on which to call a method. In Rust, dynamic dispatch is used in conjunction with *trait objects*, which define shared interfaces across different types.

In this chapter, we show (1) that existing Rust verification tools fail to handle all cases of dynamic dispatch, and (2) how to improve verification performance via a new strategy for lowering dynamic calls. Because the Rust standard library and roughly 70% of commonly used Rust libraries use dynamic dispatch, their failure to handle this language feature limits the applicability of otherwise state-of-the-art verification tools for Rust. We describe how to translate dynamic dispatch in Rust to a verification engine and how to substantially improve verification performance. Our open-source tool, Kani, uses the Rust compiler's invariants on trait object creation to limit the number of possible targets to dynamic dispatch calls, improving the verification latency by up to 15× in a case study from the Firecracker hypervisor. This chapter highlights the virtuous cycle between compilers and formal methods research: formal methods help us build better compilers, and compiler expertise can scale the impact of formal-methods-based tools.

1.3.3 Lightweight Verification for Instruction Selection

Chapter 4 addresses how lightweight formal methods can improve the reliability of high-value, high-risk components in large production compiler systems. In this chapter, we verify the rewrite-based instruction lowering of the Cranelift compiler infrastructure. Cranelift is a popular engine that compiles WebAssembly (among other languages) to optimized machine code. Compiler correctness is especially critical in the context of WebAssembly to prevent malicious, untrusted client code from exploiting compiler bugs to break security guarantees [Johnson et al.(2021)]. Cranelift uses a unique approach for selecting ISA instructions: instead of an ad-hoc matching system, Cranelift has an expressive domain-specific language, called ISLE, for polymorphic rewrite rules that use pattern matching to choose ISA instructions.

In this chapter, we present VeriISLE, a framework to verify lowering rules in ISLE. VeriISLE's key selling point is its modularity—VeriISLE includes an annotation language that allows users to add concise semantics of individual terms to be added alongside definitions of the term themselves. We provide annotations to automatically verify rules that cover WebAssembly 1.0 support for integer operations in the ARM `aarch64` ISA backend. We show that VeriISLE can reproduce 3 known bugs (including a 9.9/10 severity security issue), identify 2 previously-unknown bugs and an underspecified compiler invariant, and help analyze the root causes of a new bug. VeriISLE is developed through close collaboration with Cranelift engineers and we are currently working to merge VeriISLE into the production Cranelift repository. To our knowledge, VeriISLE is the first formal verification effort for the instruction-lowering phase of an efficiency-focused production compiler.

1.4 Previously Published Work

The technical material in this dissertation is based on the following previously-published or under-submission papers:

Chapter 2 *Vectorization for Digital Signal Processors via Equality Saturation.* Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, Adrian Sampson. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).
PDF. DOI 10.1145/3445814.3446707.

Chapter 3 *Verifying Dynamic Trait Objects in Rust.* Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP 2022).
PDF. DOI 10.1145/3510457.3513031.

Chapter 4 *Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection.* Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. Under submission to a peer-reviewed conference (April 2023). PDF.

CHAPTER 2

DIOSPYROS: A VECTORIZING COMPILER WITH EQUALITY SATURATION

2.1 Introduction

Compute-heavy embedded sensing applications, from augmented reality to 5G networking, rely on digital signal processors (DSPs). DSPs target power- and energy-constrained domains with real-time performance targets, so their design optimizes for power efficiency over programmability and software compatibility. Their simple in-order cores help meet strict real-time deadlines but also mean that unoptimized code performs poorly. Unlike modern superscalar CPUs, DSPs cannot afford complex hardware with out-of-order execution that would automatically extract good performance even from simple sequential code. For performance, DSP architectures instead expose Very Long Instruction Word (VLIW) and vector instruction sets with exotic architecture-specific extensions. These instruction sets offload the burden of parallelization onto the compiler and programmer.

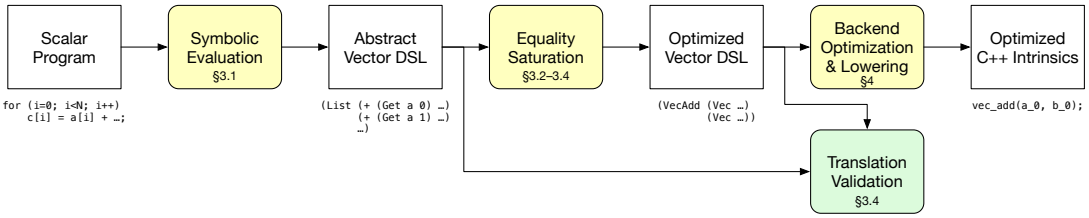

Figure 2.1: The Diospyros compiler workflow. Diospyros first lifts scalar input programs into a high-level DSL via symbolic evaluation. Diospyros then searches for equivalent optimized programs using equality saturation. Finally, Diospyros lowers the optimized program to C++ with target-specific intrinsics for compilation with a DSP toolchain.

DSP applications typically rely on on two categories of computational *kernels*[1]: (1) large-scale kernels operating on high-dimensional data (much larger than the machine's vector width), and (2) small-scale kernels operating on low-dimensional data (on the order of the vector width). In an industrial context, the distribution of kernels tends to be bimodally distributed: many have small dimensionality (~3–6), and the remaining are much larger (~100–1000). While compiler toolchains and vendor libraries for DSPs often focus their attention on large-scale kernels—shipping linear algebra libraries tuned for large, dense operations—small-scale kernels still consume a non-trivial portions of the end-to-end performance of many emerging DSP applications. Some DSP applications are bottlenecked by small-scale kernels as part of the "last mile" of a larger computation. In other words, a variety of small kernels impose an Amdahl limitation [Yavits et al.(2014), Eyerman and Eeckhout(2010), Paul and Meyer(2007)] that yields diminishing returns from speeding up just the large-scale loops. Other applications, such as simultaneous localization and mapping (SLAM) [Mur-Artal et al.(2015), Mur-Artal and Tardós(2017), Sumikura et al.(2019), Strasdat et al.(2011)] and structure from motion [Sweeney(2016)], have many components that are dominated entirely by small-scale kernels.

Compiling efficient small-scale kernels is challenging even for state-of-the-art compiler techniques because the best performance requires complex data movement strategies that are beyond the scope of most automatic vectorization. Moreover, DSP architectures are extremely diverse: they offer per-application instruction set customization and can even support custom proprietary ISA extensions [Gonzalez(2000)]. As a result, DSP engineers still man-

[1]Here, we define a kernel to be a function that consumes one or more multidimensional input matrices and produces one or more multidimensional output matrices. A kernel can be implemented as multiple nested source-level functions.

ually apply device- and kernel-specific optimizations by hand-writing vector intrinsics [Yotov et al.(2003), Mainland et al.(2013), Alvanos and Trancoso(2016)]. This manual effort does not scale with the plethora of kernels and target architectures. For example, products and convolutions of small 3×3 and 4×4 matrices are commonplace in various machine perception applications, but the most efficient implementations for these two sizes are very different. Specialized kernels for each size can vastly outperform general implementations in linear algebra libraries [Spampinato et al.(2018), Kyrtatas et al.(2015)].

This chapter designs a compiler, Diospyros, that aims to compete with manual tuning by DSP experts while baking in minimal assumptions about the target hardware. Diospyros frames compilation as a search problem in a space of candidate programs. It defines this search space using a system of rewrite rules that encompass both high-level functional specifications and low-level device-specific instructions. Crucially, the resulting program space includes implementations that use arbitrary indexing to express complex data movement patterns. Unlike traditional approaches to general-purpose vectorization [Larsen and Amarasinghe(2000)], Diospyros focuses on using the *shuffle* and *select* instructions common in DSPs to implement the irregular data movement necessary to pack as much work as possible into vector lanes.

Figure 2.1 shows the Diospyros compilation workflow. Diospyros takes a program in a scalar, imperative language and lifts it to a high-level vector DSL using symbolic evaluation. The core optimization engine is an exhaustive search in a restricted space of candidate programs from this DSL using *equality saturation* [Joshi et al.(2002), Tate et al.(2009), Willsey et al.(2021)]. Most compilers apply rewrite rules in a fixed order, which offers predictable compilation but sac-

rifices optimality. Equality saturation effectively applies *all* rewrite rules simultaneously by representing the input program as an E-graph [Nelson(1980)] and performing congruence closure using the rewrite rules as an equivalence relation. The saturated E-graph compactly represents the entire space of candidate programs, from which Diospyros can extract the most efficient one according to an abstract cost model. After extracting the optimal program, Diospyros lowers it to C vector intrinsics for code generation via a backend DSP compiler.

We implement Diospyros to target Tensilica DSPs and show that it can compile kernels that outperform optimized library functions from the Tensilica SDK by a geometric mean speedup of 3.1×. Compared to one expert-written kernel hand-tuned for a fixed matrix size, Diospyros produces code within 8% of the expert performance within 2.2 seconds of compilation time. To show that Diospyros-compiled kernels offer end-to-end speedups on realistic applications, we integrate them into code from Theia [Sweeney(2016)], an open-source computer vision library for structure from motion (SFM). The Diospyros version of this application performs 2.1× faster on our selected functionality than Theia's original implementation, which uses the Eigen template library for linear algebra [Guennebaud et al.(2010)].

This chapter's contributions include: (1) a strategy for using symbolic evaluation and equality saturation to search for SIMD implementations of high-level specifications, (2) Diospyros, an end-to-end compiler design that uses the rewrite system to optimize computational kernels for DSP architectures, and (3) an evaluation on a range of realistic DSP computations and a commercial DSP target showing performance improvement over optimized baselines.

2.2 Motivating Example

This section shows how an example DSP kernel poses challenges to traditional compilers and how hardware-specific manual tuning can outperform them. We give an overview of how Diospyros's design can mimic the hand-tuning process.

Consider optimizing a fixed-size matrix convolution for a DSP. Embedded DSP applications typically rely on specialized kernel implementations for fixed, small data sizes—for example, a convolution with a 3×5 input matrix and a 3×3 filter:

```
for (oRow = 0; oRow < 5; oRow++)
  for (oCol = 0; oCol < 7; oCol++)
    for (fRow = 0; fRow < 3; fRow++)
      for (fCol = 0; fCol < 3; fCol++) {
        fRT = 3 - 1 - fRow; fCT = 3 - 1 - fCol;
        iRow = oRow - fRT; iCol = oCol - fCT;
        if (iRow >= 0 && iRow < 3 &&
            iCol >= 0 && iCol < 5)
          o[oRow][oCol] += in[iRow][iCol] * f[fRT][fCT];
      }
```

The outer loops run 5 and 7 times because they iterate over the output matrix. This convolution "pads" the input matrix at the boundaries and produces a slightly larger output matrix.

In this example, we will optimize this convolution for the Tensilica Fusion G3 DSP [Cadence Design Systems, Inc.(2020)], which has a 4-wide floating-point SIMD vector unit. SIMD instructions are critical in DSP programming for both performance and efficiency: they both enable parallelism and amortize the energy

cost of fetching and dispatching instructions. While statically specifying the sizes allows Tensilica's vectorizing compiler to improve on this naive `for`-loop-based implementation by 1.6×, the best implementation we have found with Diospyros uses machine-specific vector intrinsics to achieve a further speedup of 22.9×. We explore why and how this gap arises in general for this category of DSP kernels, where the problem dimensions are close to the vector width. Namely, for these kernels, boundary conditions make up a large proportion of the kernel's work, which hinders straightforward approaches to parallelization.

Traditional automatic parallelization. Two commonplace compiler techniques for vectorizing sequential code are loop-level vectorization and superword-level parallelism (SLP) optimizations [Larsen and Amarasinghe(2000)]. For 2D convolution, the index math for transposing the filter (`fRT` and `fCT`) and the `if` for the boundary conditions pose a problem to loop-level vectorization. While loop-level vectorization works well when the data dimensions are large enough that there is a steady state that admits processing in 4-wide chunks, smaller loops do not have such a steady state. In this convolution example, no loop executes more times than twice the vector width—so every loop iteration is a boundary condition.

Because the array sizes for our problem are fixed, a compiler could unroll the loops and apply non-loop vectorization techniques such as SLP [Larsen and Amarasinghe(2000)]. And indeed, specializing the array sizes leads to the aforementioned 1.6× speedup over a version with variable array sizes. However, this approach still leaves some performance on the table. Because the matrix dimensions (3×5 and 3×3) are close to the machine's vector width (4), SIMD instructions do not apply "cleanly" to the input arrays. Furthermore, the

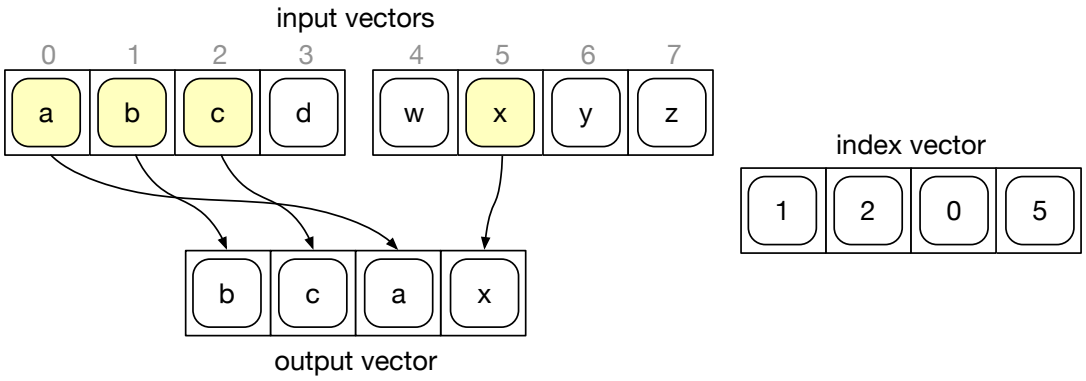

Figure 2.2: An ISA-specific shuffle instruction that takes three arguments—two input vectors and an index vector—and produces a single output vector with the specified combination of values. Experts can use similar instructions to orchestrate complex data movement strategies.

memory accesses to `f` are not contiguous, meaning that a simple vector load will not suffice to enable vectorized arithmetic. The Tensilica compiler's vectorization pass fails to find perfectly aligned runs of 4 identical operations, and it does not attempt to gather or shuffle disparate values to fill a vector. Alternatively, the `if` for the boundary condition means that a straightforward vectorized version will need to use predicated operations, wasting some potential computation bandwidth. Traditional vectorization optimizations rely on regularity in data movement and computation that is not present in specialized DSP kernels like this one, where loops are imperfect and data sizes are not much larger than the vector width.

Hand tuning. Instead, an expert programmer can use the Fusion G3's special instructions for data movement to pack computation into the vector lanes. The DSP supports gather/scatter and shuffle operations that pack data irregularly into vector registers for subsequent regular processing. For example, this intrinsic call:

```
int indices[4] = {1, 2, 0, 5};
xb_vecMx32 vec3 = PDX_SEL_MX32(vec1, vec2, indices);
```

computes a new 4-wide vector value by selecting specific hard-coded indices from the concatenation of two other vectors, `vec1` and `vec2`, as illustrated in Figure 2.2. The programmer can use this strategy to implement tactics for gathering data to fill vector lanes for later computation, like this multiplication:

```
xb_vecMx32 vec4 = PDX_MUL_MX32(vec1, vec3);
```

With judicious use of vector intrinsics and manual derivation of index operands, an expert implementation can surmount the limitations of traditional auto-vectorization. A manually tuned kernel can be an order of magnitude faster than the automatically parallelized version. However, the tuning required is specific to both the Fusion G3 target and the specific specialized size of the convolution kernel. A different vectorization strategy with completely different shuffle indices will be optimal for a 4×4 filter, for example.

Vectorization via rewriting. Diospyros uses term rewriting to search for DSP vectorization strategies that exploit this kind of irregular data layout techniques to optimize for vector unit utilization. Our system starts with an imperative reference implementation and, using symbolic evaluation (Section 2.3.1), extracts a specification describing the value to compute for each element of the kernel's output(s). For our convolution example, the specifications for the first four values of the output matrix are:

$$\boxed{i_{0,0} \times f_{1,1} + i_{0,1} \times f_{1,0} + i_{1,0} \times f_{0,1} + i_{1,1} \times f_{0,0}}$$

$$\boxed{i_{0,0} \times f_{1,2} + i_{0,1} \times f_{1,1} + i_{0,2} \times f_{1,0} + i_{1,0} \times f_{0,2} + i_{1,1} \times f_{0,1} + i_{1,2} \times f_{0,0}}$$

$$\boxed{i_{0,1} \times f_{1,2} + i_{0,2} \times f_{1,1} + i_{0,3} \times f_{1,0} + i_{1,1} \times f_{0,2} + i_{1,2} \times f_{0,1} + i_{1,3} \times f_{0,0}}$$

$$\boxed{i_{0,2} \times f_{1,2} + i_{0,3} \times f_{1,1} + i_{0,4} \times f_{1,0} + i_{1,2} \times f_{0,2} + i_{1,3} \times f_{0,1} + i_{1,4} \times f_{0,0}}$$

Here, the first expression is smaller because of the kernel's boundary condition. Diospyros uses a term rewriting system to find vectorization opportunities across these mathematical expressions. For example, the `vec_multiply_accumulate` rule can apply here to show that the above outputs are equivalent to expressing the last product in each element as a fused multiply–accumulate vectorized operation, `VecMAC`:

```
(VecMAC (...)
  (Vec (Get I 6) (Get I 7) (Get I 8) (Get I 9))
  (Vec (Get F 0) (Get F 0) (Get F 0) (Get F 0)))
```

`Vec` and `Get` are ISA-agnostic data movement abstractions that represent accessing the specified indices of a memory (with 2D arrays flattened to 1D access). Our full vector domain specific language is described in Section 2.3.1 and shown in Figure 2.3.

Due to the commutativity and associativity of $+$ and $\times$, there are many possible shuffles a programmer could use to generate valid `VecMAC` operations. Diospyros uses an equality saturation approach to consider many possible shuffles—rather than applying destructive rewrites, as a traditional compiler would—and selects the pattern best suited to an abstract model of our architecture's data movement instructions. For example, here each `Vec` references the elements of only a single input array, which can be implemented with in-register data movement.

When targeting the Fusion G3, Diospyros produces this code for the vectorized expression:

```
shuf_I = PDX_SEL_MX32(I_4_8, I_8_12, [6, 7, 8, 9]);
shuf_F = PDX_SHUF_MX32(F_0_0, [0, 0, 0, 0]);
PDX_MAC_MFX32(out_0_4, shuf_I, shuf_F);
```

The full implementation that Diospyros generates for this problem size is 22.9× faster than a naive fixed-size implementation and 4.5× faster than an optimized vendor library kernel.

2.3 Rewriting for Vectorization

Our core vectorization formulation uses equality saturation [Tate et al.(2009)] to search for optimized implementations. This section describes the optimization workflow. Programmers write an imperative reference implementation using scalar operations, symbolic evaluation lifts this to an abstract vector DSL, then Diospyros searches for an optimal vectorized program using an equality saturation engine—trading off efficiency and completeness in the search. Next, Section 2.4 shows how Diospyros compiles the optimized program back to the imperative DSL to produce efficient code for the DSP target.

2.3.1 Defining and Lifting Specifications

Diospyros takes as input scalar programs written in a simple imperative language with first-class matrix and vector objects and operations, implemented as an embedded Racket DSL. For example, this code specifies a simple vector-vector add:

```
(define (vector-add-spec A B n)
  (vec-decl 'A n 'input)
  (vec-decl 'B n 'input)
  (define C (make-vector n))
```

```
(for ([i n])
  (vector-set! C i
    (add (vector-ref A i)
         (vector-ref B i))))
C)
```

Here, `A` and `B` are vectors of input data and `n` is a compile-time parameter that determines the input size.

This input language is both convenient to write and straightforward to compile to executable code for use in validation or testing. It supports arbitrarily complex indexing expressions and control flow, as long as they are independent of the input data. The input language provides the usual scalar arithmetic operations, such as `+`, but users can also define custom scalar functions to reflect a given target DSP and application.

While we could optimize this language directly (in the spirit of Denali [Joshi et al.(2002)]), doing so would conflate details of the imperative implementation with the underlying abstract mathematical computation. To focus on the latter and simplify the search, Diospyros first *lifts* imperative input programs into a mathematical representation. It symbolically evaluates the input program using Rosette [Torlak and Bodik(2014)], which extends Racket DSLs with symbolic evaluation support.

The symbolic evaluation step produces an expression in Diospyros's vector DSL, shown in Figure 2.3. The vector DSL includes both scalar and vector versions of common arithmetic operations ($+$, $-$, $\times$, etc.), as well as operations to initialize vectors with literals or variables and to extract individual vector lanes. The lifting

$\langle prog \rangle$ ::= (`List` $\langle expr \rangle^+$) | $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle scalar \rangle$ | $\langle vector \rangle$

$\langle scalar \rangle$ ::= $\langle integer \rangle$ | $\langle variable \rangle$
| (`+` $\langle scalar \rangle$ $\langle scalar \rangle$) | (`-` $\langle scalar \rangle$ $\langle scalar \rangle$) | (`*` $\langle scalar \rangle$ $\langle scalar \rangle$)
| (`/` $\langle scalar \rangle$ $\langle scalar \rangle$) | (`sgn` $\langle scalar \rangle$) | (`sqrt` $\langle scalar \rangle$) | (`-` $\langle scalar \rangle$)
| (`Get` $\langle variable \rangle$ $\langle integer \rangle$) | ($\langle func \rangle$ $\langle scalar \rangle^*$)

$\langle vector \rangle$ ::= (`Vec` $\langle scalar \rangle^+$) | (`Concat` $\langle vector \rangle$ $\langle vector \rangle$)
| (`VecAdd` $\langle vector \rangle$ $\langle vector \rangle$) | (`VecMinus` $\langle vector \rangle$ $\langle vector \rangle$)
| (`VecMul` $\langle vector \rangle$ $\langle vector \rangle$) | (`VecDiv` $\langle vector \rangle$ $\langle vector \rangle$)
| (`VecMAC` $\langle vector \rangle$ $\langle vector \rangle$ $\langle vector \rangle$) | (`VecSgn` $\langle vector \rangle$)
| (`VecSqrt` $\langle vector \rangle$) | (`VecNeg` $\langle vector \rangle$)

$\langle func \rangle$ ::= $\langle symbol \rangle$

Figure 2.3: Diospyros's vector DSL. A top-level program is a (possibly singleton) list of outputs. Expressions operate over both scalars and vectors.

process, however, only produces the scalar subset of the language—the rewriting system in the next section will use the vector constructs. Lifting supports calls to user-defined functions by introducing uninterpreted functions. The same symbolic evaluation engine also powers the translation validation tool that Diospyros uses to verify its output (see Section 2.3.4).

To expose vectorization opportunities for the rewriting system, the lifting process converts matrix and vector outputs into a single `List` output term, with one element for each value in the program output. For example, the vector-vector add above with `n` = 2 lifts to this expression:

```
(List
  (+ (Get a 0) (Get b 0))
  (+ (Get a 1) (Get b 1)))
```

Here, `Get` is list access and `List` constructs a new output list holding the two

elements of the output vector.

2.3.2 Rewriting Strategy

To vectorize the lifted program in the abstract DSL, Diospyros uses a family of built-in (though user-extensible) rewrite rules. The key equivalence that enables vectorization is that the rewrite rules consider a `List` to be equivalent to a concatenation of fixed-size vectors. For example, Diospyros can rewrite our vector-vector add with `n` = 4 and a vector width of two this way:

```
(List (+ (Get a 0) (Get b 0))
      (+ (Get a 1) (Get b 1))
      (+ (Get a 2) (Get b 2))
      (+ (Get a 3) (Get b 3)))
⇝
(Concat (Vec (+ (Get a 0) (Get b 0))
             (+ (Get a 1) (Get b 1)))
        (Vec (+ (Get a 2) (Get b 2))
             (+ (Get a 3) (Get b 3))))
```

`Vec` constructs a vector from a configurable machine–width number of scalar values (here, two), and `Concat` concatenates two vectors into a list. In real DSP code, they correspond to vector load and store instructions (see Section 2.4). Diospyros's rewrite rules can pad lists with zeros if their lengths are not a multiple of the vector width.

This rewriting into vector-sized chunks creates opportunities to use vectorized computation. The rewrite system finds Vec expressions that contain similar scalar

expressions and replaces them with their vectorized equivalents. For example, the rule for introducing vectorized add instructions, `VecAdd`:

$$(\texttt{Vec}\ (\texttt{+}\ a\ b)\ (\texttt{+}\ c\ d)) \rightsquigarrow (\texttt{VecAdd}\ (\texttt{Vec}\ a\ c)\ (\texttt{Vec}\ b\ d))$$

applies twice to the example above, producing:

```
(Concat (VecAdd (Vec (Get a 0) (Get a 1))
                (Vec (Get b 0) (Get b 1)))
        (VecAdd (Vec (Get a 2) (Get a 3))
                (Vec (Get b 2) (Get b 3))))
```

Here, the indices in the `Get` expression determine the data movement strategy required for this program. In this case, the pairs of indices `0, 1` and `2, 3` can each be implemented by a vector load without additional data movement. This example is now fully vectorized because all `Vec` expressions contain simple memory lookups and no scalar computations expressions remain.

Diospyros's code generation backend (Section 2.4) produces DSP code from this vectorized program by emitting C intrinsics resembling this pseudocode:

```
vecreg a_0_2 = load(a, 0, 2);
// ...
vecreg b_2_4 = load(b, 2, 2);

vecreg add_1 = vec_add(a_0_2, b_0_2);
vecreg add_2 = vec_add(a_2_3, b_2_4);

store(out, add_1, 0, 2);
store(out, add_2, 2, 2);
```

While this simple example has perfectly aligned vector accesses, most realistic code requires nontrivial data movement to fill the vector registers. Diospyros's backend consumes these `Vec` expressions to produce actual loads and data movement instructions based on the high-level strategy found by the rewrite engine. During code generation, the backend selects vector shuffle code to implement each given `Vec` expression. Similarly, real code mixes both vector and scalar computation; Diospyros generates a mixture of both.

2.3.3 Searching for Rewrites

In general, applying the rewrite rules directly (like a traditional compiler) does not promise optimality—we must be sure to apply the right rules in the right order to find the optimal program (with respect to our rule set). This section describes how Diospyros searches the space of all rewrite rule applications by representing the lifted program as an *equality graph* (*E-graph*) [Nelson(1980)] and using *equality saturation* [Tate et al.(2009)] for efficient search.

Equality saturation. An E-graph is a data structure for efficiently representing a large set of terms and equivalences between them. The nodes of an E-graph are function symbols or terminals, and subgraphs represent terms. Each node is associated with an equivalence class, and the E-graph guarantees that two nodes are in the same equivalence class if and only if the program terms rooted at them are equivalent. When used for program optimization, the equivalence relation is program equivalence.

Initially, the E-graph represents only one program and its sub-terms (the in-

(a) Before rewriting (b) After rewriting

Figure 2.4: An E-graph before and after applying a rewrite rule for fused multiply–accumulate. Solid boxes are nodes and represent program terms. Dashed boxes represent equivalence classes. After rewriting, the VecAdd and VecMAC terms are in the same equivalence class.

put program in the abstract DSL). Equality saturation then applies rewrite rules (program transformations) to the E-graph, which introduces new nodes into the graph and annotates them with the appropriate equivalence classes to maintain congruence. For example, this is a rewrite rule for fused multiply–accumulate:

$$(\texttt{VecAdd}\ a\ (\texttt{VecMul}\ b\ c)) \longleftrightsquigarrow (\texttt{VecMAC}\ a\ b\ c)$$

Figure 2.4 illustrates the application of this rewrite rule to an E-graph which initially represents the program (`VecAdd v1 (VecMul v2 v3)`). Applying the rule introduces a new `VecMAC` node into the graph, with the variables `v1`, `v2`, and `v3` as children, and adds the new node to the equivalence class of the existing `VecAdd` node.

Equality saturation iteratively applies all rewrite rules (possibly multiple times), terminating when no potential rewrite rule application would change the graph—the graph has *saturated*—or a timeout is reached. At this point (unless the timeout is reached), the saturated E-graph represents all programs that could be produced by applying the rewrite rules in any order. This property allows us to avoid the phase ordering problem common to compilers.

We use the egg [Willsey et al.(2021)] library for E-graphs and equality satura-

tion. In egg, a rewrite rule comprises two parts: a *searcher* that looks for nodes that can be rewritten, and an *applier* that applies a rewrite. egg exposes a pattern DSL to specify simple syntactic rewrites and a Rust API to implement custom searchers and appliers with more complex logic than simple pattern matching.

Custom matching for vectorization. Simple unary scalar operations can be vectorized using rules of the form shown in Section 2.3.2. However, DSP kernels often do not fit exactly within the target architecture's vector lanes (for example, a 3×3 matrix multiply on an architecture with vector width 4). To vectorize operations while maximizing hardware utilization, Diospyros provides rewrite rules that work even when some lanes of a vector computation are empty. For example, the following concrete rewrite is sound and enables vectorizing an addition with irregular shape:

$$(\texttt{Vec}\ (\texttt{+}\ a\ b)\ 0\ (\texttt{+}\ c\ d)\ 0) \rightsquigarrow (\texttt{VecAdd}\ (\texttt{Vec}\ a\ 0\ c\ 0)\ (\texttt{Vec}\ b\ 0\ d\ 0))$$

To avoid specifying every permutation of zeros on the left-hand side of this rule, and repeating this specification for each binary operation, Diospyros uses egg's support for custom rewrite rules that go beyond pattern matching. The custom rule first matches on the outer vector and then identifies whether each lane matches either the operator pattern $(\langle op \rangle\ x\ y)$ or chosen concrete values (in this case, a constant zero). Using these custom rules makes it easier to extend Diospyros with DSP-specific instructions without developing a comprehensive new rewrite rule family.

Associativity & commutativity. A common challenge in rewrite systems is handling operators that are associative or commutative (or both). For example,

we want this rewrite:

$$(+\ (+\ a\ b)\ 0) \leftrightsquigarrow (+\ a\ b)$$

to also apply to associative or commutative variants of the LHS such as (+ a (+ b 0)). But applying associative and commutative variants of such rules to saturation dramatically increases the size of an E-graph; the decision problem of whether two terms can be unified modulo associativity and commutativity (the *AC-matching* problem) is NP-complete [Benanav et al.(1987)]. This theoretical problem is also a scalability challenge for equality saturation in practice [Nandi et al.(2020)].

Diospyros addresses AC-matching by optionally allowing users to disable associativity and commutativity rules during saturation. This approach sacrifices completeness in terms of missing some potential rewrites, but reduces memory requirements and thus allows Diospyros to compile kernels with deeper syntax trees over associative and commutative operators. To regain some of the power of associativity and commutativity, we use more complex rewrite rules to selectively re-enable some limited forms of AC rules that we have found to be profitable in practice.

For example, consider the following 4-wide vector:

```
(Vec (+ a0 (* b0 c0))
     (+ a1 (* b1 c1))
     (+ a2 (* b2 c2))
     (+ (* b3 c3) a3))
```

We would like to optimize the scalar operations in this vector into a single vectorized multiply–accumulate. However, without a general commutativity rule for `+`, the fourth lane prevents introducing a `VecMAC` operation. We work around

this limitation using a custom searcher that matches on each lane independently with one of several pattern options, and then combines the results. For vector multiply–accumulate, each lane must match one of these patterns:

$$(+\ a\ (*\ b\ c)) \qquad (+\ (*\ b\ c)\ a) \qquad (*\ b\ c) \qquad 0$$

The applier (right-hand side) of this rule collects the arguments into vectors (mapping "missing" values to zero) and applies the fused operation:

```
(VecMAC (Vec a0 a1 a2 a3)
        (Vec b0 b1 b2 b3)
        (Vec c0 c1 c2 c3))
```

Unlike an approach that includes AC rules when saturating the E-graph, this custom searcher approach does not persist its discovered equivalences. This difference trades off memory for compute: rather than persisting these equivalences in the E-graph, we re-compute them every time we try to apply the custom searcher. In practice, we have found this to be a worthwhile trade-off, allowing larger kernels that previously exhausted the memory of a 512 GB host to successfully compile. We expect that similar customizations for AC searching would be beneficial in a variety of domains beyond vectorization.

Floating point accuracy. Diospyros's rewrite rules are correct with respect to the real numbers. They do not adhere to strict floating point semantics which, for example, would not allow associativity in addition or multiplication. Diospyros shares this characteristic with other modern optimizing compilers for compute kernels that prioritize speed over numerical stability [Ragan-Kelley et al.(2013a), Kamil et al.(2016)]. We measure floating point error in our evaluation (Section 2.5) and find Diospyros-generated code to match reference implementations within sev-

eral decimal places.

2.3.4 Extraction

After equality saturation completes, Diospyros has a single E-graph representing many programs that are equivalent to the input program (according to the rewriting system). Each program would be a valid solution to the compilation problem, but we want to extract the most efficient solution. We cannot explicitly enumerate the programs to search for an optimal one—doing so would sacrifice the compactness of the E-graph representation. Prior equality-saturation-based superoptimizers [Joshi et al.(2002)] extract efficient code by generating cost-related verification conditions from the E-graph and discharging them with a SAT solver, but this requires a detailed architecture-specific cost model.

Diospyros extracts an efficient solution from the E-graph using a cost model that assigns a fixed cost to each operator in the vector DSL. This cost model reflects the time and energy savings of vectorization as well as the cost of reading values from registers versus memory. To support efficient extraction from the E-graph (linear in the number of E-graph nodes rather than the number of candidate programs), this cost function must be strictly monotonic, i.e., an expression's cost is greater than the sum of the costs of its subexpressions. This limitation makes extraction efficient because it avoids the need to explore all the zero-cost variants of a candidate expression. While this restriction limits the cost models Diospyros can express, in our experience we can still extract fast programs, as Section 2.5 demonstrates.

Our cost model for data movement is intentionally high-level—Diospyros as-

signs a lower cost to shuffles that gather data from a single input array (or zeros) than to shuffles across different inputs or non-zero scalars. The Fusion G3's fast, unrestricted shuffle instruction allows this abstract cost model to serve as a good proxy for data movement costs. This approach may be a poorer fit for architectures without support for flexible shuffles (Section 2.6 discusses this limitation further).

Timeouts. Saturating an E-graph guarantees that it captures all possible orderings of the rewrite rules. In practice, saturation can be very expensive, and so we impose both a wall-clock timeout and an E-graph node limit to terminate early. Diospyros can still produce a solution from a timed-out compilation by applying the above extraction process to the partially saturated E-graph. Half of our benchmarks in Section 2.5 time out, and yet most still outperform optimized libraries. Section 2.5.5 studies the impact of timeouts on the quality of Diospyros's output.

Translation validation. Diospyros depends on a set of rewrite rules to define the search space of equivalent programs. The equality saturation engine trusts these rules; while most rules are simple, an incorrect one can cause Diospyros to miscompile a program. We address this risk by re-using the symbolic evaluation engine from Section 2.3.1. We use this engine to optionally perform translation validation on the final extracted program, using Rosette [Torlak and Bodik(2014)] to prove that the extracted program is equivalent to the scalar input program for all possible inputs.

The validation assigns no semantics to the uninterpreted functions that represent user-defined functions, so programs involving them may produce spurious validation failures (for example, we would not know that a user-defined `square` function only produces non-negative values). The user can optionally provide

(possibly partial) semantics for user-defined functions by writing a Racket function, which Rosette lifts to operate on symbolic inputs and uses to validate translations. Diospyros uses these semantics only at the translation validation stage and not within the rest of the compiler.

Translation validation removes the equality saturation engine and the rewrite rules themselves from the trusted computing base of the compiler. However, the validation is between two programs in the vector DSL, and so both the initial lifting from imperative code into that DSL and the backend code generation (Section 2.4) are still trusted. Diospyros's translation validation models values in the theory of real arithmetic, rather than with precise floating point semantics. Anecdotally, we have found translation validation useful when developing and debugging new rewrite rules and vector DSL extensions.

2.4 Lowering and Code Generation

After extraction from the E-graph, we are left with a vectorized program in an idealized vector DSL. This section describes how Diospyros compiles this program, first to a lower-level vector IR and then to C++ specific to the target DSP architecture.

Abstract vector IR. To capture the essence of vector computation with data movement, the Diospyros backend defines a machine-independent vector intermediate representation (IR). At this abstraction level, kernels operate on user-specified input arrays to produce outputs using an imperative language free of control flow. The IR includes common vectorized operations such as memory loads and stores,

arithmetic, and data shuffles, as well as user-defined uninterpreted functions for both scalar and vector operations. While the IR is at a fairly low level of abstraction, it abstracts away concrete details of the DSP architecture, deferring them to a later architecture-specific instruction selection phase (Figure 2.1).

One key challenge to solve at this compilation step is how to translate instances of `Vec` in the vector DSL. `Vec` terms represent vector initializations, and each vector lane can be populated from an arbitrary memory location. For example, the quarternion product benchmark we evaluate in Section 2.5 includes a `Vec` term in its output of the form:

```
(Vec (Get a 1) (Get a 2) (Get a 0) (Get a 3))
```

To initialize this vector, the backend IR includes a vector *shuffle* operation:

```
(vec-shuffle inputs indices)
```

that takes as input an array of `indices` defining where to move each element of `inputs`. The IR does not restrict the possible values of `indices`, offering the flexibility to compile vectorization patterns discovered by equality saturation that require complex data movement. Lowering this instruction to the target DSP architecture requires selecting an instruction sequence that achieves this desired movement using the architecture's available shuffle operations.

IR-level optimization. Diospyros's compilation flow includes fully unrolling loop nests, which can create extraneously large programs with redundant terms. This redundancy is not an issue during equality saturation, because the E-graph representation implicitly de-duplicates redundant terms. However, a naive lowering from the high-level vector DSL would include this redundancy and produce kernels

far too large for resource-constrained targets. The Diospyros backend implements a local value numbering (LVN) pass to eliminate redundant terms. This pass is highly effective: for the quarternion product benchmark in Section 2.5, it reduces the output size from over 100,000 lines of C++ to under 500 lines.

Instruction selection. The final phase of compilation is to perform instruction selection for a concrete architecture. Diospyros delegates much of this work to the vendor-supplied DSP compiler toolchain, avoiding the need to integrate deep target-specific knowledge into Diospyros for each new DSP target architecture. The lowering phase translates the low-level IR into C++ compiler intrinsics that are then compiled with the DSP toolchain. The programmer can provide the name and type signature of target-specific instructions for both scalar and vector operations.

2.5 Evaluation

Our evaluation has two main components: a demonstration of speedups for individual kernels compiled with Diospyros (Section 2.5.4), and a more detailed examination of an application that can benefit from replacing library calls to fixed-sized linear algebra kernels with Diospyros kernels (Section 2.5.7).

2.5.1 Implementation

Diospyros currently targets Tensilica's Xtensa Fusion G3 family of DSP architectures [Cadence Design Systems, Inc.(2020)]. The backend lowers the `vec-shuffle`

instruction in the low-level IR to the Xtensa `PDX_SHFL_MX32` (single-register shuffle) and `PDX_SEL_MX32` (two-register select) intrinsics. To implement arbitrary shuffles with more than two registers, Diospyros uses nested select instructions.

Diospyros's implementation spans two languages. 4,800 lines of Racket, using the Rosette framework [Torlak and Bodik(2014)], implement the domain-specific vector languages, lifting, translation validation, and backend compilation phases. 1,400 lines of Rust implement the rewrite rules and cost model using the egg [Willsey et al.(2021)] equality saturation library.

2.5.2 Methodology

We report cycle counts from Tensilica's cycle-level simulator for the Fusion G3 DSP processor [Cadence Design Systems, Inc.(2020)], `xt-run`. We use use `xt-run`'s default memory model, which assumes an ideal, unit-delay memory for all accesses. The simulator is deterministic, so we report results for a single execution. We compile all implementations (baseline loops, library-provided functions, and Diospyros-generated code) with the `xt-xcc`/`xt-xc++` compiler from the Tensilica Xtensa SDK at the highest optimization level, `-O3`.[2] We run experiments on a machine with two Intel Xeon E5-2620v4 CPUs running CentOS 7.6. We give Diospyros a 3-minute timeout for equality saturation with a node limit of 10,000,000. We run without full associativity and commutativity enabled (as described in Section 2.3.3).

[2]Tensilica also provides a second compiler, called `xt-clang++`, that is not well-documented in our version of the Xtensa SDK. Xtensa specifies that `xt-clang++` does not include a loop transformation framework, such as the one in `xt-xc++` at the `-O3` optimization level; however, it does perform better on some scalar code due to more aggressive inlining and a different software pipelining scheduler. We use the better documented, default `xt-xc++` compiler.

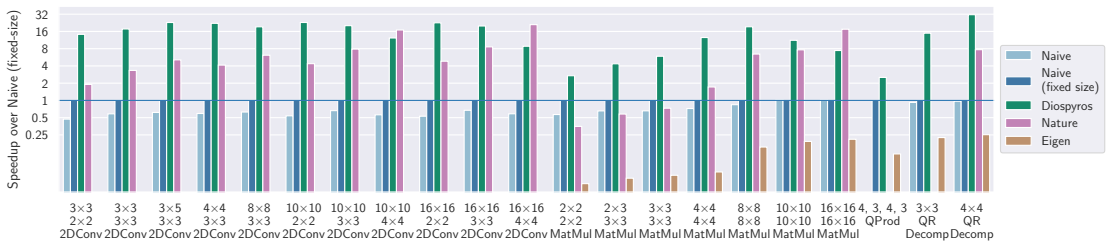

Figure 2.5: Speedup over Naive (fixed size) in simulated cycles, log scale. Bars above the blue line indicate a speedup. *Naive* is a naive loop nest, *Naive (fixed size)* is a loop nest with fixed bounds, *Diospyros* is our system, *Nature* is a vendor-supplied library function, and *Eigen* is a C++ template linear algebra library.

We compare Diospyros with the Nature DSP library included with Tensilica's SDK. Nature is optimized specifically for the Fusion G3 using vector intrinsics, so it performs better than naive C++; however, the library's performance is limited by the need to be generic over matrix sizes. Not all sizes have Nature comparisons because the library often restricts dimensions to multiples of 4 to match the machine vector width. We also compare with Eigen [Guennebaud et al.(2010)], a portable (not Xtensa-optimized) C++ template library for linear algebra, where available. Although Nature and Eigen are the competitive baselines, we also include straightforward loop-nest-based implementations for reference: one with parametric sizes and one with sizes fixed at compile time (with `#define`). Figure 2.5 normalizes simulated cycle times as speedups over the fixed-size naive baseline.

2.5.3 Kernel Benchmarks

Table 2.1 lists the benchmark kernels we use, which are inspired by use cases in computer vision and machine perception. QProd, for instance, is a Euclidean Lie group product [Strasdat(2015)], which includes quaternion and translational product components and appears in applications such as pose estimation or camera

Benchmark	Ref. LOC	Size	Time	Memory
2DConv	131	3×3, 2×2	2.2s	145 MB
		3×3, 3×3	5.6s	145 MB
		3×5, 3×3	30.3s	626 MB
		4×4, 3×3	23.8s	370 MB
		8×8, 3×3	3m 16s†	3.8 GB
		10×10, 2×2	21.6s	401 MB
		10×10, 3×3	3m 24s†	4.1 GB
		10×10, 4×4	3m 11s†	5.0 GB
		16×16, 2×2	1m 8s	1.2 GB
		16×16, 3×3	3m 9s†	4.7 GB
		16×16, 4×4	3m 57s†	4.4 GB
MatMul	71	2×2, 2×2	1.9s	144 MB
		2×3, 3×3	2.2s	136 MB
		3×3, 3×3	2.7s	124 MB
		4×4, 4×4	5.8s	130 MB
		8×8, 8×8	3m 22s†	4.0 GB
		10×10, 10×10	3m 30s†	6.0 GB
		16×16, 16×16	3m 38s†	4.5 GB
QProd	144	4, 3, 4, 3	6.7s	128 MB
QRDecomp	174	3×3	4m 38s†	2.2 GB
		4×4	4h 25m†	35.4 GB

† Equality saturation timed out after 180s.

Table 2.1: Benchmark kernels used to evaluate Diospyros. We list the lines of code in the reference implementation and show the time and maximum memory used for compilation, including symbolic evaluation, optimization, and code generation but not translation validation. 2DConv is a 2D convolution, MatMul is a 2D matrix multiple, QProd is a quaternion product, and QRDecomp is a QR matrix decomposition.

models.

The table also shows the total compilation time for each benchmark. While we set the timeout for equality saturation at just 3 minutes, some benchmarks take a significant amount of time to do backend optimization and code generation. QRDecomp at the 4×4 size is a pathological case. The kernel when fully unrolled is extremely large: the extracted specification alone is a 509 MB text file. As a result,

the E-graph does not saturate and it finds no vector instructions. The expression is heavily redundant, so our post-processing optimizations (Section 2.4) take several hours and gigabytes of memory to remove redundancy before generating output program, producing only 457 lines of C as output. Here, the performance benefits of the additional common subexpression elimination enabled by symbolic evaluation (and exploited by our local value numbering optimization) are enough to beat the naive and library implementations, even without vectorization. We discuss this effect further in Section 2.5.6.

2.5.4 Kernel Performance Results

Figure 2.5 compares the Diospyros-generated kernels against straightforward loop-based implementations (with both parametric sizes and inlined fixed sizes to facilitate more aggressive `-O3` optimizations) and the Nature DSP and the Eigen library functions. On average, Diospyros-optimized kernels outperform the best non-expert baseline by 3.1×.

The Diospyros-generated matrix multiply kernels are between 2.7× and 19.3× faster than the fixed-size naive loop nests. The trends in Figure 2.5 indicate that even highly-optimized code such as Nature can perform poorly on small kernels, such as the 2×2 square matrix product, due to the control overhead of the parametrized unrolling.

In the case of 2DConv, our example from Section 2.2, Diospyros finds solutions that are up to 7.5× faster than the library implementations. Nature outperforms Diospyros on 2DConv at two sizes that are greater than or equal to the vector width: input sizes 16×16 and 10×10, with filter size 4×4. The Nature library's 2D

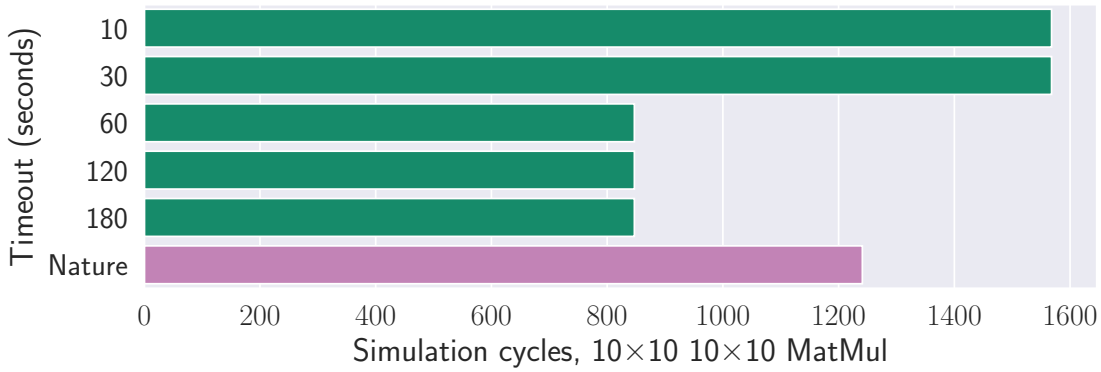

Figure 2.6: Effect of search timeout on MatMul performance.

convolution makes extensive use of vector intrinsics for loads, stores, and arithmetic operations; however, its unrolling strategies are not amenable to cases where the filter size is near but not equal to the vector width.

In the case of matrix multiply, we also have access to proprietary hand-tuned code written for the Fusion G3 by a DSP expert for a single fixed size, 2×3 by 3×3. The Diospyros-generated kernel compiles with full equality saturation in 2.7 seconds and produces run-time performance that is within 8% of the expert performance (39 vs. 36 cycles). The Diospyros kernel and the expert kernel perform the same number and type of vector operations (two multiplies and four multiply–accumulates), but Diospyros's logic to load elements into registers from main memory is less efficient. We believe this performance gap could be eliminated with additional engineering effort in improving code generation.

2.5.5 Timeout Ablation Study

Diospyros's rewrite engine uses a timeout to emit suboptimal solutions even when it does not reach full equality saturation. Shorter timeouts stop Diospyros from completely vectorizing the kernel but still emit an executable C kernel. Figure 2.6

shows the effect of increasing the timeout on our MatMul benchmark for the largest size, 10×10 by 10×10. With a 10-second timeout, the Diospyros generated kernel performs far better than a naive kernel (1,568 cycles), but not as well as the size-agnostic implementation in the Nature library (1,241 cycles). Increasing the timeout improves the quality of the generated benchmark, ultimately saturating the E-graph and finding a kernel that beats even the Nature library taking 847 cycles. This formulation allows programmers to trade off compilation time for run-time performance of the generated kernel.

2.5.6 Vectorization Ablation Case Study

As the results for QRDecomp at the 4×4 size demonstrate, symbolic evaluation alone enables loop unrolling and common subexpression elimination that yield performance benefits even without explicit vectorization. To isolate the performance advantage of our vectorization strategy over other factors, we measure performance for Diospyros with all vector rewriting rules disabled. Compiling kernels with Diospyros without these vector-related rules (but with symbolic evaluation, scalar rewrite rules, and common subexpression elimination) yields code that performs 2.2× better than the best non-Diospyros baseline, compared to 3.1× with vector rewrite rules. In 4 out of 21 kernels, the non-vectorized code is actually faster than the Diospyros-vectorized code because the vendor's compiler can produce more heavily optimized scalar code. We believe Diospyros could improve on these cases with a better cost model that reflects the overheads of vector packing and engineering enhancements to the backend code generation.

2.5.7 Application Case Study

We implement a piece of a digital signal processing application that can use Diospyros-generated kernels to observe their effect in context. Sensing applications such as structure from motion (SFM) [Sweeney(2016)] are rich with small-scale linear algebra kernels calls that can become bottlenecks if they are implemented in a generic way.

This section studies a camera model computation from the Theia open-source SFM package [Sweeney(2016)], which is representative of the kinds of embedded vision workloads that are common on DSPs. Theia is well optimized and uses the popular Eigen [Guennebaud et al.(2010)] library of matrix kernels, but it is not specifically optimized for DSP architectures. It uses a camera model to define how points in 3D space project into a 2D image plane captured by the sensor array. We focus on this initialization function in Theia's camera model:

```
bool Camera::InitializeFromProjectionMatrix(
  const int image_width,
  const int image_height,
  const Matrix3x4d projection_matrix)
```

The core functionality is in `DecomposeProjectionMatrix`, a function that initializes camera parameters projecting to a rotation matrix using a Jacobi SVD decomposition and then decomposing the matrix using RQ decomposition. We port `DecomposeProjectionMatrix` to Tensilica's Fusion G3 DSP. We compare against a version using single-precision floating-point numbers (the original code uses double-precision FP, but both the original and our optimized versions are accurate within 10^{-6} even with single precision). We found that 61% of the run

time was spent on a call to a 3×3 QR decomposition from the Eigen library.

We substitute a QR decomposition kernel generated by Diospyros for the Eigen implementation to measure its effect on the overall computation. QR decomposition is a linear algebra kernel that takes as input a square matrix A and finds a right triangular matrix R and an orthogonal matrix Q such that $A = Q \times R$. Both Eigen and our implementation use the Householder algorithm to iteratively build both outputs, using a series of matrix multiplications along with scalar computations. The number of floating point multiplications is cubic in relation to the matrix size. We implement QR decomposition with about 170 lines of imperative Racket. The resulting SMT-based specification has over 65,000 calls to floating point multiply, demonstrating the complexity of this kernel.

For the complete projection matrix computation, the Diospyros-optimized version is 2.1× faster than the original Eigen-based implementation (30,552 vs. 64,025 cycles). The QR decomposition kernel alone is an order of magnitude faster than Eigen's implementation (see Section 2.5.4), and these savings translate to a substantial speedup in the complete computation.

2.6 Limitations and Portability

While Diospyros's design aims to generalize across DSP architectures, we built the prototype in this chapter to target the Tensilica Fusion G3 specifically. Aspects of the rewriting strategy in Section 2.3.2 reflect the Fusion G3's ISA: namely, the vector width, the available vector arithmetic operations, and the support for flexible "shuffle" instructions for data movement. However, Diospyros's equality saturation engine is parametric over most of these target details—for example, a

simple compile-time setting controls the target vector width.

To target a different DSP, a designer would need to add or remove rewrite rules that reflect the available primitive operations. For example, consider a DSP with a vectorized fast reciprocal operation. To add support for this instruction, a user would need to: (1) add a scalar rewrite rule like (/ 1 x) $\rightsquigarrow$ (`recip` x), relying on existing support for division; (2) inform the rewrite engine that `recip` has a vector equivalent, using a rule builder available in the Diospyros library; and (3) add the target-specific intrinsic to the backend (one to two lines of code to map `VecRecip` to the vendor intrinsic).

An important assumption in Diospyros is that the target can support flexible data movement between vector registers. Its rewrite rules allow unrestricted data movement during equality saturation, with a relatively abstract cost model that assigns a higher cost to gathering data across different inputs or from non-zero scalars. We expect this approach to be most appropriate for architectures with a flexible "shuffle" instruction that uses an index vector to change positions within a vector. For architectures without this kind of flexible data movement, the backend would need to fall back to scalar operations more frequently, which would be more expensive.

2.7 Related Work

Vectorizing compilers. Classical vectorization techniques—from loop dependency analysis [Allen and Kennedy(1987)] to modern auto-vectorization techniques [Mendis and Amarasinghe(2018), Nuzman et al.(2006), Mainland et al.(2013)]—typically do not attempt to aggressively shuffle data into irregular patterns. Ex-

isting techniques prioritize efficient compilation over optimality: they are designed to run on millions of lines of code but miss vectorization opportunities.

Previous work has used the Halide language [Ragan-Kelley et al.(2013a)] to target DSPs, but has not supported exploration of a large search space of irregular data movement strategies [Vocke et al.(2017)]. Other approaches can generate target-specific shuffles to implement known permutations, but do not find the permutation strategies themselves [Franchetti and Püschel(2008), McFarlin et al.(2011)]. Our search strategy can discover novel shuffles and data movement, automating the labor-intensive hand-tuning process at the cost of increased compilation time.

SLinGen [Spampinato et al.(2018)], a part of SPIRAL [Puschel et al.(2005), Franchetti et al.(2006)], optimizes small linear algebra kernels by first applying optimizations like loop reordering and vectorization and then autotuning. Like SLinGen, Diospyros works at a higher abstraction level to enable optimizations that would not be apparent at the assembly level. However, our work uses equality saturation both to avoid hand-crafting specific optimization patterns (including for custom functional units that are common on DSPs) and to offer higher coverage of the search space than autotuning.

Program synthesis. Program synthesis techniques can expend compilation time to discover novel optimized programs. [Barthe et al.(2013)] develop an auto-vectorizer using inductive synthesis but focus on general-purpose code rather than linear algebra and so do not generate shuffles. [Cowan et al.(2020)] generate quantized machine learning kernels using syntax-guided synthesis. Their sketches exploit the reduction structure of these kernels and so cannot invent new data move-

ment. MSL [Xu et al.(2014)] is a synthesizer that generates bulk-synchronous parallel programs. The synthesizer reduces the parallel problem to a sequential one, uses a syntax-guided synthesis tool [Abate et al.(2018)] to solve the sequential problem, and then compiles the result to message-passing parallel code.

Swizzle Inventor [Phothilimthana et al.(2019)] infers permutations of data and computation (*swizzles*) that are optimized for GPU memory hierarchies. Unlike Swizzle Inventor, Diospyros has the ability to change the compute code itself (e.g., by fusing multiply–accumulates) rather than just the data movement. Swizzle Inventor also requires users to provide a sketch identifying the sites of possible swizzles; Diospyros's rewrite rule system does not require sketching.

Unlike many synthesis techniques, Diospyros has the ability to extract *partial* solutions if the synthesis process takes too long. Recent work [Peleg and Polikarpova(2020)] explores synthesis techniques that are *best effort*, returning partially valid solutions. Diospyros's rewrite rules are sound, and so the partial solutions it returns are always valid, but the partial solutions are not provably optimal (even with respect to the limited rewrite rules). This design allows Diospyros to avoid expensive optimality proofs that can dominate synthesis time [Bornholt et al.(2016)]. Incorporating unsound rewrite rules that can be repaired at code generation time is an appealing direction for future work.

An earlier version of Diospyros [VanHattum et al.(2020)] relied on syntax-guided synthesis backed by an SMT solver. It generated optimized linear algebra kernels but encountered scaling issues even on small (2×2) kernels because it needed to reason about bit-level instruction semantics during synthesis. Diospyros now abstracts away arithmetic semantics and focuses on vectorization by using term rewrite instead, so it can scale to kernels $10\times$ larger than the SMT-based

version. In addition, the previous version of Diospyros required a full program sketch in addition to a specification for each kernel. The current Diospyros system allows users can reuse the same rewrite rules across different kernels.

Term rewriting systems. Diospyros's optimization approach is based on equality saturation [Tate et al.(2009), Willsey et al.(2021)], a technique for optimizing compilation using equality graphs (E-graphs). Equality saturation alleviates the phase ordering problem of traditional compilers by applying rewriting rules to an E-graph, implicitly capturing all possible phase orderings. Recent work expands equality saturation to new compilation domains such as CAD models [Nandi et al.(2020)]. These approaches exploit the insight that equality saturation does not require backtracking so it admits an asymptotically more efficient E-graph implementation [Willsey et al.(2021)]. Diospyros instantiates this approach for vectorization, using equality saturation to exhaustively search candidate vectorized programs that include data movement.

Denali [Joshi et al.(2002)] is an equality saturation-based superoptimizer for Alpha assembly code. It saturates an E-graph using assembly-level rewrite rules and then extracts an optimal program by using a SAT solver to compute a detailed cost model. Diospyros's rewriting happens instead over an abstract DSL, which sacrifices some target-specific optimality in favor of reasoning about data movement; such higher-level optimizations are typically where expert DSP developers focus their hand-tuning efforts.

2.8 Chapter Summary

Diospyros combines symbolic evaluation, equality saturation, and translation validation to build an end-to-end compiler for high-performance DSP code. Diospyros is extensible: users can bring domain- and architecture-specific insights by adding new rewrite rules to the equality saturation scheme. A main avenue for future work is to exploit this flexibility to target more DSP targets and other esoteric, customizable hardware architectures beyond DSPs.

2.9 Chapter Acknowledgements

We thank Jacob Delgado-López for his implementation contributions and Armin Alaghi and Max Willsey for early feedback on this work. Many thanks to the anonymous ASPLOS 2021 reviewers and our shepherd, Shoaib Kamil, for their detailed feedback.

The work in this chapter was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. It was also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). Support included NSF awards #1845952 and #1723715. This material was based upon work supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-1650441. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 3

KANI: VERIFYING DYNAMIC TRAIT OBJECTS IN RUST

3.1 Introduction

Rust has made significant inroads as a popular *safe* systems programming language over the decade since its release. Stack Overflow has named Rust the "most loved language" every year since 2016.[1] One of the language's main selling points is its focus on reliability—the ownership type system is a success story of programming language memory safety research breaking into the mainstream. The borrow checker eliminates certain high-impact classes of bugs, including null pointer dereferences, use-after-frees, and most forms of leaked memory. A team writing safety- or security-critical code, though, may seek an even higher level of assurance than what the current type system alone provides.

While Rust's type system rules out most memory safety bugs in checked *safe* code, there remain many ways for execution to go wrong. The language provides an "unsafe" dialect that allows programmers to bypass restrictions to regain more expressivity for lower-level regions of code. Even in safe Rust, the type system does not rule out dynamic panics from out-of-bounds or indexing errors (for example, consider the well-type-checked snippet `let v = vec![1, 2]; v[3]`). Finally, engineers may want assurance of *functional correctness*—the ability to assert specific properties about the result of a program under all possible inputs.

The Kani Rust Verifier (Kani) is an open-source tool for sound, bit-precise symbolic analysis of Rust programs—initially motivated by use cases at Amazon Web

[1] `https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/`

Services (AWS). In our previous work on symbolic correctness proofs of production C code, we found that (1) embedding specification into proof harnesses similar to unit tests, and (2) integration with existing developer workflows were key to broad impact on software engineering teams [Chong et al.(2020)]. To this end, one of our primary goals with the Kani project is to support enough of the Rust language surface to seamlessly integrate into large, existing projects. In Section 3.4.2, we show how Kani performs on components of the open-source Firecracker virtual machine monitor,[2] which provides virtualization for two publicly-available serverless compute services at Amazon Web Services: Lambda and Fargate [Agache et al.(2020)].

We have found an unexpected challenge in Rust language coverage to be correctly modeling dynamic dispatch through virtual method tables. Rust does not have classes or class inheritance like other object-oriented languages; rather, *traits* are the primary mechanism for defining interfaces and abstracting over implementations. The official Rust blog states:[3]

> The trait system is the secret sauce that gives Rust the ergonomic, expressive feel of high-level languages while retaining low-level control over code execution and data representation.

By default, trait method calls are *monomorphized*—that is, the compiler statically resolves which concrete function to call at each function call site (see Section 3.2.1).

However, users can add a `dyn` keyword to gain the expressivity of dynamic dispatch to trade-off dynamic run time for improved code size and compilation times (see Section 3.2.2). Further, Rust's closures, or anonymous functions, can also be dynamically dispatched through trait objects. As we show in Section 3.4.1,

[2] `https://firecracker-microvm.github.io/`
[3] `https://blog.rust-lang.org/2015/05/11/traits.html`

37% of the 500 most popular Rust crates (packages) explicitly invoke dynamic dispatch in their source code, and 70% implicitly include code that uses it.

Despite Rust's minimal runtime, supporting these dynamic trait objects is challenging because (1) they require non-trivial dynamic dispatch semantics that are not explicitly specified in any Rust documentation, and (2) they require heavy use of function pointers, which can be challenging for static analysis and symbolic execution algorithms [Milanova et al.(2004), Lu and Hu(2019)]. While dynamic trait objects are easy to avoid in hand-crafted verification examples, their use in the Rust standard library and throughout realistic, real-world crates motivates providing full support within our Kani tool. We have also seen in practice that faithfully modeling dynamic trait semantics causes our verification times to become intractable due to the large number of function pointers. In Section 3.3.3, we show how Kani leverages semantic information about traits to restrict the number of possible targets for function pointers, moving a Firecracker proof from intractable to completing successfully in 16 minutes.

Verification for Rust is a growing field, but to the best of our knowledge, Kani is the only symbolic model checking tool that targets Rust's Mid-level Intermediate Representation (MIR) and can reason about dynamic trait objects and dynamic closures. Other verification tools that target MIR either do not provide soundness guarantees over symbolic inputs (MIRI [Jung et al.(2019)], MIRAI [Experimental(2021)]) or do not support all cases of dynamic traits (Prusti [Astrauskas et al.(2019)], CRUST [Toman et al.(2015)], Crux-MIR [Galois, Inc(2020)]); other tools target LLVM-IR and thus do not leverage MIR-level type information (SMACK [Baranowski et al.(2018)], SeaHorn [Gurfinkel et al.(2015)], RVT-KLEE [Reid et al.(2020)]).

Kani is implemented as a backend for the Rust compiler that uses a mature, industrial-strength model checking tool—the C Bounded Model Checker (CBMC) [Clarke et al.(2004)]—as a verification engine. Kani translates Rust's Mid-level Intermediate Representation (MIR) into *Goto-C*, CBMC's C-like intermediate representation. Specifications in Kani are written as Rust-source-level `assert!(...)` statements, with simple extensions to specify assumptions and non-deterministic symbolic input (Section 3.3.1). Kani can be invoked on individual Rust files or on crates with the `Cargo` Rust build tool. In addition to the user-added assertions, Kani checks for arithmetic overflow, out-of-bounds memory accesses, and invalid pointers. CBMC performs bounded unrolling of loops and recursion in the program, but Kani by default is run with assertions that guarantee that if code is verified, loops are sufficiently unrolled (via an assertion that any iterations beyond the unrolling bounds are unreachable).

In this chapter, we identify dynamic trait objects as an essential language feature for Rust verification tools to tackle in order to enable use on large, real-world Rust projects. Our contributions are as follows:

1. We describe the Kani Rust Verifier, an open-source bit-precise symbolic model checker for Rust programs. We show that covering dynamic trait objects semantics is necessary to reason about real-world Rust, and we identify nuanced interactions between dynamic dispatch and the Rust borrow checker that must be correctly modeled by tools that target Rust's Mid-level Intermediate Representation (MIR).
2. We show how Kani uses MIR-level semantic information about traits to restrict possible targets for function pointers, which pose a well-known performance challenge for symbolic execution tools.

3. We provide a case study on the open-source Firecracker repository that shows that function pointer restrictions unlock a previously intractable proof, with verification performance (under 20 minutes) suitable for use in continuous integration.
4. We share an open-source suite of verification test cases for dynamic trait objects and compare the results of several related tools.

3.2 Rust Trait Overview

Traits are a core Rust language feature for specifying when types should share a common interface. By default, Rust uses a monomorphization process to concretize each possible method implementation with a specific type. But, programmers can instead opt-in to dynamic dispatch when they use a trait to trade off run-time performance with improved code size and compilation times.

3.2.1 Traits and Monomorphization

To understand dynamic dispatch, we first describe Rust's default static dispatch techniques for trait objects. We start with a motivating example which defines an interface for objects that have an integer count method:

```
trait Countable { fn count(&self) -> usize; }
```

We can implement this trait for two data structures, the Rust standard library's `Vec` and our own custom `Bucket` struct:

```
impl Countable for Vec<i8> {
  fn count(&self) -> usize { self.len() }
}
struct Bucket {
  item_count: usize,
}
impl Countable for Bucket {
  fn count(&self) -> usize { self.item_count }
}
```

Now, we can use the `Countable` type to refer to any object that implements the trait:

```
fn print_count<T: Countable>(obj: T) {
  print!("Count = {}", obj.count());
}
```

This implementation specifies that the function takes a generic type `T` that must implement the `Countable` trait. Lower-level languages like assembly do not, of course, support generics. How, then, does the compiler resolve the call in the body of `print_count` into an actual function call jump (that is, which `count` implementation should be called)?

By default, the Rust compiler uses *monomorphization:* it creates a specialized `print_count` function for each concrete type. This process happens at the MIR level, but the effect is roughly equivalent to this Rust source code:

```
fn print_count_vec_i8(obj: Vec<i8>) {
```

```
  print!("Count = {}", obj.count::<Vec<i8>>());
}
fn print_count_bucket(obj: Bucket) {
  print!("Count = {}", obj.count::<Bucket>());
}
```

Monomorphization means that every function that uses a generic type bound must be duplicated for every possible implementation.

Closures as dynamic trait objects.

Closures are anonymous functions that can capture (and if specified, mutate) values in the environment where they are defined. Each closure has its own unique concrete type (that is, even closures that share the same signature do not share a concrete type). This creates a difficulty: what type should be used when a closure is passed into a higher-order function, such as `map`? Rust solves this using traits: all closures must implement at least one of three standard-library-defined traits: `FnOnce`, `FnMut`, or `Fn`, depending on whether they consume, mutably reference, or immutably reference the captured environment (see Section 3.3.2).

For example, we could define a function that takes in an item cost and a closure to calculate the price of that item with tax:

```
fn price<T: Fn(f32)->f32>(cost: f32, with_tax: T)
  -> f32 {  with_tax(cost) }
```

To call this function, we simply specify a closure as the second argument:

```
let tax_rate = 1.1;
```

```
price(5., |a| a * tax_rate); // Price is: 5.5
price(5., |a| a + 2.);       // Price is: 7
```

Rust will monomorphize the code at compile time to call the right implementation (we use `[closure@...]` to represent the closure environment, which stores the `tax_rate` in the first closure and is empty in the second):

```
fn price_closure@main:1(cost: f32) -> f32 {
  closure@main:1([closure@main:1], cost)
}
fn price_closure@main:2(cost: f32) -> f32 ...
```

The costs of traits.

With this monomorphization strategy, developers pay no run-time efficiency cost compared to code that manually specifies each implementation without using generics or abstraction. However, monomorphization can have undesirable effects: an increase in code size and compilation time, especially as the number of possible implementations grows.

Verification tools can often avoid reasoning about monomorphization by consuming Rust code *after* monomorphization completes, either by running MIR's default monomorphizer or by targeting a lower-level of code in compilation, such as LLVM IR. From the perspective of a verification tool, it is feasible to handle Rust code with statically dispatched trait objects by instead using only the monomorphized, concrete functions.

3.2.2 Dynamic Trait Objects

To trade off run-time efficiency with improved code side and compilation time, developers can use *dynamic trait objects* to opt in to dynamic dispatch (and out of monomorphization). For example, using our same `Countable` trait, a developer could have this alternative implementation of `print_count`:

```
fn print_count(obj: &dyn Countable) {
  print!("Count = {}", obj.count());
}
```

To pass a trait object to this function, developers need to cast it `as` a dynamic trait object:

```
print_count(&Bucket::new(1) as &dyn Countable);
```

Here, the `dyn` keyword expresses that this object should have method calls dynamically dispatched. That is, the Rust compiler will use a different strategy to answer the question: "which implementation should `obj.count()` call?"

Rather than creating a new function signature per concrete type for `print_count`, the Rust compiler will use a single instance of `print_count` that takes a single type that can represent all objects that implement `Countable`. In Rust, this type is an instance of a *fat pointer*—a double-wide pointer type that represents both data and essential metadata. Fat pointers for dynamic trait objects consist of a `data` pointer to the object itself and a pointer to the *virtual method table* (vtable) [Driesen and Hölzle(1996)] that maps trait-defined methods to their implementations.

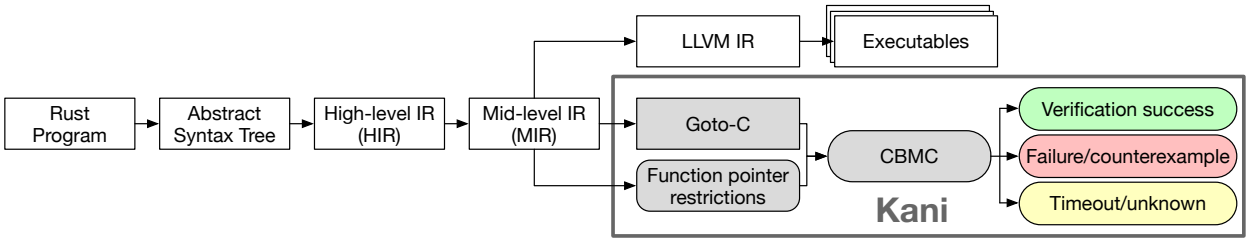

Figure 3.1: Architecture of Kani. Unfilled indicates the Rust compiler with the canonical LLVM backend; filled indicates Kani. *IR* is Intermediate Representation. Kani translates Rust's MIR to CBMC's C-like Goto-C language, uses MIR type information to emit function pointer restrictions, and outputs a successful verification, a failure with a counterexample trace, or a timeout.

Rust's implementation using vtables.

Rust Mid-level Intermediate Representation (MIR) uses abstract trait types, so it is up to each backend to implement vtables as they lower to their corresponding lower-level representation. Because vtables require jumps to a dynamically computed address, they can potentially be exploited in security attacks (e.g., in C++ [Jang et al.(2014)]), and hence their precise implementation has security implications. Although Rust's informal specification does not specify the exact vtable layout, MIR provides utility functions for building vtables of a specific form. When we lack documented semantics for how Rust treats dynamic trait objects, we use the canonical LLVM backend as a reference. Our descriptions are based on Rust 1.55.0, the latest version of the compiler at the time of writing.

In the canonical LLVM backend for the Rust compiler, vtables have a specific layout that contains object metadata (the size and alignment of the data) as well as pointers for each method implementation. Every vtable includes a pointer to the concrete type's `drop` (destructor) method implementation. The remainder of the vtable contains pointers to the concrete implementation of all methods defined by that trait. A new vtable is defined at compile time for every cast statement

between a unique pair of concrete object type and trait type, and stored in a new global variable. Dynamic trait objects that share the same concrete type can thus share the same vtable.

The vtable for our `Countable` example is (conceptually):

sizeof<Bucket>	8
align<Bucket>	8
&Bucket::drop	0x7ffe02d0ba88
&Bucket::count	0x7ffe02d0ba90

The fat pointer for our `&Bucket as &dyn Countable` object would have one pointer to the `Bucket` and one pointer to the vtable above. Calls to methods that take `self` then can pass the `data` pointer as self. For example, `print_count` would be implemented as roughly:

```
fn print_count(obj: &dyn Countable) {
  print!("Count = {}",
         *(obj.vtable.count)(obj.vtable.data));
}
```

For dynamic closures, the `data` half of the fat pointer points to the closure's environment. The vtable consists of the same `size`, `align`, and `drop` metadata, then pointers to functions defined for `Fn`, `FnMut`, and/or `FnOnce`.

Summary.

Dynamic trait objects allow developers to compile code with dynamic objects that carry metadata specifying which trait implementations of methods to call, rather

than statically duplicating code through monomorphization. Dynamic trait objects are used throughout the Rust standard library, so even if programmers do not opt-in to dynamic dispatch within their own source code, they are likely to pull in Rust source that constructs and uses vtables (see Section 3.4.1). Rust's dynamic dispatch poses a challenge for verification both because *how* to implement them is not precisely specified by the Rust language definition, and because function pointers require pointer analyses that are a known challenge for symbolic reasoning [Milanova et al.(2004), Lu and Hu(2019)].

3.3 Methodology and Implementation

3.3.1 The Kani Rust Verifier

Architecturally, Kani is implemented as code generation backend to the `rustc` compiler (Figure 3.1).[4] Instead of translating to machine code (e.g., via the LLVM compiler infrastructures for the standard backend or Cranelift for an experimental debug backend), Kani translates to `Goto-C`, the C-like intermediate representation for CBMC [Clarke et al.(2004)]. Kani then invokes CBMC on the generated `goto` program, which ultimately runs symbolic execution and discharges formulas to an off-the-shelf SAT or SMT solver (by default, MiniSAT [Eén and Sörensson(2003)]).

Properties checked and soundness.

Kani by default checks for memory safety (pointer type safety, invalid pointer indexing in unsafe code, slice/vector out-of-bounds), arithmetic overflow, run-time

[4] `https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html`

panics, and violations of user-added assertions. Users can additionally specify `assert!` and `kani::assume` statements using Rust syntax. To reason about all possible inputs, users specify variables as non-deterministic symbolic inputs using a special generic `kani::any<T>()` function. We use *sound* to indicate that Kani never misses violations of the checked properties in the `rustc`-produced binary execution on some input. We have made the conscious choice when developing Kani to prioritize soundness over completeness, so Kani fails prior to verification if it encounters a Rust language feature it does not yet support. Kani currently focuses on sequential Rust and thus fails on any concurrency constructs. Kani also fails on some compiler intrinsics, including a subset of SIMD (vector single instruction, multiple data) operations.

While CBMC can act as a bounded model checker, Kani uses it for unbounded verification. By default, CBMC is bounded because it requires either a heuristic or a user-specified unrolling bound to unroll each loop and set of recursive function calls. When symbolic execution reaches the specified bound, CBMC defaults to inserting an `assume(false)`, which stops further exploration of the execution. However, CBMC provides an `unwinding-assertions` flag that asserts that any loop iteration beyond the specified bounds is unreachable. Kani enables this flag—this causes us to be potentially incomplete on programs where the bounds cannot be specified, but provides an assurance of soundness for all cases where Kani returns "SUCCESS".

Choice of input representation.

The Rust compiler translates a Rust program between a series of increasingly low-level representations, as shown in Figure 3.1. One of the key architectural

choices when designing a Rust verifier is what level of representation the verification tool should take as input. Each level has both advantages and disadvantages for verification. On the one hand, each step lower in the representation tends to use a smaller set of more uniform constructs. Defining a formal semantics is therefore easier at lower levels. Tools such as SMACK operate at the LLVM intermediate representation (LLVM-IR) level, which has the additional benefit of allowing a shared verification backend between different languages, such as C and C++.

On the other hand, lower-level representations lose information about the original structure of the program and hence about the original intent of the programmer. For example, the compiler may give implementation-defined semantics in a lower-level representation to an operation that is undefined behavior at a higher level. We have found that the Rust Mid-level Intermediate Representation (MIR) to be an effective interface for verification. MIR is a (fairly) clean and compact representation that retains most of the semantic Rust type information. Kani invokes monomorphization before analysis takes place, so we do not explicitly need to reason about generic constructs. As we demonstrate in Section 3.4.2, MIR's rich type information is crucial to enabling high-performance verification of dynamic trait objects.

The need for bit-precision.

`unsafe` Rust code can both read and modify objects as a collection of raw bytes, bypassing the borrow checker and type system. For example, Rust code can use `transmute` to reinterpret bytes of one type as bytes of another or use raw pointer indexing to directly view and modify the bytes of a type. These features are used for performance and portability benefits in production Rust code, including in the

Rust standard library.

For example, the standard library's `OsStr` implementation notes:

```
/* FIXME: `OsStr::from_inner` current implementation relies on
`OsStr` being layout-compatible with `Slice`... */
pub struct OsStr { inner: Slice, }
```

In order to verify such code, it is necessary that the bitwise layouts used by Kani match those used by the Rust compiler itself. While relying on implementation details like this is undefined behavior for source-level Rust code, the standard library is able to rely on stronger implementation-level guarantees from the Rust compiler. Kani's CBMC backend provides the bit-level reasoning necessary to handle such cases.

3.3.2 Dynamic Trait Objects in Kani

Goto-C (and C) do not have native support for method dispatch, so Kani must lower MIR to C in a manner that removes traits but maintains the same semantics. Our primary strategy is to follow the LLVM backend's vtable implementation, emitting Goto-C instead of LLVM IR.

vtable construction.

Dynamic objects are created at cast sites, where a concrete type is cast to a `dyn` type explicitly or implicitly. Like the LLVM code generation backend, Kani keeps a cache of vtables that constructs a new vtable for every unique concrete object, trait type tuple. Vtables generated by Kani are Goto-C structs that map the

metadata identifier to the corresponding data.

Naming vtable fields was less straightforward than we anticipated. In the LLVM code generation backend, vtables are global allocations *without* named fields (rather, each individual element is accessed through pointer arithmetic). To keep our generated Goto-C code more debuggable by Kani developers (and counterexample traces more readable for users), we opted to use a struct with named fields (because each field is the size of one pointer, the memory layout is the same). An earlier version of Kani mapped the method name to the method implementation function pointer. However, we found this failed to handle cases where an object implemented two traits with the same method name.

Unlike some other languages, Rust allows a type to implement two traits with identically-named methods (regardless of whether their signature is the same):

```
trait A { fn is_odd(&self) -> i32; }
trait B { fn is_odd(&self) -> bool; }
impl A for i32 { ... };
impl B for i32 { ... }
trait C : A + B {}
impl C for i32 {}
// The vtable for x has two 'is_odd' entries
let x: &dyn C = &3 as &dyn C;
```

To resolve this ambiguity, Kani now uses *the index of the item* in the vector returned from a Rust MIR API call—`vtable_entries`—to uniquely identify methods. We confirmed this strategy in discussions with Rust compiler developers.[5]

[5] `https://rust-lang.zulipchat.com/#narrow/stream/144729-wg-traits/topic/.E2.9C.94.20object.20upcasting/near/246857652`

Specifically, we create a new vtable when we see a cast from a sized pointer type to an `unsized` (non-slice) pointer type, where we have not already created a vtable for this concrete type, trait type pairing. At construction, we iterate over the Rust compiler's new (June 2021) `vtable_entries`[6] results. We construct `size` and `alignment` using the Rust compiler's API for layout and drop resolution.[7] For each method defined explicitly for that trait type, we add an entry indexed by position in the canonical `vtable_entries`.

Virtual calls through vtables.

Dynamic dispatch occurs when a statement calls a method on a dynamic trait object.

At the MIR level, we construct a dynamic call through a vtable when we encounter a virtual call terminator. We obtain the object's self pointer and vtable pointer by accessing the respective components of the fat pointer. We use the index `idx` provided by the virtual call object to determine the vtable method—which corresponds with the index into the vector returned by the Rust compiler's `vtable_entries`.

Casts of dynamic objects.

Rust does not currently support general dynamic trait *upcasting* (see Section 3.6): i.e., one cannot cast an object of type `&dyn Foo` to one of type `&dyn Bar` even if

[6] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_trait_selection/traits/fn.vtable_entries.html`

[7] `https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/layout/trait.LayoutOf.html`,`https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/ty/instance/struct.Instance.html#method.resolve_drop_in_place`

one is a subtype of the other (unlike, for example, Java subtyping). The underlying reason is that Rust prefers to stay as close to *zero cost abstractions*[8] as possible—giving users high level language features without sacrificing performance. Totally generic trait upcasting would require modifying or rebuilding vtables (or additional pointer indirection), imposing a run-time cost.

Kani initially encoded the assumption that dynamic trait objects could thus not be the *source* of cast statements. When we tested Kani on the standard library, Kani found violations of this assumption when handling types like `&dyn Error + Send`. Looking more into the Rust documentation for traits, we found:

> The `Send`, `Sync`, [...] and `RefUnwindSafe` traits are auto traits. *Auto traits have special properties.* [...]

Because auto traits like `Send` *have no associated methods*, the underlying vtable does not need to change when a cast involves only auto-traits. The Rust compiler therefore allows adding and removing auto traits in dynamic trait objects casts, breaking Kani's initial assumption. To reason about the Rust standard library as-is, verification tools must be able to handle this type of cast.

Closure signatures.

Our initial implementation of dynamic trait objects in Kani (which, like the current version, prioritized soundness over completeness) failed to verify due to a CBMC pointer error on the following input:

[8] `https://blog.rust-lang.org/2015/05/11/traits.html`

```
let f: Box<dyn FnOnce(i8)> = Box::new(|x| {
  assert!(x == 1);
});
f(1);
```

A nearly-identical version of this case with `Fn` replacing of `FnOnce` verified successfully. The root issue was a surprisingly subtle interaction between Rust's borrow checker and dynamic dispatch.

The Rust documentation includes the following:

> Use `FnOnce` as a bound when you want to accept a parameter of function-like type and only need to call it once. If you need to call the parameter repeatedly, use `FnMut` as a bound; if you also need it to not mutate state, use `Fn`.

FnOnce thus has a method signature that *moves* ownership of its self type by taking it by-value: `fn call_once(self, args: Args)-> Self::Output`. This allows the Rust borrow checker to give errors on attempted reuse such as:

> `‘f‘ moved due to this call. This value implements ‘FnOnce‘, which causes it to be moved when called`

In comparison, *Fn* has this method signature: `fn call(&self, args: Args)-> Self::Output`. Both `Fn` and `FnOnce` are used for dynamic dispatch via vtable calls, using the `self` parameter as one argument. This is the root cause of our verification failure—the machinery we have described for closures and vtables requires that the vtable's `self` argument be a *pointer* to an object.

Rust uses a *vtable shim* to work around this mismatch:

```
/*`<T as Trait>::method` where `method` receives unsizeable `self:
   Self`...The generated shim will take `Self` via `*mut Self` -
   conceptually this is `&owned Self` - and dereference the
   argument to call the original function. */
VtableShim(DefId),
```

However, the full translation is not complete at the MIR level: before code generation, backends must be sure to correct the function call signature Ror example, from the Rust compiler:

```
if let InstanceDef::VtableShim(..) = self.def {
  // Modify `fn(self, ...)` to `fn(self: *mut Self, ...)`
```

Backends can either disregard the MIR function signature and use a separate `fn_abi_of_instance`, or apply this same correction to the MIR function signature. Verification tools can reasonably make either choice—but using the MIR function signature alone in this case will lead to incorrect results.

3.3.3 Leveraging Trait Semantics for Function Pointer Restrictions

One of Kani's key advantages over more language-agnostic verification tools is that it can exploit Rust's semantics to improve verification completeness and performance. While other tools (i.e., SMACK [Baranowski et al.(2018)], RVT-KLEE, RVT-SeaHorn [Reid et al.(2020)]) that work at the level of LLVM IR must work with vtables as opaque allocations generated by the standard Rust backend, Kani

can offer a more direct interpretation of dynamic dispatch that allows us to combat path explosion in the verification state space. At the MIR level, we have access to rich trait information; by the time Rust is lowered to LLVM, traits are gone and replaced by non-specific LLVM pointer types which are more difficult to reason about. Specifically, Kani uses information about dynamic trait object creation (at object cast sites) and use (at function call sites) to restrict the set of possible targets for vtable function pointer calls.

The verification challenge of dynamic dispatch.

In general, indirect function calls pose a scalability challenge for program verification due to the pointer analysis involved [Milanova et al.(2004), Lu and Hu(2019)]. Before running symbolic execution, CBMC removes all function pointers by lowering them to conditional `if` blocks between possible target functions. By default, CBMC considers all functions in the code generation unit of the correct function signature to be possible targets (this is sound when CBMC is run with pointer checks, which verify that all pointers are to objects of the correct type).

In a simple case, this permissive approach works well. Consider two functions with the same signature:

```
fn a(x: i32) -> bool { x == 2 }
fn b(x: i32) -> bool { x != 0 }
```

When a pointer to a function of this type is used, for example, `(*f)(2)` CBMC's algorithm conceptually emits the following:

```
if (f == &a)      a(2);
else if (f == &b) b(2);  // ...
```

CBMC can then use standard symbolic execution techniques for conditional control flow to soundly reason about this code.

This strategy becomes problematic when run on large code bases that pull in numerous dependent crates. *Every* dynamic trait object uses a function pointer every time a method is called, because each trait-defined method call is resolved through a vtable entry. The number of possible function pointer targets especially proliferates for calls to `drop`—the destructor function. Every object's `drop` function signature shares a shape: a method that takes a single `self` parameter and returns the unit type (analogous to a void return in C). When the `self` type is something from the standard library, such as `std::io:Error`, the number of possibilities skyrockets. In Section 3.4.2, we show how such a case can lead to hundreds of possible targets, rendering this approach to verification intractable.

Restricting call destinations using Rust semantics.

We recognized that, with Kani's semantic understanding of traits at the MIR level, we have a much more precise notion of which implementations vtable function pointers could target. In particular, we can guarantee (short of the user using `unsafe` to transmute the vtable memory) that a call through a vtable will be one of the trait-defined methods for that trait type that we have encountered during code generation. To maintain soundness even under unsafe memory transmutes, we `assert!(false)` if the actual function pointer does not match one of our identified possibilities. This also allows us to soundly under-approximate possible targets by not explicitly accounting for casts between trait types, as described in Section 3.3.2.

We implement function pointer restrictions by tracking possible implementations (at object cast sites, when the vtable is built) and uses of vtable pointers

(at function call sites). For the first, we build a map that builds a set of possible implementations (in our case, symbol names for each Goto-C function) keyed by the tuple of trait type and method index for vtables of that trait type. For call sites, we build a list of `structs` with trait name, method index for vtables of that trait type, and information for identifying that call site in CBMC. Because dynamic dispatch calls can occur across crate boundaries, we emit a file with this information for each crate (using a stable unique hash for trait types). Finally, after code generation, we combine the per-crate data by iterating over the list of call sites and looking up the possible implementations we have found for that trait index tuple. When there are no possible implementations (possible in functions that are never invoked), we emit the empty set.

Kani combines the restrictions for each Rust dependency and the crate itself into an auxiliary JSON file to be consumed by CBMC as function pointer restrictions. We have seen CBMC times drop by an order of magnitude with this restriction strategy, as we describe in the next section.

3.4 Evaluation

For our evaluation, we used an Amazon EC2 `m5d.4xlarge` instance with 16 cores and 64GB of memory, running Ubuntu 20.04.2.[9]

[9] `https://github.com/avanhatt/icse22ae-kani`

3.4.1 Prevalence of Dynamic Trait Objects

We conducted a simple empirical study to estimate the prevalence of dynamic trait objects in the 500 most downloaded crates on `crates.io`, the Rust package repository. We found that while only 185 of these 500 crates (37%) use the explicit `dyn` keyword within their source code, 349 (70%) include at least one vtable when compiled with rustc.

We downloaded the top 500 crates sorted by greatest number of downloads on October 2, 2021. To estimate the implicit use of dynamic trait objects, we invoked a debug build of the Rust compiler via `cargo build` and searched the debug output for the line `get_vtable`, which is logged at vtable use. This is likely an over-estimate of the dynamic trait objects that are actually used in functionality a user might want to verify for these crates, but it does provide an indication of how often verification tools that integrate with Cargo will encounter linked dynamically-dispatched code.

3.4.2 Case Study: Firecracker

As a real-world case study, we consider how two different variants of Kani—one without vtable function pointer restrictions, and one with—perform on examples from the open-source Firecracker hypervisor. This case study highlights the challenges in moving from small, standalone verification examples to proofs that sit alongside large scale codebases.

Implemented in Rust, Firecracker provides the underlying virtualization technology for two publicly-available serverless compute services at Amazon Web Ser-

```
pub trait BusDevice: AsAny + Send {
  fn read(&mut self, offset: u64,
          data: &mut [u8]);
  fn write(&mut self, offset: u64, data: &[u8]);
}
```

Figure 3.2: The `BusDevice` trait used for explicit dynamic dispatch in Firecracker's serial device.

vices: Lambda and Fargate [Agache et al.(2020)]. A core characteristic of serverless computing is multitenancy, meaning that multiple customer workloads (e.g., functions or containers) may run on the same hardware. Consequently, Firecracker is crucial for ensuring the *isolation* of customer workloads.

Firecracker Serial Device.

Firecracker provides console emulation for a guest virtual machine by emulating a serial device (16550A UART). The guest virtual machine sends and receives bytes by writing and reading to device registers mapped into the guest memory. Since read and writing to a device through memory is a common interface, Firecracker defines a trait `BusDevice` which defines `write` and `read` methods (Figure 3.2). Multiple devices are wrapped in a `Bus` container which maps address ranges to a particular device and routes write and read requests as dynamic calls to the underlying device.

Verification task. We aim to demonstrate a simple proof harness using the serial device behavior in loopback mode, where bytes are read and written to the same port. Firecrack's serial device specifies that only a single byte can be read or written in a given call. Figure 3.3 shows a small proof harness that checks that for *any* single byte we can write through the dynamically-dispatched call, the same

```
fn serial_harness() {
  let mut serial = SerialDevice {
    serial: Serial::new( ... ) };
  // Model arbitrary input as symbolic
  let bytes: [u8; 1] = kani::any();
  let mut buf = [0x00; 1];
  // Call functions-under-verification
  <dyn BusDevice>::write(&mut serial, 0u64, &bytes);
  <dyn BusDevice>::read(&mut serial, 0u64, &mut buf);
  assert!(bytes[0] == buf[0]);
}
```

Figure 3.3: Our proof harness for simple `read`/`write` functionality. `kani::any()` is an Kani construct that returns a non-deterministic, symbolic value of the inferred type.

byte is read back. Kani checks this user-added assertion, as well as memory safety, arithmetic overflows and division by zero, and pointer safety.

Function pointer restrictions. With our function pointer restrictions enabled, Kani identifies exactly the correct function pointer to consider a target for both `read` and `write`. In Kani without function pointer restrictions, CBMC's default function pointer strategy finds 8 possible calls for each of `read` and `write`. For example, the call for `write` includes these two options, which are from an entirely different module of Firecracker but are included because they share the same function signature:

```
if(v.vtable->6 == Block_VirtioDevice_read_config)
  goto __CPROVER_DUMP_L11;
if(v.vtable->6 == Block_VirtioDevice_write_config)
  goto __CPROVER_DUMP_L12;
```

For this simple illustrative case, Kani runs in 4 minutes and 4 seconds with the restrictions and 4 minutes and 13 seconds without, representing a modest 5%

speedup (with a tradeoff in code size increasing from 1.14GB to 1.20GB due to the auxiliary restrictions files). In the next example, we show how implicit vtable calls can cause far worse performance differences.

Firecracker Block Device Parser.

For our next function-under-verification, we consider the emulated block device available to guest virtual machines (VMs) for storage (i.e., reading and writing to disk). The device is visible to the guest VM through MMIO (memory-mapped IO), using the virtio API [Russell(2008)]. The guest VM allocates a set of `virtqueue` data structures in guest memory to support generic data transport between the guest and hypervisor. Each entry in a `virtqueue` is a descriptor: a pointer with metadata, such as length and read/write permissions, to a buffer in guest memory. Descriptors can be chained so that multiple buffers can be transported in a single transaction. For the block device, a read (respectively, write) transaction consists of three descriptors pointing to three buffers, containing (1) the request type and disk sector, (2) the data buffer to be filled/read, (3) a status byte returned by the device. The primary task of the emulated block device is to parse and execute guest transactions that it receives through this interface.

Verification task. Isolation between guest VMs requires that no input from a guest, no matter how malformed, can cause Firecracker to panic. Figure 3.5 gives a straightforward proof harness for the `parse` function of the block device (Figure 3.4). The parse function is responsible for taking the raw untrusted bytes of descriptors from guest memory and returning either a request object or an error. We use symbolic inputs (generated with `kani::any()` on line 3) to model input

```
pub fn parse(
    avail_desc: &DescriptorChain,
    mem: &GuestMemoryMmap,
) -> result::Result<Request, Error> { ... }
```

Figure 3.4: `parse`, our function-under-verification.

```
fn block_proof_harness() {
  // Model arbitrary descriptor from guest as symbolic
  let desc : DescriptorChain = kani::any();
  // ..., call function-under-verification
  match parse(&desc, /*...*/) {
    Ok(req) => {},
    Err(_) => {},
  }
}
```

Figure 3.5: Our proof harness for `parse`.

from the guest as well as to over-approximate data values read from the guest memory. Successfully verifying this proof harness using Kani shows that the block device has no run-time panics under any guest behavior.[10] Kani can be used to verify deeper functional properties, in addition to panic freedom—for the purpose of this case study, we note that even this simple harness is intractable without MIR-level type reasoning.

Although the code-under-verification never uses the `dyn` keyword to explicitly

[10]Running this proof with the default set of Kani flags gives spurious pointer check errors (which we are investigating), so the results in this section are for Kani with pointer checks disabled.

```
pub enum devices::virtio::block::Error {
  // Guest gave us a descriptor that was too short
  DescriptorLengthTooSmall,
  // Getting a block's metadata fails for any reason
  GetFileMetadata(std::io::Error),
  // ...
}
```

Figure 3.6: The error type used in the `Result` returned by `parse`.

invoke a dynamic trait, the `parse` function returns the type `result::Result<Request, Error>`, where `Error` is a custom enum `devices::virtio::block::Error`. As shown in Figure 3.6, one enum value uses the standard library type `std::io::Error`, which is implemented using traits. When the returned object goes out of scope (when `block_proof_harness` returns), Rust automatically inserts a call to destruct the object with `std::ptr::drop_in_place`. This `drop_in_place` function uses a dynamic trait object of type `Drop`, which is routed through the object's vtable.

Impact on verification. Even in this simple case, the hidden use of dynamic trait objects poses a huge challenge for verification with CBMC—`std::io::Error` is so commonly used within Firecracker and its dependencies that CBMC identifies 314 possible function targets for this virtual call to `drop`. Each of these functions must then be unwound for symbolic execution. CBMC's symbolic execution engine was unable to complete this unwinding within a four hour timeout (and hence never even reached the stage of discharging the actual proof obligations to a satisfiability solver).

Our trait-based function pointer restrictions allow our proof harness for parse to terminate successfully in 16 minutes—at least a 15× improvement in verification performance. Code size again increases slightly, from 0.96GB for the proof harness without restrictions to 1.02GB for the successful proof. For the problematic call to `drop` on `std::io::Error`, Kani correctly identified that the `GetFileMetadata(std::io::Error)` type is never used in this harness or function-under-verification. That is, Kani emits 0 possible functions that could actually be the target of the precise `Error` type in this context, rather than the extremely permissive 314 possible options. Since Kani soundly replaces the call to `drop` with `assert(false)`, verifi-

cation of the test case also serves as verification of the function-pointer restriction set. As an additional sanity check, we modified the function-under-verification to non-deterministically return a `std::io::Error` in some cases, which caused Kani to fail with spurious, false positive verification errors. Our manual inspection of these failures indicates that they do not affect soundness, but we are investigating them as a top priority.

3.4.3 Dynamic Dispatch Test Suite

In developing Kani, we have produced a suite of over 40 verification test cases for dynamic trait objects. This test suite has been open source throughout its development.[11] We encourage other developers of Rust verification tools to use and modify these test cases as they add more support for dynamic trait objects. Our full suite includes versions of the functions-under-verification that are expected to succeed and versions that are expected to fail.

Table 3.1 shows our understanding of other tools' support for a subset of test cases. We used the following versions:

- SMACK: `version 2.8.0`.
- Crux-MIR: commit hash `3451423`.
- Rust Verification Tools: commit hash `b179e90`.
- Prusti: `rustc 1.56.0-nightly (3d0774d0d 2021-08-18)`.
- Crust: no longer actively developed, the paper specified that dynamic traits were unsupported [Toman et al.(2015)].

[11] `https://github.com/model-checking/kani`

Test name	Description	Code snippet
`trait-ptr`	Simple trait, pointer	`&3 as &dyn T`
`trait-box`	Simple trait, boxed	`Box::new(o) as Box<dyn T>`
`auto-ptr`	Auto trait, pointer	`&3 as &dyn Send`
`fn-ptr`	`Fn` closure, pointer	`$||{} as &dyn Fn`
`fnonce-box`	`FnOnce` closure, boxed	`Box::new(||{}) as Box<FnOnce>`
`gen-ptr`	Generic trait, pointer	`trait T: S<i8> + S<u8>`
`drop-box`	Explicit drop, boxed	`impl Drop for T...Box<dyn T>`
`drop-ptr`	Explicit drop, pointer	`impl Drop for T...&dyn T`

Tool	**Kani**	**Crux-MIR**	**RVT-SH**	**RVT-KLEE**	**SMACK**
Focus	Soundness	Soundness	Soundness	Bug-finding	Soundness
Test name					
`trait-ptr`	✓	✓	✓	✓	✓
`trait-box`	✓	✗	✗	✓	✗
`auto-ptr`	✓	✗	✗	✓	✓
`fn-ptr`	✓	✗	✓	✓	✓
`fnonce-box`	✓	✗	✗	✓	✗
`gen-ptr`	✓	✗	✓	✓	✓
`drop-box`	✓	✗	✓	✓	✗
`drop-ptr`	✓	✓	✓	✓	✓

Table 3.1: Dynamic trait object test cases to compare Kani to other verification tools (✓ is supported, ✗ is unsupported). Crux-MIR is an MIR-based "static simulator." RVT-SH (RVT-SeaHorn), RVT-KLEE, and SMACK all start with LLVM IR; respectively, they are a model checker, a bug finder, and a Boogie-based verifier. Prusti and CRUST did not support any form of dynamic trait object.

3.5 Trusted Computing Base and Limitations

Kani is designed as a sound verifier with respect to the properties checked, but because neither MIR nor Goto-C currently have formal semantics, the full Kani toolchain itself is not formally verified. Kani's trusted compute base includes our translation from MIR to Goto-C, CBMC itself, and the backend SAT or SMT solver.

Compared to some other Rust formal methods tools, Kani's use of Rust syntax for assertions and assumptions limits us to a smaller space of expressible properties. Supporting richer specifications—including support for first-class loop invariants, explicit existential quantifiers, and modular verification—is future work.

3.6 Discussion and Future Work

In this chapter, we have outlined how a language feature that is thought to be well-understood—dynamic dispatch—can pose unanticipated verification challenges. Prior efforts to formalize Rust semantics have (reasonably) focused on other unique language features, primarily the borrow checker. For example, the RustBelt[Jung et al.(2017)] project's λ_{Rust} "omits some orthogonal features of Rust such as traits (which are akin to Haskell type classes)". The Oxide: Essence of Rust[Weiss et al.(2019)] paper similarly references Haskell type classes and does not see traits as an "essential part of Rust". We have a slightly different goal than this prior work: because we want to embed verification of Rust in real world codebases, we *needed* to wrangle with the semantics of trait objects, and we found that doing so was far from trivial. From this, we can argue a point broader than just Rust—verification tool designers should be prepared to model the complex and subtle ways *all* language features interact. This is especially true when languages provide a standard library that is not formally specified but uses the desired language feature.

Trait upcasting coercion. Rust has an in-progress proposal to add a *trait upcasting coercion* feature for dynamic trait objects.[12] This language feature would

[12] `https://rust-lang.github.io/dyn-upcasting-coercion-initiative/CHARTER.html`

allow developers to cast between dynamic trait object types as long as the source type is a subtrait of the destination type. Implementing such coercions requires a more complicated vtable strategy, since they require the underlying vtable to change. Kani could be extended to support these trait coercions once they are enabled by default in Rust. To do so, we would need to extend our vtable generation and method lookup to model the Rust compiler semantics, which will likely be vtables with a nested structure that can require multiple pointer indirections. This would also be additional motivation to extend our existing strategy for restricting function pointers to include directed type cast information.

3.7 Related Work

MIR-based verification.

Other tools target Rust's Mid-level Intermediate Representation; but to our knowledge no other tool provides sound verification of symbolic inputs and supports the breadth of dynamic trait objects.

CRUST [Toman et al.(2015)] is a similar bounded model checker for Rust that also uses the CBMC tool as a verification backend. However, CRUST explicitly does not support dynamic trait objects or dynamic dispatch and is no longer being actively developed.

Prusti [Astrauskas et al.(2019)] is a Rust compiler plugin built on the Viper verification infrastructure that can verify user-added specifications, as well as the absence of panics. Like Kani, Prusti leverages MIR type information to improve verification results. Prusti has a more expressive language for proof annotations

than Kani, including supporting loop invariants that allow verification of programs Kani cannot currently verify. However, Prusti has limited support for unsafe code and does not support dynamic trait objects (our tests fail with compiler errors). A recent extension to Prusti adds additional support for closures; however, to our knowledge, this extension does not handle dynamic closures [Wolff et al.(2021)].

Crux-MIR is a symbolic execution that similarly targets Rust's MIR [Galois, Inc(2020)] using Galois' Crucible verification infrastructure. Crux-MIR can verify simple cases of dynamic dispatch through `&dyn` pointer references. However, the tool fails with unimplemented for boxed dynamic objects (e.g., `Box<dyn T>`) and dynamic closure objects (e.g., `&dyn Fn()-> i32`).

Facebook's experimental MIRAI is an abstract interpreter for MIR [Experimental(2021)]. MIRAI explicitly prioritizes a low false-positive rate for bugs rather than a low false-negative rate, and thus does not claim to provide sound verification.

LLVM-IR based Rust verification.

Several LLVM-based tools have been extended to better support Rust code. As we showed in Section 3.4.2, the generality of supporting Rust at the LLVM IR level comes with the downside of being unable to apply Rust-type-level semantic understanding. However, LLVM-backed solutions tend to be less dependent on supporting changes to Rust, which currently evolves more quickly than LLVM.

The SMACK toolchain has been used to verify Rust by using the existing `rustc` backend to produce LLVM IR [Baranowski et al.(2018)]. SMACK's toolchain was initially designed to primarily support Clang as a front end and thus required changes (primarily to alias analysis) to support Rust programs. Further, SMACK's

handling of the `Box` datatype requires that the box type be `Sized`, which seems to render the tool unable to reason about boxed dynamic closures.

Google Research's Rust Verification Tools (RVT) Project [Reid et al.(2020)] aimed to build on a range of existing verification tools, from property testing to symbolic execution. Their tool supports multiple symbolic execution engines, each based on LLVM IR. RVT includes a KLEE [Cadar et al.(2008)] backend that can cover our full test suite of cases. However, KLEE is designed with a focus toward bug finding rather than unbounded, sound verification. RVT's SeaHorn backend uses the SeaHorn Verification Framework [Gurfinkel et al.(2015)] and provides sound verification, but fails on some boxed closure test cases.

Analyzing virtual calls.

Indirect function calls pose well-known problems for program analysis in general because identifying the code being invoked entails pointer analysis [Milanova et al.(2004), Lu and Hu(2019)]. Symbolic execution tools, for example, sometimes resort to requiring user annotations to handle indirect calls [Ramos(2015), Section 3.4] [Maksimović et al.(2021)]. In languages with built-in support for virtual calls, such as object-oriented languages, optimizing compilers typically attempt *devirtualization*, opportunistically replacing indirect calls with direct calls when pointer information is sufficient, to make programs more analyzable [Padlewski(2017), Ishizaki et al.(2000)]. The function-pointer restriction technique in this work (Section 3.3.3) resembles a form of devirtualization that relies solely on type information from Rust's trait system, with the goal of improving model-checking efficiency and precision.

3.8 Chapter Summary

For verification of Rust to be deployed in large-scale projects, tools need to reason about the dynamic trait objects that are pervasive throughout the Rust standard library. In this chapter, we demonstrated how our model-checking tool, Kani, successfully translates Rust's dynamic trait semantics. We show that by targeting Rust at the Mid-level Intermediate Representation level rather than LLVM-IR, we can leverage trait-based type information to improve verification time up to 15×. Our Firecracker case study highlights how this semantic understanding of traits unlocked previously intractable verification results. We encourage the other verification projects to use and build on our open-source suite of tests for dynamic dispatch, and we look forward to working with the Rust community to build an ecosystem where developers can verify functional correctness of security- and safety-critical Rust programs.

3.9 Chapter Acknowledgements

Thank you to Ted Kaminski, Zyad Hassan, Kareem Khazem, Adrian Palacios, Niko Matsakis, Michael Tautschnig, Rachit Nigam, and the anonymous reviewers for ICSE SEIP 2022 for their feedback in improving this chapter. Thank you to the AWS Automated Reasoning Group for creating such a welcoming and exciting research environment.

CHAPTER 4

VERIISLE: LIGHTWEIGHT, MODULAR VERIFICATION FOR INSTRUCTION SELECTION

4.1 Introduction

WebAssembly [Haas et al.(2017)] (Wasm) is a portable bytecode format originally designed for the browser, with three main goals: safety, speed, and portability. Wasm's machine-independent but low-level semantics make compilation and execution fast on any platform; its type system and bounded memory regions work together to prevent programs from reading or writing data outside of their own heap (their *sandbox*). This isolation guarantee is essential when users interact with the web, because each click leads to untrusted code.

Isolation has made Wasm popular beyond the web, too. *Edge cloud* services from Cloudflare [Kenton Varda(2018)], Vercel [Vercel Inc.(2023)], and Fastly [Pat Hickey(2019)], for example, run users' Wasm code on geographically distributed content delivery networks. To improve startup time, these Wasm-based services can co-locate different untrusted code modules *within the same process*; Wasm's lightweight isolation enforcement takes the place of more traditional, costly process- or VM-based isolation.

Unlike a process or VM, however, Wasm's safety guarantee relies on the correctness of the underlying compiler. The compiler inserts dynamic checks that confine a module to its own memory region *before* generating native code for that module. Code generation, then, is a pillar of every Wasm-backed system's trusted compute base: almost *any* miscompilation, however seemingly benign or

rare, could be exploited to produce code that bypasses Wasm's security guarantees [Fallin(2021a), Crichton(2022c), Crichton(2022a), Crichton(2022b)]. Code generation bugs can let malicious Wasm code steal data from—or corrupt the execution of—completely unrelated modules or the host runtime itself.

As one example, a code generation CVE[1] in Cranelift [Bytecode Alliance(2023b)], a compiler backend used in several industrial Wasm runtimes, permitted this kind of sandbox escape [Crichton(2023)]. The bug was in Cranelift's `x86-64` instruction selection, which uses addressing modes to implement complex address computations with a single instruction. `x86-64` addressing modes can apply small left shifts, so a single `movl` instruction is enough to implement code like the following Wasm snippet:

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

To lower this code to `x86-64`, Cranelift must convert 32-bit Wasm addresses into offests from an instance's *base address* in the target machine's 64-bit address space. This conversion requires zero-extending the 32-bit Wasm address, computing the 64-bit address as `base+zext(addr)` (where `addr` is the original 32-bit Wasm address, `base` is the base address for the module's memory region, and `zext` is a zero-extension). Unfortunately, the Cranelift instruction selector lowered the above Wasm code to `x86-64` instructions that computed `base+zext(x)<<3` *instead of* `base+zext(x<<3)`. This mistake lets attackers break out of the Wasm sandbox by giving them access to an extra *3 significant bits of native address space*. In Wasmtime [Bytecode Alliance(2023c)], a popular Wasm engine that uses Cranelift, this allows a guest Wasm instance to silently read and write memory 6 to 34 GB away from its own sandbox. Clearly, *even simple bugs in instruction selection can create*

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

serious security vulnerabilities.

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

To help compiler developers *automatically* reason about the correctness of their instruction-lowering rules, we present VeriISLE. VeriISLE verifies rules written in Cranelift's ISLE domain-specific language (DSL) for specifying how IR terms translate to machine code sequences. To use VeriISLE, developers annotate their ISLE lowering rules with specifications; VeriISLE uses a Satisfiability Modulo Theories (SMT) solver [Barrett et al.(2010)] to automatically verify full functional equivalence—i.e., that a rule translates an IR instruction to a native code sequence with equivalent semantics. VeriISLE allows developers to *gradually* annotate new rules, and to quickly update annotations as rules evolve. This modularity is essential because Cranelift is an evolving production compiler: lowering rules—and entire backends!—are subject to change. To our knowledge, our work with VeriISLE is the first formal verification effort for the instruction-lowering phase of an efficiency-focused production compiler.

In sum, in this chapter, we:

1. Create VeriISLE, a framework for verifying instruction-lowering rules in the

ISLE domain-specific language.

2. Verify Cranelift's implementation of all integer operations in the latest major WebAssembly release—1.0 [Rossberg(2019)]—for the ARM `aarch64` Instruction Set Architecture (ISA).
3. Use VeriISLE to reproduce and detect previously-fixed bugs (Section 4.4.3) and vulnerabilities (Section 4.4.3), including the example bug from this section.
4. Use VeriISLE to help Cranelift developers identify (Section 4.4.4, Section 4.4.4) and fix (Section 4.4.4) new bugs and under-specified compiler invariants (Section 4.4.4).

We begin by introducing background on instruction lowering and the ISLE DSL (Section 4.2.1). Then, we present VeriISLE's design (Section 4.3), and evaluate its results on Cranelift (Section 4.4), a production Wasm compiler backend. Finally, we discuss plans to build on VeriISLE towards fully-verified Wasm compilers (Section 4.6).

4.2 Background

This section provides background for understanding VeriISLE verification (Section 4.3) by describing the instruction lowering problem (Section 4.2.1) and Cranelift's ISLE domain-specific language (DSL) for writing lowering rules (Section 4.2.2). Finally, it introduces SMT solvers [Barrett et al.(2010)], the tools that power the VeriISLE verification engine (Section 4.2.4).

4.2.1 Instruction Lowering

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

Most modern instruction selectors *do* support more general N-to-M matching; in fact, a good instruction selector often embodies a good *pattern matcher*. It detects arrangements of multiple operators in the IR that can be translated, together, into machine instructions. In full generality, this is an NP-hard combinatorial search problem; as a result, most production compilers use heuristic shortcuts for practicality (e.g., greedy pattern matching, as in the "maximal munch" scheme [Cattell(1978)]).

More complex ISAs and ISA extensions yield more complex matching strategies. For an extreme example, bit-permutation and swizzling instructions vary widely across ISAs, and lowering of a general permutation operator sometimes requires a "solver"—or at least a bevy of heuristic special cases to produce good code [Ren et al.(2006), Mendis and Amarasinghe(2018), VanHattum et al.(2021)]. This is part of what makes instruction selection (and instruction selection verification!) interesting: it is *not* simply the task of mapping mostly-equivalent operators, like translating IR addition to the machine's integer addition instruc-

tion. The most subtle reasoning—and many bugs—occur when there is a large semantic gap between the IR and ISA, and when producing efficient machine code is a first-order priority [Yang et al.(2011), Lopes and Regehr(2018)].

Production compilers today use a mix of hand-written and DSL-based descriptions of their instruction lowering rules: e.g., LLVM [Lattner and Adve(2004)] has a 46K-line C++ file specifying `x86-64` lowerings, while the Go compiler uses a term-rewriting DSL where developers can specify expression-tree patterns [Go Authors(2023)]. In this chapter, we focus on the Cranelift compiler's lowering DSL.

4.2.2 The ISLE Lowering DSL

The Cranelift compiler project [Bytecode Alliance(2023b)] introduced the ISLE (***I****nstruction* ***S****election* ***L****owering* ***E****xpressions*) DSL [Fallin(2021b), Alliance(2023), Fallin(2023)] in 2021 in order to replace handwritten instruction-lowering code with declarative patterns. ISLE is broadly a term-rewriting system [Dershowitz and Jouannaud(1991), Visser et al.(1998)]. In the next sections, we give a brief overview, and then walk through an example of instruction lowering in ISLE.

ISLE's term rewriting for lowering

The main body of a program in ISLE consists of a series of rules. These rules are written in S-expression syntax and consist of a *left-hand side (LHS)* and *right-hand side (RHS)*. The LHS is a pattern, and can use pattern-matching operators such as wildcards, variable captures, or destructuring (matching a term and then feeding

its arguments to sub-patterns). The RHS is an expression consisting of a tree of terms, possibly using variables captured from the LHS. A rule indicates that the RHS expression is produced whenever the instruction selector encounters a term tree matching the LHS.

To express instruction lowering as term rewriting, ISLE introduces a top-level term `lower` that takes an expression tree as its argument. For example, to lower an integer add operator (`iadd`) to the `add` instruction in the ISA (e.g., `x86-64` or `aarch64`), one would write:[2]

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

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

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

Because of the type system's restrictions, Cranelift expresses all lowerings as rewrites from `(lower (IR_operator ...))` to term trees representing machine code expressions, potentially passing through multiple intermediate terms. The term `lower` is necessary because the LHS and RHS of a rule must have the same type—but top-level LHS patterns return IR `Inst`s, while top-level RHS expressions

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

return machine Registers. `lower`, with type signature `(decl lower (Inst)Reg)`,[3] does the `Inst` to Register conversion that allows lowerings rules to type check by giving the LHS and RHS the same type.

Finally, ISLE's type system supports *automatic type conversions*. In the `iadd` example, such conversions apply to `x` and `y`, which are variables of type `Value` bound by the left-hand side of the rule. The RHS, in contrast, operates on `x` and `y` Registers. To reconcile these incompatible types, the ISLE compiler automatically inserts type conversions if a conversion rule has already been specified for a pair of types. In this case, ISLE wraps the latter uses of `x` and `y` with the user-defined term `put_in_reg`, which converts `Value`s to `Reg`s.[4]

4.2.3 ISLE by Example: Lowering Rotations

In this section, we walk through Cranelift's lowerings for a few specific instructions; this sets us up to *verify* such lowerings in the next section (Section 4.3).

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

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

[3] We elide an indirection via another type for clarity.
[4] We describe the semantics of `put_in_reg` in Section 4.3.1.

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

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

Cranelift must instead special-case on narrow values:

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

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

The rotate rule also uses an *intermediate term*, `small_rotr`. `small_rotr` only

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

```
(rule
  (lower (has_type (fits_in_16 ty) (rotl x y)))
  (let ((neg_amt Reg (a64_sub I32 (zero) y)))
    (small_rotr ty (zext32 x) neg_amt)))
```

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

4.2.4 Satisfiability Modulo Theories (SMT)

To verify lowering rules written in ISLE, VeriISLE uses an Satisfiability Modulo Theories (SMT) solver [De Moura and Bjørner(2008)]. SMT solvers are tools that determine whether logical formulas are *satisfiable* for some assignment of values to

all variables in the formula.

Unlike SAT formulas [Moskewicz et al.(2001)], SMT formulas allow users to express higher-level statements (e.g., "`x < y[2]`") using a rich set of operators and types (e.g., integers and arrays) that are defined in the SMT-LIB standard [Barrett et al.(2010)]. VeriISLE lowers ISLE rules to SMT formulas in the theory of bitvectors and integers; we discuss this further in the next section.

4.3 VeriISLE Design

VeriISLE is a framework for verifying rewrite rules in the ISLE domain-specific language for instruction selection. VeriISLE uses an SMT solver [De Moura and Bjørner(2008)] to show functional equivalence of the left- and right-hand sides of individual rules.[5] An equivalent left and right side mean that the rule has preserved IR semantics at the machine code level; a differing left and right side indicate a bug in the lowering.

To verify their lowering rules, compiler developers write annotations on ISLE terms in VeriISLE's *annotation language* (Section 4.3.1). This language makes it simple to express term semantics (e.g., that `fits_in_16` means that a type can losslessly be represented with 16 bits). VeriISLE consumes ISLE's program representation for rules, combines this with the compiled annotations to create its own intermediate representation, and performs *type inference* (Section 4.3.1). Type inference is necessary for VeriISLE to lower its IR to an SMT formula, a logical formula that asks whether a rule's right and left-hand sides are equivalent.

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

Finally, VeriISLE feeds the resulting formula into the SMT solver. If the right and left-hand sides of a rule differ, the solver returns a counter-example showing a set of inputs that cause the divergence; otherwise, the rule is verified.

In this section, we walk through the verification pipeline, from VeriISLE's annotation language (Section 4.3.1) to how it constructs and customizes verification conditions (Section 4.3.2).

4.3.1 The Annotation Language

It is impossible to verify functional correctness without precise semantics on terms within ISLE. While there are formal semantics for certain ISAs (e.g., ARM [Armstrong et al.(2019)] and Intel [Dasgupta et al.(2019)]), there are no semantics for Cranelift's intermediate representation—or for ISLE helper terms (e.g., `has_type`) and intermediate terms (e.g., `small_rotr`). The *challenge* in specifying these semantics is that production compilers are living software: engineers change rules, add rules, and occasionally add entire new back-ends. To support modular verification of an evolving codebase, VeriISLE introduces an annotation language that allows rule authors to define specifications *as they go*, introducing a term's semantics inline, next to the term itself.

For example, consider our annotation on the helper term `fits_in_16`:[6]

```
(spec (sig (args arg) (ret))
 (provide (= ret arg))
 (require (<= arg (16:Int))))
(decl fits_in_16 (Type) Type)
```

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

⟨*annot*⟩ ::= (`spec` ⟨*sig*⟩ (`provide` ⟨*ex*⟩$^+$) (require ⟨*ex*⟩$^+$))
⟨*sig*⟩ ::= (`sig` (args ⟨*bound*⟩$^+$) (⟨*bound*⟩))
⟨*bound*⟩ ::= (⟨*ident*⟩ `:` ⟨*type*⟩)
⟨*type*⟩ ::= `bv` | `bv` ⟨*int*⟩ | `Int` | `Bool`
⟨*width*⟩ ::= ⟨*int*⟩ | ⟨*ex*⟩
⟨*const*⟩ ::= `true` | `false` | ⟨*int*⟩ | ⟨*bitvector*⟩
⟨*ex*⟩ ::= ⟨*ident*⟩ | ⟨*const*⟩ | ⟨*encoding*⟩ ⟨*ex*⟩$^+$ | (⟨*unop*⟩ ⟨*ex*⟩) | (⟨*binop*⟩ ⟨*ex*⟩ ⟨*ex*⟩)
| (⟨*conv*⟩ ⟨*width*⟩ ⟨*ex*⟩) | (`extract` ⟨*int*⟩ ⟨*int*⟩ ⟨*ex*⟩) | (`int2bv` ⟨*width*⟩ ⟨*ex*⟩)
| (`bv2int` ⟨*ex*⟩) | (`widthof` ⟨*ex*⟩) | (`concat` ⟨*ex*⟩$^+$)
| (`if` ⟨*ex*⟩ ⟨*ex*⟩ ⟨*ex*⟩) | (`switch` ⟨*ex*⟩ (⟨*ex*⟩ ⟨*ex*⟩)$^+$)
⟨*unop*⟩ ::= `!` | `~` | `-` | ...
⟨*binop*⟩ ::= `=` | `!=` | `>=` | `<=` | `<` | `>` | `sgt` | `sgte` | `slt` | `slte` | `ugt` | `ugte` | `ult`
| `ulte` | `+` | `-` | `*` | `sdiv` | `udiv` | `srem` | `urem` | `&` | `|` | `xor` | `sdiv` | `rotl` | `rotr`
| `shl` | `shr` | `ashr`
⟨*conv*⟩ ::= `sign_ext`| `zero_ext` | `convto`
⟨*encoding*⟩ ::= `cls` | `clz` | `rev` | `subs` | `popcnt`

Figure 4.1: VeriISLE's annotation language, which combines SMT-LIB constructs with conveniences (e.g., `switch`) and VeriISLE-specific constructs (e.g., `convto` and `widthof`).

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

The annotation language grammar and semantics

Figure 4.1 shows the VeriISLE annotation language grammar. Most operations in the annotation grammar map directly to SMT-LIB constructions. For example, `+` applied to a bitvector maps to SMT-LIB's `bvadd` bitvector addition function.

VeriISLE adds conveniences like `switch` and a variadic `concat` operation, both of which desugar to folding SMT-LIB's fixed-argument `ite` (if-then-else) and `concat` (bitvector concatenation) operators over any number of arguments. `switch` also adds a verification condition that enforces that its branches are exhaustive, which has helped surface faulty annotations.

VeriISLE provides constructs for introspecting on and modifying bitvector widths. `widthof` returns the width—often only known directly at solving time (Section 4.3.2)—of a given bitvector value. `convto` changes the width of its bitvector argument with the following semantics: if the destination width is more narrow, `convto` extracts the relevant bits; if the destination width is wider, `convto` leaves the upper bits unspecified by concatenating a fresh SMT variable with unrestricted bits.

VeriISLE also provides higher-level versions of SMT-LIB constructs. For example, SMT-LIB rotates must have statically-provided widths; VeriISLE instead offers symbolic rotates, which it implements with shift and bitvector logic instructions. Finally, VeriISLE includes keywords that map to custom encodings in its backend: (1) `cls` and `clz`, which count the number of leading sign and zero bits, respectively (Section 4.4.3), (2) `rev`, which reverses the order of bits, (3) `subs`, which performs subtraction-with-flags, and (4) `popcnt`, which counts the number of 1 bits.

`provide` blocks specify the semantics of a term, typically by relating the returned value bound in the specification to one or more of the arguments. `require` blocks specify preconditions, which are assumed when a term is used on the left-hand side of a rule but checked—that is, verified to hold—when a term is used on the right-hand side of a rule. This is analogous to more traditional Hoare-style verification [Hoare(1969), Barnett et al.(2005)], where function preconditions may be assumed within the body of a function but must be checked at function call site.

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

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

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

The annotation language type system

Types in VeriISLE are integers, booleans, and bitvectors. The VeriISLE annotation language must support polymorphism over bitvector widths, since most of Cranelift's ISLE rules operation on its `Value` type, which is polymorphic over integer values in the Cranelift intermediate representation. (Section 4.2.2).

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

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

Type inference

The annotation language supports polymorphism over bitvector types, but its target representation does not: all bitvector operations in SMT-LIB operate on fixed-width bitvectors [Niemetz et al.(2019)]. Therefore, VeriISLE must transform its high-level intermediate representation, which allows polymorphic bitvector types, into several SMT formulas, each over a different set of bitvector widths. VeriISLE uses two passes of *type inference* to determine those widths. The first inference pass produces an assignment of SMT types (e.g., bitvector) for each variable in a term or its specification. The second pass resolves the bitvector widths.

First pass. First, VeriISLE runs a variant of classic unification-based type inference [Martelli and Montanari(1982)] in order to rule out type errors between annotations. This first pass yields an SMT type (kind)—either an integer, boolean, or bitvector—for each variable in both the specification and the term it describes. The first pass, however, *does not* necessarily resolve the width of each bitvector.

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

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

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

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

$s \rightarrow d$

for s in $\{i8, i16, i32, i64\}$

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

Most terms on the right-hand side of Cranelift's ISLE rules operate on types

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

For most rules, this second pass produces a single possible type assignment. For some rules, there are multiple valid type assignments—in this case, we continue the verification process until the SMT solver says there are no more unique possible type assignments (similar to counter-example guided inductive synthesis [Abate et al.(2018), Solar-Lezama et al.(2006)]).

4.3.2 Generating Verification Conditions

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

1. `Success`: the rule is verified.
2. `Failure with counterexample`: the rule is broken, and the solver provides a set of inputs that exposes the bug, formatted in ISLE surface syntax.
3. `Rule inapplicable`: for the given type instantiation, the rule does not match. This indicates that the rule contains predicates on the left-hand side—or guarded `if/if-let` clauses (see Section 4.4.4)—such that the rule never matches on this type instantiation.

To produce these 3 outcomes, VeriISLE uses (at least) two additional SMT queries.

The first query determines if the rule is applicable by querying the solver to see if there exists a model in which all the necessary preconditions hold; if not, VeriISLE produces a `Rule inapplicable` result. The second query determines whether the lowering rule preserves equivalence; if so, `Success`, and if not, `Failure with counterexample`.

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

The first query: applicability

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

$$\exists\{i_0, \dots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} | P^{LHS} \wedge R^{LHS} \wedge P^{RHS} \tag{4.1}$$

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

step would be unable to find any counterexamples—because in first order logic, false implies anything). Including P^{RHS} in the query makes such a rule fail at the applicability check.

The optional model distinctness check. The applicability check succeeds as long as at least one assignment of input terms is applicable—*even if* there is just one set of applicable inputs. VeriISLE implements an optional check that looks for distinct input sets (i.e., checks that multiple SMT models are feasible in which every bitvector input term is distinct). VeriISLE creates a formula that asserts that each bitvector input differs from the one in the original model; if the query is unsatisfiable, there is only one set of matching inputs. This check identified a previously unknown bug where an ISLE rule never fired in practice (Section 4.4.4).

The second query: equivalence

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

$$\forall \{i_0, \ldots, i_{n-1}\} \cup A^{LHS} \cup A^{RHS} | \\ (P^{LHS} \wedge R^{LHS} \wedge P^{RHS}) \Rightarrow (ret^{LHS} = ret^{RHS}) \wedge R^{RHS} \quad (4.2)$$

To convert this statement to an SMT query, VeriISLE plays the standard trick of asking if there are counterexample inputs such that the verification conditions do

not hold (by switching the quantifier to an existential and negating the implication).

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

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

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

Optional custom verification conditions and assumptions. Some compiler transformations intentionally break strict equivalence. For example, Cranelift attempts to rewrite comparisons that include a statically-known argument to prefer an even integer immediate: as a mathematical identity, $A \geqslant B + 1 \rightarrow A - 1 \geqslant B \rightarrow A > B$. This rewrite is profitable because even values are more likely to fit

in ARM64's 12-bit immediate encodings, improving code size.

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

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

4.3.3 Implementation and Trust Model

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

[7]Plus 26,465 lines for our auto-generated annotation language parser.
[8]Forked at commit `9556cb1`.

way for future work in fuzzing VeriISLE's annotations).

Caveats and the trusted computing base. VeriISLE is limited to reasoning about individual rewrite rules written in ISLE; it reasons about correctness in instruction lowering itself, but trusts other passes in the Cranelift compiler and Wasm runtime. Cranelift and the Wasmtime engine invoke instruction selection *after* WebAssembly safety checks are inserted, but prior to a couple final compiler stages (e.g., register allocation).[9] VeriISLE also trusts the semantics of ISLE terms as written in the annotation language (though our `provide` and `require` distinction and concrete tests help find bad specifications). For example, we found that an old version of VeriISLE did not require condition codes to fall into a valid range. Finally, VeriISLE currently reasons about each rule individually. Support for verifying properties over multiple rules (e.g., reasoning about rule priorities) is future work.

4.4 Evaluation

This section answers the following evaluation questions:

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

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

Q3 Can VeriISLE reproduce prior, known Cranelift bugs?

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

Q4 Can VeriISLE help identify and fix new bugs?

We answer **Q1** by verifying a natural subset of rules, those necessary to compile integer computations in the latest major release of WebAssembly ("1.0" [Rossberg(2019)]). Section 4.4.2 addresses **Q2**—we find that the rules we verify comprise 19.8% of the lowering rules invoked by the WebAssembly reference test suite.

To answer **Q3**, we choose two previously-discovered CVEs in ISLE rules (out of 14 Wasmtime CVEs, 10 of which do not involve ISLE); we also select an ISLE bug that was not assigned a CVE because it affects non-Wasm types. We annotate the buggy rules and present the counterexamples VeriISLE produces in Section 4.4.3.

Finally, in Section 4.4.4 we address **Q4**, outlining 3 new faults (2 patched) that VeriISLE discovered, and 1 *compiler mid-end* bug that VeriISLE helped root-cause and patch. These case studies highlight that instruction-lowering rules are error-prone even for experienced compiler engineers: many of the issues were subtle interactions between constants, sign- and zero- extensions, and tricky bitwidth-specific reasoning. Moreover, to our knowledge, no new bugs have been discovered by any other means (e.g., any Cranelift fuzzers [Arteaga et al.(2022)]) in rules verified by VeriISLE.

4.4.1 Is VeriISLE Applicable to Real Rules?

We use VeriISLE to verify the instruction-lowering rules for all integer operations[10] from WebAssembly's 1.0 release to the ARM `aarch64` backend. In addition, we

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

	Rules	Type Instantiations
Total	96	388
Success	84 (all types) / 93 (any type)	217
Timeout	10 (any type) / 1 (all types)	28
Inapplicable	N/A	139
Failure	2 (0)	4 (0)

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

verify most of the new integer operations in WebAssembly's 2.0 version, which is currently in draft status [Rossberg(2023)]. We choose these rules because WebAssembly uses integers for addressing computations, which means that logical issues in integer codegen can lead to security vulnerabilities. We verify `aarch64` rules because this backend is less well-tested than `x86-64`. The ARM backend rules we *do not* verify fall into four categories: (1) i128 types; (2) floating point; (3) SIMD (vector) instructions; and (4) side effects and control flow. We discuss further in Section 4.6.

Verification requires 182 total annotations (1075 lines of code). For some ISA terms, we modify or cross-reference formal semantics from SAIL-ISLA [Armstrong et al.(2019), Armstrong et al.(2021)], a symbolic execution engine for ISAs. For Cranelift IR and external Rust terms, we refer to WebAssembly's specification, Cranelift documentation, and the external Rust definitions.

In total, our verification effort covers 96 distinct rules with 388 type invocations, since each rule is tested against 1 to 10 possible type assignments. For most rules, we consider all Cranelift-supported integers up to 64 bits (i.e., `i8`, `i16`, `u/i32`, and `u/i64`), though we note that WebAssembly 1.0 only supports 32-bit and 64-bit numbers. `rustc_codegen_cranelift`, an alternative backend for the Rust language,

uses the narrower types VeriISLE supports [Nelson(2020), Baron et al.(2023)].

Table 4.1 shows the verification results for all 388 total type invocations. Recall that the six verification failures do not represent real bugs, since the context in which they are used does not require bitvector equivalence. With custom verification conditions, these rules verify successfully. 360 of the 388 invocations complete, in sum, within 5 minutes on a laptop.[11] The 10 rules that timeout on some type instantiations contain multiplication, division, remainder, and `popcnt` operations on bitvectors, which are difficult for SMT solvers to reason about for wider widths [Jha et al.(2009)].[12] Each of these rules fails with a counterexample within 10 seconds if we inject a flaw in the rule logic.

4.4.2 What Proportion of Invoked Rules has VeriISLE Verified?

We instrument Cranelift to determine, on various targets, what proportion of invoked ISLE rules VeriISLE has verified. For the WebAssembly reference test suite, VeriISLE verifies 19.8% (50/253) of the unique ISLE rules used during compilation. (We use a version of the WebAssembly specification's test suite that corresponds to the language features in Wasm 1.0, which notably excludes SIMD instructions.) To assess our coverage on integer types narrower than those that Wasm supports, we repeat this experiment on the `rustc_codegen_cranelift` test suite, an alternative backend for the Rust compiler that uses Cranelift as its code generator [Nelson(2020), Baron et al.(2023)]. Verified rules make up 15.8% (24/152) of

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

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

the unique ISLE rules used during compilation. These numbers will grow as we enhance VeriISLE to additional memory operations and floating point (Section 4.6).

4.4.3 Can VeriISLE Detect Known Bugs?

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

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

In under one second on a laptop, VeriISLE detects a 2023 CVE in `x86-64` instruction lowering that permitted a WebAssembly sandbox escape (Section 4.1) [Crichton(2023)]. The reproduction requires 13 new annotations to support terms in the `x86-64` backend, which we had not previously covered (Section 4.4.1).

The bug appeared in this ISLE rule:[13]

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

This rule intends to take advantage of an `x86-64` addressing mode that allows

[13]Lightly edited for brevity

shifts to be computed within the instruction itself, before adding together address components. However, the core problem with this rule (Section 4.1) is that the LHS performs a shift on a 32-bit value (throwing away any bits that are shifted left beyond 32 bits), while the RHS performs the shift on a 64-bit value (throwing away bits shifted left beyond *64* bits), which lets the emitted shift modify bits beyond WebAssembly's effective address space.

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

VeriISLE provides the following counterexample:[14]

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

[14]Lightly edited for brevity.

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

#x0000_0000_70c0_6580 => #x0000_0003_70c0_6580
```

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

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

aarch64 unsigned divide CVE (moderate severity)

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

The ISLE rules that matched on constant divisors for both `udiv` and `sdiv` —unsigned and signed divide—used the term `imm` on the RHS. `imm` models an immediate value that can be encoded in a machine instruction itself, lowering

both the number of instructions and register pressure. At the time of this CVE, the ISLE signature for `imm` was:

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

This term's intention was to take the immediate's value as a `u64` and place it in a register. When trying to annotate this term and the terms for signed constant divisors, though, an issue was immediately clear: `imm` provides *no argument* for whether narrow values should be sign- or zero-extended. Annotating zero-extension causes signed division to fail; choosing sign-extension causes unsigned division to fail. In practice, the external Rust implementation sign-extended, so the bug surfaced in `udiv` instructions. The patched version of `imm` takes in an argument for the type of extension, and the rules for `udiv` and `sdiv` now successfully verify.[15]

aarch64 count-leading-sign bug

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

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

This rule matches on `cls` computations over 8-bit values. The RHS extends 8-bit `x`

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

to 32 bits using `zext32`, and then computes `a64_cls` on this wider value. Finally, it subtracts 24 bits $(32 - 8)$ to obtain the leading bit count on the narrow type. VeriISLE reports the following counterexample:

```
(lower (has_type I8 (cls [x|#b11111100]))) =>
(output_reg
  (a64_sub_imm I32
    (a64_cls I32 (zext32 [x|#b11111100])) 24))

#b00000101 => #b11111111
```

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

4.4.4 Can VeriISLE Find New Bugs?

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

Another addressing mode bug

VeriISLE discovered a new correctness bug in an `x86-64` addressing mode rule related to the one discussed in Section 4.4.3 (which was not identified by Cranelift engineers even in a subsequent close look at addressing mode rules). This rule was

identical except that it did not have an explicit `uextend` (line 3 in Section 4.4.3)—the same bug could surface on a direct load of a 32-bit address. Cranelift developers determined that the bug would not be triggered in practice because on 64-bit targets, all addresses should be 64-bit typed, and front ends generate code in this form. However, nothing in the compiler backend validated this IR invariant and the bug *could* be triggered if front-end implementations changed. Cranelift engineers patched this issue immediately after we notified them of VeriISLE's result.

Flawed negated constant rules

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

This is one of the buggy rules VeriISLE discovered:

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

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

bit)—checking for negated values as well doubles the number of possible constants.

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

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

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

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

Imprecise semantics for constants in Cranelift IR

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

optimizations caused by the same imprecise semantics.

A mid-end root cause analysis

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

A Cranelift engineer ran Souper—a superoptimizer for originally designed for LLVM [Sasnauskas et al.(2018), Mukherjee et al.(2020)]—on a subset of Cranelift IR to look for additional optimization opportunities. Souper found that Cranelift was missing the boolean rewrite `or(and(x, y), not(y))== or(x, not(y))`. To port this to ISLE, the engineer wrote a new rule with an explicit guard to check the for a bitwise-not between constants `y` and `z`:[16]

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

This rule passed code review and was merged, but broke an integration test with a `wasm trap` error that did not point to a root cause. Before the Cranelift engineers were able to complete a manual investigation, we extended VeriISLE analyze this rule (e.g., added annotations for mid-end terms) in under two hours. VeriISLE produced the following counterexample:[17]

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

[16] Lightly edited for clarity and brevity.
[17] Example truncated to 1 bit for brevity.

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

VeriISLE surfaces a subtle bug related to the semantics of ISLE's `if` construct. Recall that terms in ISLE are partial functions. The semantics of ISLE's terms with external Rust implementations are that a match should continue if the return value is `Some(...)` and should not match if any LHS term returns `None`.

The prior rule's `if` desugars to this `if-let` guard with a wildcard for the left-hand pattern:

```
(if (u64_eq zk (u64_not y))) =>
  (if-let _ (u64_eq zk (u64_not y)))
```

Deceptively, because the Rust external definition of term `u64_eq` in the prior rule returned `Some(false)` instead of `None` (that is, the boolean was *defined*, just *false*) this guard as written *always* allowed the match to proceed!

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

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

ISLE's own program representation: VeriISLE was able to root-cause this bug because it must reason about core ISLE semantics.

4.5 Related Work

Compiler verification. Compiler verification research falls into two broad categories: lightweight verification of (parts of) existing compilers using solvers (e.g., [Kundu et al.(2009), Lerner et al.(2005), Lerner et al.(2003)]), and clean-slate, foundational verification using proof assistants [Bertot and Castéran(2013)] (e.g., CompCert [Leroy(2009a), Kumar et al.(2014)]). Foundational verification provides end-to-end correctness guarantees at the cost of time and performance: typically, such verification takes experts many years [Stewart et al.(2015)], and makes serious optimizations impractical. There are manually verified lowering passes for CompCert [Leroy(2009b)] and CakeML [Tan et al.(2019), Fox et al.(2017)], but not for production compilers that consider performance a first-class concern.

Other works use solver-backed methods to verify portions of industrial compilers. Most closely related to VeriISLE, Alive [Lopes et al.(2015)] verifies LLVM [Lattner and Adve(2004)] peephole optimization rules written in a DSL. Alive's main challenge is undefined behavior; in contrast, VeriISLE need not reason about undefined behavior, but must instead reconcile IR and ISA types. Further afield, Alive2 [Lopes et al.(2021)] does translation validation on LLVM IR, and VeRA [Brown et al.(2020)] verifies range analysis in the Firefox JavaScript engine. Finally, Jitterbug [Nelson et al.(2020)] verifies lowering from BPF, a setting where instruction selection entails simple "macro expansion" of one instruction at a time.

WebAssembly verification. VeriWasm proves that individual binaries do not violate Wasm's safety guarantees [Johnson et al.(2021)]. VeriWasm does not prove *compiler* correctness, though, and places restrictions on how Wasm compilers can emit native code.[18] In [Bosamiya et al.(2022)], the authors present a non-optimizing compiler to `x86-64` that is verified to preserve sandbox safety, and a non-optimizing compiler from Wasm to Rust; in contrast, we verify the correctness of a production, optimizing compiler.

There is also work on mechanizing the Wasm specification [Watt(2018)] and formalizing Wasm in the K framework [Hjort(2020)]. Other verification efforts look beyond the language and compiler: WaVE [Johnson et al.(2023)] verifies that interactions between the Wasm runtime and the host OS preserve safety guarantees; SecWasm [Bastys et al.(2022)] extends Wasm's guarantees using information flow control; [Protzenko et al.(2019)] bring verified cryptography to Wasm; and CT-Wasm extends Wasm itself with constant-time guarantees [Watt et al.(2019)].

Synthesizing instruction selectors. The complexity of instruction selection has inspired work on automatically generating rules based on machine-language semantics. Because of their focus on portability vs. correctness, many instruction selector generators use *ad hoc* search procedures instead of solver-aided techniques [Hoover and Zadeck(1996), Cattell(1980), Ceng et al.(2005), Dias and Ramsey(2010)]. Others use solver-aided synthesis: LibFIRM [Buchwald et al.(2018)], for example, uses SMT to synthesize new rules that cover about 75% of input instructions, while using an existing, handwritten rule set for the rest. [Daly et al.(2022)] uses a solver to generate high-coverage

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

selection simple rules for diverse target architectures. Rake [Ahmad et al.(2022)] synthesizes lowering rules from Halide [Ragan-Kelley et al.(2013b)] to digital signal processor ISAs, but its focus is on capturing complex data movement mechanics within vector registers instead of general-purpose instruction semantics. Though many compilers use a DSL to express instruction selection rules, to our knowledge VeriISLE is the first tool for verifying existing rules by modeling DSL semantics.

Formal semantics for ISAs. Several efforts formalize ISA semantics, including the SAIL language [Armstrong et al.(2019)] and the K Framework [Dasgupta et al.(2019)]. In the future, we will extend VeriISLE to exploit these existing semantic models.

4.6 Future Work

VeriISLE annotations are currently trusted. We can address this issue by deriving certain annotation from existing formal models. For example, VeriISLE can integrate SAIL semantics for `aarch64` [Armstrong et al.(2019)] and K framework semantics for `x86-64` [Dasgupta et al.(2019)]. While neither Cranelift IR nor external Rust term definitions have formal semantics, we can raise assurance in our specifications by, for example, verifying them against their external Rust implementations [Astrauskas et al.(2019), Baranowski et al.(2018), Reid et al.(2020)].

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

extended to automatically reason about rule priorities and to cover other backends and the mid-end optimizer.

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

4.7 Chapter Summary

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

4.8 Chapter Acknowledgements

Thank you to Jamey Sharp, Nick Fitzgerald, Trevor Elliott, Björn Roy Baron, Till Schneidereit, other members of the Bytecode Alliance, and participants in the Foundations of WebAssembly Dagstuhl seminar for feedback on the work in this chapter.

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

The three projects in this dissertation all highlight the potential for lightweight formal methods to free systems engineers from having to choose between efficiency and correctness. There is a bright future for formal solutions to practical problems in systems programming, from applications in real-world compilers to building new paradigms for reasoning about correctness in systems programs.

Production compilers do far more than lower instructions—and we can raise trust in increasingly complex passes. For example, the Cranelift compiler now uses an e-graph based mid-end optimizer—making it the first large-scale production compiler (to their knowledge [Bytecode Alliance(2023a)]) to use e-graphs as a unified optimization framework and to share a term rewriting DSL (ISLE) across mid-end optimizations and instruction selection. Future work can extend SMT-based verification of rewrite rules to compositionally verify interacting components of large production compilers. There is also rich potential for work that combines SMT-based verification with complementary approaches, such a fuzzing and property-based testing against reference implementations and observable run-time behavior.

There is also a need for new automated formal methods techniques for reasoning about control-flow transformations, such as loop-invariant code motion and global value numbering, in production optimizing compilers. New research could extend SMT-based verification systems to create infrastructure for *composable* compiler specification and verification. Security-critical compiler bugs have arisen due to missed interactions between passes: for example, a lowering pass may incorrectly assume an earlier optimization zeroes out the upper bits of a narrow

value. Future systems may need to design new, flexible specification paradigms to find correctness issues at the interfaces between inter-related compiler components. This work is part of a broader push toward multi-language compiler correctness [Patterson et al.(2022)] and gradual verification [Bader et al.(2018)], where systems can opt-in to varying levels of specification as they mature.

There is currently an opportunity for rethinking the mechanisms for integrating across compiler verification tools, especially in the context of specifications for ISAs and compiler intermediate representations. Automated compiler verification is a fast-growing area, and different frameworks and tools for reasoning about program representations abound [Lopes et al.(2021), Armstrong et al.(2019), Armstrong et al.(2021), Dasgupta et al.(2019)]. Interacting between these systems and semantics, though, is difficult and can require building one-off translators or manually copying semantics. Future research will rely on infrastructures that allows users to specify key design-space considerations with interoperability and reuasbility as first-order goals.

Infusing lightweight formal methods into the compiler and systems programming stacks has the potential to give both engineers and end-users access to more reliable and efficient systems.

BIBLIOGRAPHY

[Abate et al.(2018)] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *International Conference on Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-96145-3_15

[Agache et al.(2020)] Alexandru Agache et al. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *USENIX Symposium on Networked System Design and Implementation (NSDI)*.

[Ahmad et al.(2022)] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 2022. Vector Instruction Selection for Digital Signal Processors Using Program Synthesis. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3503222.3507714

[Allen and Kennedy(1987)] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[Alliance(2023)] Bytecode Alliance. 2023. ISLE Language Reference. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/isle/docs/language-reference.md.

[Alvanos and Trancoso(2016)] Michail Alvanos and Pedro Trancoso. 2016. Video SIMDBench: Benchmarking the compiler vectorization for multimedia applications. In *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE, 168–175.

[Armstrong et al.(2019)] 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 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3290384

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

[Arteaga et al.(2022)] Javier Cabrera Arteaga, Nicholas Fitzgerald, Martin Monperrus, and Benoit Baudry. 2022. Wasm-mutate: Fuzzing WebAssembly Compilers with E-Graphs. In *E-Graph Research, Applications, Practices, and Human-factors Symposium*. https://www.jacarte.me/assets/pdf/wasm_mutate.pdf

[Astrauskas et al.(2019)] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3360573

[Bader et al.(2018)] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://link.springer.com/chapter/10.1007/978-3-319-73721-8_2

[Baranowski et al.(2018)] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*.

[Barnett et al.(2005)] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. https://doi.org/10.1007/978-3-540-30569-9_3

[Baron et al.(2023)] Björn Roy Baron et al. 2023. Cranelift codegen backend for Rust. https://github.com/bjorn3/rustc_codegen_cranelift

[Barrett et al.(2010)] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT)*. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[Barthe et al.(2013)] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Principles and Practice of Parallel Programming (PPoPP)*.

[Bastys et al.(2022)] Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. 2022. SecWasm: Information Flow Control for WebAssembly. In *Static Analysis*.

[Benanav et al.(1987)] Dan Benanav, Deepak Kapur, and Paliath Narendran. 1987. Complexity of matching problems. *Journal of Symbolic Computation* 3, 1 (1987), 203–216.

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

[Bornholt et al.(2016)] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[Bosamiya et al.(2022)] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe multilingual software sandboxing using WebAssembly. In *USENIX Security Symposium*. `https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya`

[Brown et al.(2020)] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Buchwald et al.(2018)] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. `https://doi.org/10.1145/3168821`

[Bytecode Alliance(2023a)] Bytecode Alliance. 2023a. Cranelift. `https://cranelift.dev`.

[Bytecode Alliance(2023b)] Bytecode Alliance. 2023b. The Cranelift compiler. `https://github.com/bytecodealliance/wasmtime/tree/main/cranelift`.

[Bytecode Alliance(2023c)] Bytecode Alliance. 2023c. Wasmtime: A fast and secure runtime for WebAssembly. `https://wasmtime.dev`.

[Cadar et al.(2008)] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[Cadence Design Systems, Inc.(2020)] Cadence Design Systems, Inc. 2020. Tensilica Customizable Cores. https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

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

[Cattell(1978)] R G G Cattell. 1978. *Formalization and Automatic Derivation of Code Generators*. Ph. D. Dissertation. Carnegie Mellon University. https://apps.dtic.mil/sti/pdfs/ADA058872.pdf.

[Ceng et al.(2005)] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. 2005. C Compiler Retargeting Based on Instruction Semantics Models. In *Design, Automation & Test in Europe (DATE)*.

[Chong et al.(2020)] Nathan Chong et al. 2020. Code-Level Model Checking in the Software Development Workflow. In *ICSE-SEIP*.

[Clarke et al.(2004)] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[Cowan et al.(2020)] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.

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

[Crichton(2022b)] Alex Crichton. 2022b. Miscompilation of constant values in division on AArch64. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-7f6x-jwh5-m9r4.

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

[Crichton(2023)] Alex Crichton. 2023. Guest-controlled out-of-bounds read/write

on x8664. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8.

[Daly et al.(2022)] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. 2022. Synthesizing Instruction Selection Rewrite Rules from RTL using SMT. In *Formal Methods in Computer-Aided Design (FMCAD)*. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_20

[Dasgupta et al.(2019)] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314601

[De Moura and Bjørner(2008)] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://dl.acm.org/doi/10.5555/1792734.1792766

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

[Dias and Ramsey(2010)] João Dias and Norman Ramsey. 2010. Automatically Generating Instruction Selectors Using Declarative Machine Descriptions. https://doi.org/10.1145/1706299.1706346

[Driesen and Hölzle(1996)] Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[Eén and Sörensson(2003)] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver.

[Experimental(2021)] Facebook Experimental. 2021. MIRAI. https://github.com/facebookexperimental/MIRAI.

[Eyerman and Eeckhout(2010)] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *International Symposium on Computer Architecture (ISCA)*. 362–370.

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

[Fallin(2021b)] Chris Fallin. 2021b. RFC: Design of the ISLE Instruction-Selector DSL. https://github.com/bytecodealliance/rfcs/pull/15.

[Fallin(2023)] Chris Fallin. 2023. Cranelift's Instruction Selector DSL, ISLE: Term-Rewriting Made Practical. https://cfallin.org/blog/2023/01/20/cranelift-isle/.

[Fox et al.(2017)] Anthony Fox, Magnus O Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3018610.3018621

[Franchetti and Püschel(2008)] Franz Franchetti and Markus Püschel. 2008. Generating SIMD Vectorized Permutations. In *International Conference on Compiler Construction (CC)*.

[Franchetti et al.(2006)] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2006. A rewriting system for the vectorization of signal transforms. In *International Conference on High Performance Computing for Computational Science (VECPAR)*.

[Galois, Inc(2020)] Galois, Inc. 2020. Crux: Introducing our new open-source tool for software verification. https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/.

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

[Gonzalez(2000)] Ricardo E Gonzalez. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro* 20, 2 (2000), 60–70.

[Guennebaud et al.(2010)] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[Gurfinkel et al.(2015)] Arie Gurfinkel et al. 2015. The SeaHorn Verification

Framework. In *International Conference on Computer-Aided Verification (CAV)*.

[Haas et al.(2017)] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

[Hjort(2020)] Rikard Hjort. 2020. Formally Verifying WebAssembly with KWasm. https://odr.chalmers.se/server/api/core/bitstreams/a06be182-a12e-46ce-94d3-cff7a5dc42ba/content

[Hoare(1969)] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. In *Communications of the ACM (CACM)*. https://doi.org/10.1145/363235.363259

[Hoover and Zadeck(1996)] Roger Hoover and Kenneth Zadeck. 1996. Generating Machine Specific Optimizing Compilers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/237721.237779

[Ishizaki et al.(2000)] Kazuaki Ishizaki et al. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[Jang et al.(2014)] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security(NDSS)*.

[Jha et al.(2009)] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. 2009. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-642-02658-4_53

[Johnson et al.(2023)] 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 *IEEE Security and Privacy (Oakland)*. https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00114

[Johnson et al.(2021)] 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 *Network and Distributed System Security(NDSS)*. `https://cseweb.ucsd.edu/~lerner/papers/wasm-sfi-ndss2021.pdf`

[Joshi et al.(2002)] Rajeev Joshi, Greg Nelson, and Keith H. Randall. 2002. Denali: A Goal-directed Superoptimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Jung et al.(2017)] Ralf Jung et al. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[Jung et al.(2019)] Ralf Jung et al. 2019. Stacked Borrows: An Aliasing Model for Rust. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[Kamil et al.(2016)] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Kenton Varda(2018)] Kenton Varda. 2018. WebAssembly on Cloudflare Workers. `https://blog.cloudflare.com/webassembly-on-cloudflare-workers/`.

[Kumar et al.(2014)] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. `https://doi.org/10.1145/2578855.2535841`

[Kundu et al.(2009)] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. `https://doi.org/10.1145/1542476.1542513`

[Kyrtatas et al.(2015)] Nikolaos Kyrtatas, Daniele G. Spampinato, and Markus Püschel. 2015. A Basic Linear Algebra Compiler for Embedded Processors. In *Design, Automation & Test in Europe (DATE)*.

[Larsen and Amarasinghe(2000)] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets.

In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Lattner and Adve(2004)] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665

[Lerner et al.(2003)] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically Proving the Correctness of Compiler Optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/781131.781156

[Lerner et al.(2005)] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1047659.1040335

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

[Leroy(2009b)] Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

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

[Lopes et al.(2015)] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with Alive. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Lopes and Regehr(2018)] Nuno P. Lopes and John Regehr. 2018. Future Directions for Optimizing Compilers. In *ArXiV*. https://arxiv.org/pdf/1809.02161.pdf

[Lu and Hu(2019)] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *CCS*.

[Mainland et al.(2013)] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2013. Exploiting Vector Instructions with Generalized Stream Fusion. In *ACM International Conference on Functional Programming (ICFP)*.

[Maksimović et al.(2021)] Petar Maksimović et al. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *International Conference on Computer-Aided Verification (CAV)*.

[Martelli and Montanari(1982)] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*. `https://doi.org/10.1145/357162.357169`

[McCarthy and Painter(1966)] John McCarthy and James A. Painter. 1966. Correctness of a compiler for arithmetic expressions.

[McFarlin et al.(2011)] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *Proceedings of the International Conference on Supercomputing*.

[Mendis and Amarasinghe(2018)] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. `https://doi.org/10.1145/3276480`

[Milanova et al.(2004)] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise Call Graphs for C Programs with Function Pointers. In *Automated Software Engineering (ASE)*.

[Milner and Weyhrauc(1972)] R. Milner and R. Weyhrauc. 1972. Proving compiler correctness in a mechanized logic. In *Proc. 7th Annual Machine Intelligence Workshop*.

[Morrisett et al.(1999)] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[Moskewicz et al.(2001)] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*. https://doi.org/10.1145/378239.379017

[Mukherjee et al.(2020)] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-Based Pruning for Speeding up Superoptimization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3428245

[Mur-Artal et al.(2015)] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics* 31, 5 (2015), 1147–1163.

[Mur-Artal and Tardós(2017)] Raul Mur-Artal and Juan D Tardós. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262.

[Nandi et al.(2020)] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Nelson(1980)] Greg Nelson. 1980. *Techniques for program verification*. Ph. D. Dissertation. Stanford University.

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

[Nelson et al.(2020)] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://dl.acm.org/doi/abs/10.5555/3488766.3488769

[Newcomb et al.(2020)] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[Niemetz et al.(2019)] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. 2019. Towards Bit-Width-Independent Proofs in SMT Solvers. In *International Conference on Automated Deduction (CADE)*. https://doi.org/10.1007/978-3-030-29436-6_22

[Nuzman et al.(2006)] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Padlewski(2017)] Piotr Padlewski. 2017. Devirtualization in LLVM. In *SPLASH Companion*.

[Pat Hickey(2019)] Pat Hickey. 2019. Lucet Takes WebAssembly Beyond the Browser — Fastly. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

[Patterson et al.(2022)] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. https://doi.org/10.1145/3519939.3523703

[Paul and Meyer(2007)] JoAnn M Paul and Brett H Meyer. 2007. Amdahl's law revisited for single chip systems. *International Journal of Parallel Programming* 35, 2 (2007), 101–123.

[Peleg and Polikarpova(2020)] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *European Conference on Object-Oriented Programming (ECOOP)*.

[Phothilimthana et al.(2019)] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodík. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[Protzenko et al.(2019)] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *IEEE Symposium on Security and Privacy (SP)*. https://doi.ieeecomputersociety.org/10.1109/SP.2019.00064

[Puschel et al.(2005)] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[Ragan-Kelley et al.(2013a)] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013a. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Ragan-Kelley et al.(2013b)] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013b. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. `https://doi.org/10.1145/2491956.2462176`

[Ramos(2015)] David A. Ramos. 2015. *Under-constrained symbolic execution: correctness checking for real code*. Ph. D. Dissertation. Stanford University. `https://searchworks.stanford.edu/view/11061347`.

[Reid et al.(2020)] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards making formal methods normal: meeting developers where they are. `https://arxiv.org/abs/2010.16345`. In *ArXiV*.

[Ren et al.(2006)] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing Data Permutations for SIMD Devices. 118–131. `https://doi.org/10.1145/1133981.1133996`

[Ringer et al.(2020)] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2020. QED at Large: A Survey of Engineering of Formally Verified Software. In *Computing Research Repository (CoRR)*. `https://arxiv.org/abs/2003.06458`

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

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

[Russell(2008)] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*.

[Sasnauskas et al.(2018)] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. In *ArXiV*. `https://arxiv.org/abs/1711.04422`

[Solar-Lezama et al.(2006)] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[Spampinato et al.(2018)] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.

[Stewart et al.(2015)] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. `https://doi.org/10.1145/2775051.2676985`

[Strasdat(2015)] Hauke Strasdat. 2015. Sophus Project Website. `https://strasdat.github.io/Sophus/`.

[Strasdat et al.(2011)] Hauke Strasdat, Andrew J Davison, JM Martìnez Montiel, and Kurt Konolige. 2011. Double window optimisation for constant time visual SLAM. In *IEEE International Conference on Computer Vision (ICCV)*. 2352–2359.

[Sumikura et al.(2019)] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. 2019. OpenVSLAM: A Versatile Visual SLAM Framework. In *Proceedings of the 27th ACM International Conference on Multimedia* (Nice, France) *(MM '19)*. ACM, New York, NY, USA, 2292–2295. `https://doi.org/10.1145/3343031.3350539`

[Sweeney(2016)] Chris Sweeney. 2016. Theia Multiview Geometry Library: Tutorial & Reference. `http://theia-sfm.org`.

[Tan et al.(2019)] Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler

backend. *Journal of Functional Programming* 29 (2019). https://cakeml.org/jfp19.pdf

[Tate et al.(2009)] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[Toman et al.(2015)] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *Automated Software Engineering (ASE)*.

[Torlak and Bodik(2014)] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[VanHattum et al.(2020)] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2020. A Synthesis-Aided Compiler for DSP Architectures (WiP Paper). In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[VanHattum et al.(2021)] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3445814.3446707

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

[Visser et al.(1998)] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/289423.289425

[Vocke et al.(2017)] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. In *ACM Transactions on Architecture and Code Optimization (TACO)*.

[Watt(2018)] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3167082

[Watt et al.(2019)] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-driven secure cryptography for the web ecosystem. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3290390

[Weiss et al.(2019)] Aaron Weiss et al. 2019. Oxide: The Essence of Rust. https://arxiv.org/abs/1903.00982.

[Willsey et al.(2021)] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[Wolff et al.(2021)] Fabian Wolff, Aurel Bílý, et al. 2021. Modular Specification and Verification of Closures in Rust. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[Xu et al.(2014)] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. 2014. MSL: A Synthesis Enabled Language for Distributed Implementations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[Yang et al.(2011)] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993532

[Yavits et al.(2014)] Leonid Yavits, Amir Morad, and Ran Ginosar. 2014. The effect of communication and synchronization on Amdahl's law in multicore systems. 40, 1 (2014), 1–16.

[Yotov et al.(2003)] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 63–76.