Translation Validation for LLVM's AArch64 Backend

RYAN BERGER, NVIDIA, USA
MITCH BRILES, University of Utah, USA
NADER BOUSHEHRINEJAD MORADI, University of Utah, USA
NICHOLAS COUGHLIN, Defence Science and Technology Group, Australia and University of Queensland, Australia
KAIT LAM, Defence Science and Technology Group, Australia and University of Queensland, Australia
NUNO P. LOPES, INESC-ID, Portugal and Instituto Superior Técnico - University of Lisbon, Portugal
STEFAN MADA, University of Utah, USA
TANMAY TIRPANKAR, University of Utah, USA
JOHN REGEHR, University of Utah, USA

LLVM's backends translate its intermediate representation (IR) to assembly or object code. Alongside register allocation and instruction selection, these backends contain many analogues of components traditionally associated with compiler middle ends: dataflow analyses, common subexpression elimination, loop invariant code motion, and a first-class IR—MIR, the "machine IR." In effect, this kind of compiler backend is a highly optimizing compiler in its own right, with all of the correctness hazards entailed by a million lines of intricate C++. As a step towards gaining confidence in the correctness of work done by LLVM backends, we have created ARM-TV, which formally verifies translations between LLVM IR and AArch64 (64-bit ARM) code. Ours is not the first translation validation work for LLVM, but we have advanced the state of the art along multiple fronts: ARM-TV is a checking validator that enforces numerous ABI rules; we have extended Alive2 (which we reuse as a verification backend) to deal with unstructured mixes of pointers and integers that are typical of assembly code; we investigate the tradeoffs between hand-written AArch64 semantics and those derived mechanically from ARM's published formal semantics; and, we have used ARM-TV to discover 45 previously unknown miscompilation bugs in this LLVM backend, most of which are now fixed in upstream LLVM.

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

Additional Key Words and Phrases: translation validation, lifting, semantics, application binary interface

ACM Reference Format:
Ryan Berger, Mitch Briles, Nader Boushehrinejad Moradi, Nicholas Coughlin, Kait Lam, Nuno P. Lopes, Stefan Mada, Tanmay Tirpankar, and John Regehr. 2025. Translation Validation for LLVM's AArch64 Backend. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 369 (October 2025), 26 pages. https://doi.org/10.1145/3763147

Authors' Contact Information: Ryan Berger, NVIDIA, Santa Clara, USA, ryanbberger@gmail.com; Mitch Briles, University of Utah, Salt Lake City, USA, mitch.briles@utah.edu; Nader Boushehrinejad Moradi, University of Utah, Salt Lake City, USA, nader.bushehri@gmail.com; Nicholas Coughlin, Defence Science and Technology Group, Brisbane, Australia and University of Queensland, Brisbane, Australia, n.coughlin@uq.edu.au; Kait Lam, Defence Science and Technology Group, Brisbane, Australia and University of Queensland, Brisbane, Australia, kait.lam@uq.edu.au; Nuno P. Lopes, INESC-ID, Lisbon, Portugal and Instituto Superior Técnico - University of Lisbon, Lisbon, Portugal, nuno.lopes@tecnico.ulisboa.pt; Stefan Mada, University of Utah, Salt Lake City, USA, Stefan.Mada@utah.edu; Tanmay Tirpankar, University of Utah, Salt Lake City, USA, tanmay.tirpankar@utah.edu; John Regehr, University of Utah, Salt Lake City, USA, regehr@cs.utah.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART369
https://doi.org/10.1145/3763147

1 Introduction

Several years ago, one of us was discussing compiler-related challenges with a software architect at a large automobile manufacturer, who expressed a desire to track LLVM's top-of-tree—its latest development version—as part of his company's toolchain for 64-bit ARM code. In contrast, the current best practice for safety-critical software projects is to stick with a single compiler version throughout the entire lifespan of a product; the goal is not to use a correct compiler, but rather to use a compiler whose deficiencies (and their workarounds) are well-understood. While mulling over the question "How might we be able to help safety-critical software engineers trust fresh versions of LLVM?" we decided to start working on ARM-TV, which uses translation validation [14, 15, 21, 22] to prove that individual executions of LLVM's AArch64 backend are correct, or wrong.

Why would we focus our attention on an LLVM backend? For one thing, LLVM for AArch64 is broadly important—it is the code generator for the operating system and other important software that runs on three billion Android devices and two billion iOS devices. For another, a modern compiler backend does a lot of work. Traditionally, the primary jobs of a compiler backend were relatively modest: it was responsible for register allocation, instruction selection, and conforming to an application binary interface (ABI)—a specification that allows separately-compiled code modules to interoperate. However, in a modern compiler such as LLVM, a backend is additionally responsible for performing numerous optimizations, including some that perform inline substitution of library functions and others that change the memory behavior or control flow of the code being compiled. In this example—which ARM-TV validates in about 0.5 s—both kinds of transformation are performed:

```
define i32 @func(ptr %p0, ptr %p1) {
  %4 = call i32 @memcmp(ptr %p0, ptr %p1, i64 8)
  %5 = icmp slt i32 %4, 0
  br i1 %5, label %bb1, label %bb2
bb1:
  ret i32 7
bb2:
  ret i32 11
}
```

$\longrightarrow$

```
func:
    ldr     x9, [x0]        ; load 8 bytes via %p0
    ldr     x10, [x1]       ; load 8 bytes via %p1
    mov     w8, #11         ; w8 = 11
    rev     x9, x9          ; reverse byte order
    rev     x10, x10        ; reverse byte order
    cmp     x9, x10         ; compare, set flags
    mov     w9, #7          ; w9 = 7
    csel    w0, w9, w8, lo  ; ret val = 7 or 11
    ret
```

In the LLVM IR on the left, memcmp(), a C library function, is invoked to compare the 8-byte memory objects that function arguments `%p0` and `%p1` point to, returning a positive, zero, or negative result depending on their lexical order. Then, the function returns 7 if the memcmp() result is negative, and 11 otherwise. Looking at the assembly, LLVM has inlined memcmp(). The `ldr` instructions load eight bytes in little endian fashion into each of registers `x9` and `x10`; the `rev` instructions reverse the byte order of both registers; and then, a single 64-bit integer comparison can determine lexical order, given big endian data.[1] The backend also replaces control flow with `csel`, a conditional select instruction.

Their willingness to perform interesting transformations makes LLVM backends difficult to adequately test. In fact, these backends are sophisticated enough that they have their own internal optimization pass structure and operate on their own "machine IR"—that is distinct from LLVM IR—that supports both SSA semantics before register allocation and non-SSA semantics after register allocation. Together, the "CodeGen" and "Target" subdirectories of the LLVM source tree implement the LLVM backends in about a million lines of C++.

[1]To implement this same optimization, the memcmp() implementation for early (32-bit ARM) Raspberry Pi boards used the `setend` instruction to briefly switch the entire processor into big endian mode. The people developing software such as QEMU and Valgrind declined to support `setend`. The `rev` instruction provides a cleaner alternative.

1.1 ARM-TV

ARM-TV is reasonably feature-complete with respect to user-mode code. It supports integer and floating point operations, the Neon vector ISA (instruction set architecture), memory operations, and function calls. To create it, we needed formal semantics for both LLVM IR and AArch64. LLVM has become a large, complex language with numerous subtle corner cases [7, 9, 27]. Its semantics are defined by its Language Reference,[2] an English-language document that can be frustratingly informal. To avoid replicating existing work on formalizing LLVM, we built ARM-TV on top of Alive2 [13], which has gained significant acceptance inside the LLVM community. For example, more than 900 issues in the LLVM project's GitHub issue tracker contain links to our online version of Alive2.[3] To support the free-form mixes of pointer and integer operations that are commonly seen in assembly code, we had to add support for *physical pointers*—those that are effectively just 64-bit indices into anywhere in the address space—to Alive2.

AArch64 is not subtle or complicated, but it is large: the ISA reference that we spent a lot of time with while building ARM-TV is 4000 pages long [10]. Happily, there is an alternative to formalizing AArch64 from its documentation: ARM's machine-readable architecture (MRA), which specifies the behavior of each instruction using ASL[4] (Architecture Specification Language). The ASL is authoritative, but meticulous: it contains far more complexity and detail—for example, regarding hardware exceptions, interrupts, and memory address translation—than is necessary or desirable for ARM-TV. This complexity threatened to make it impractical for us to reuse the MRA. To extract from it a concise summary of each instruction's user-mode behavior, we adapted ASLp [5], an existing partial evaluation tool for ASL, to specialize away aspects of the semantics that we were not interested in. Independently, we implemented about 1,400 instructions (as LLVM counts them) by hand, allowing us to perform—as far as we know—the first apples-to-apples comparison between mechanically derived and hand-written ISA semantics in terms of verification performance.

We found that in order to get good verification performance, we had to do significant optimization work. First, we wrote a vectorizer for ASLp that allows it to generate native vector operations from the scalarized instruction semantics that it takes as input. Second, we implemented several new optimizations within Alive2's memory subsystem specifically to support pointer behaviors seen in assembly code.

Most prior work on translation validation has prioritized demonstrating the absence of compiler bugs, for particular runs of some compiler. We have given equal emphasis to demonstrating the presence of bugs using translation validation, based on the premise that we would like to make LLVM better—by eliminating latent miscompilation bugs—even for program developers who do not, themselves, choose to use translation validation. In a bug-finding workflow, ARM-TV serves as a strong oracle for detecting miscompilations. In contrast, most prior compiler bug-finding work uses differential testing—executing two compilers' outputs, and comparing their effects—as an oracle [6, 12, 26]. Whereas differential testing can only detect a miscompilation if it is revealed by that one specific execution path through the generated code, translation validation accounts for all possible executions of it. Moreover, we designed ARM-TV to check as many ABI-level properties—such as whether the values of callee-saved registers are preserved—as possible. So far, we have found and reported 45 miscompilation defects, many of which have been fixed.

[2] https://llvm.org/docs/LangRef.html
[3] http://alive2.llvm.org
[4] https://developer.arm.com/Architectures/Architecture%20Specification%20Language

1.2 Refinement

When a program transformation results in the new program having a subset of the old program's behaviors, the transformation is a *refinement*. Ensuring that every transformation is a refinement is the usual top-level correctness criterion for a compiler. In some cases, such as lowering a 64-bit `add` instruction in LLVM IR to a 64-bit addition instruction in assembly language, the refinement is a trivial special case: an equivalence relation (i.e., the subset of behaviors is not a proper subset). However, compilers perform numerous transformations that are non-trivial refinements—these are irreversible because the meaning of the code is changed. For example, if a compiler for a concurrent programming language coalesces several atomic sections into a single, larger atomic section, it is likely to be eliminating interleavings that had previously been possible. If a compiler for C++ implements an int-typed variable using 32 bits of machine storage, it is discarding valid interpretations of the C++ program where the int was, for example, 16 bits or 64 bits. In the LLVM middle end and backends, non-trivial refinements are most often due to undefined behaviors. A simple example is lowering LLVM's `sub nuw` instruction (integer subtraction that is undefined when unsigned overflow occurs) to a standard hardware subtraction instruction, which has a defined result for all inputs. A key property of refinement is compositionality: if every individual transformation performed by a compiler is a refinement, then so is the entire compilation process.

1.3 A Representative Failure of Refinement

Consider this LLVM code:

```
%a = load <16 x i8>, ptr %p
%b = insertelement <16 x i8> %a, i8 %val, i32 %idx
store <16 x i8> %b, ptr %p
```

The first instruction loads a 16-wide vector of eight-bit elements from memory at address `%p` into `%a`. The second instruction creates a new vector, `%b`, that has the same contents as `%a`, except that the value at position `%idx` has been replaced with `%val`. Then, the last instruction writes this new vector back to memory at address `%p`.

The AArch64 backend translated this sequence of operations into:

```
strb    w1, [x0, w2, uxtw]  ; *(%p + %idx) = new byte value
```

Which simply updates a single byte in memory: `strb` is "store byte." Here, the bottom byte of `w1` contains the value to be stored, `x0` is a 64-bit register containing the base address of the vector, and `w2` is the bottom 32 bits of a 64-bit register containing the index of the element to be updated. The `uxtw` specifier indicates that `w2` should be zero-extended to 64 bits before being added to `x0`.

As long as `%idx` (and therefore `w2`) is in the range 0..15, there is no problem: the AArch64 code updates the correct lane of the vector in memory. On the other hand, if the index contains a larger value, then this assembly code will overwrite unrelated memory. To see that the translation is incorrect, we turn to the specification for `insertelement` in the LLVM Language Reference,[5] which states: "If idx exceeds the length of val for a fixed-length vector, the result is a poison value." Poison is a kind of deferred undefined behavior; informally, a poison value can be understood to be similar to an IEEE 754 quiet NaN: an undefined value that propagates contagiously, that should not be branched on or passed to external code. (For a detailed discussion of poison and its consequences, see Lee et al. [9].)

ARM-TV correctly flagged this translation as a failure of refinement. In practice, the ARM code seems like it could be dangerous: if the vector index is controlled by external program input, an attacker might be able to create an exploit by overwriting appropriate data elsewhere in the address space.

[5]https://llvm.org/docs/LangRef.html#insertelement-instruction

After we reported this bug (as issue #74248) it was fixed; the compiler now emits:

```
and     x8, x2, #0xf  ; reduce %idx modulo 16
strb    w1, [x0, x8]  ; *(%p + %idx) = new byte value
```

This clamps the index within the range 0..15, causing the vector element at (`%idx` mod 16) to be overwritten. This result, while not being useful when `%idx` $\geq$ 16, is a valid refinement of a poisoned vector. There are many instances in LLVM IR where its developers have preferred this kind of deferred undefined behavior. The "useless value, but not broadly harmful behavior" for erroneous situations allows operations—such as the `insertelement` instruction here—to be safely speculatively executed, enabling important optimizations such as loop invariant code motion.

A subtle failure of refinement such as this one would not necessarily be easy to find using differential testing, since the resulting memory corruption would need to be noticed and diligently backtracked to its root cause. Additionally, this bug had a sibling in LLVM's `extractelement` instruction—that we also found and reported—that would be even harder to detect using standard testing techniques because the out-of-bounds memory operation was a load, not a store.

1.4 Research Questions

Our evaluation, in Section 3, answers these questions:

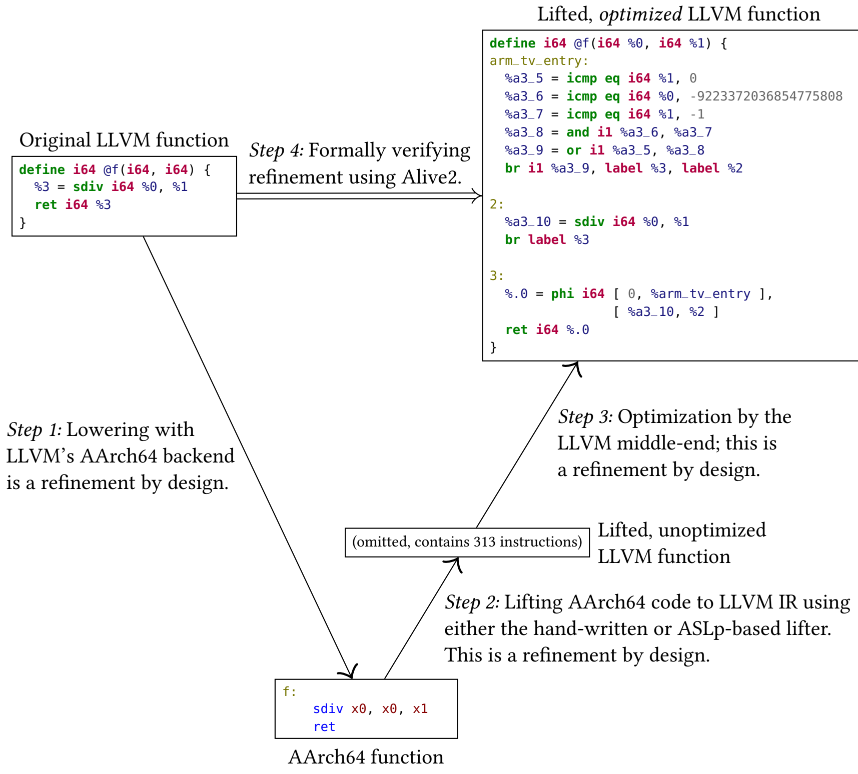
(RQ1) Can we create a practical translation validation tool for LLVM's AArch64 backend, without modifying LLVM?
(RQ2) Can we optimize Alive2 for assembly-style code that freely intermixes integers and pointers, that defeat many of its existing memory verification optimizations?
(RQ3) What are the tradeoffs between writing semantics for AArch64 instructions by hand and deriving semantics from ARM's machine-readable architecture description?
(RQ4) Does LLVM's AArch64 backend have latent miscompilation bugs that we can discover using translation validation?

2 Translation Validation by Lifting

ARM-TV assigns a formal semantics to AArch64 code by *lifting* it to LLVM IR. While this implementation choice had some drawbacks—that we discuss below—it had the significant advantages of allowing us to reuse Alive2 and also the LLVM middle-end optimization pipeline. Given a function to validate, the top-level steps performed by ARM-TV are:

(1) Invoke LLVM's AArch64 backend, lowering the function to assembly.
(2) Lift the assembly back to LLVM IR, being careful to preserve refinement. We offer a choice between a hand-written lifter that tends to produce idiomatic LLVM IR and a mechanically-generated lifter that makes direct use of ARM's machine-readable architecture [17].
(3) Run the LLVM middle end optimizer on the lifted IR. Typically, this makes it substantially more compact.
(4) Verify that the lifted and optimized LLVM IR refines the original IR using a version of Alive2 [13] that we modified to support an assembly-level memory model.

Fig. 1 illustrates these four steps. Steps 2 and 4 are where we spent our effort. Step 2, implemented using about 14,400 lines of C++, divides into two main parts. First, we need to generate glue code to emulate—at the LLVM level—the execution environment that ARM instructions find themselves in, including elements such as a stack and a register file. Second, we lift each ARM instruction into a collection of LLVM instructions that achieve the same effect. The rest of this section describes these steps in detail.


Lifted, *optimized* LLVM function
Original LLVM function
Step 4: Formally verifying refinement using Alive2.
Step 1: Lowering with LLVM's AArch64 backend is a refinement by design.
Step 3: Optimization by the LLVM middle-end; this is a refinement by design.
(omitted, contains 313 instructions)
Lifted, unoptimized LLVM function
Step 2: Lifting AArch64 code to LLVM IR using either the hand-written or ASLp-based lifter. This is a refinement by design.
AArch64 function

```
define i64 @f(i64, i64) {
  %3 = sdiv i64 %0, %1
  ret i64 %3
}
```

```
define i64 @f(i64 %0, i64 %1) {
arm_tv_entry:
  %a3_5 = icmp eq i64 %1, 0
  %a3_6 = icmp eq i64 %0, -9223372036854775808
  %a3_7 = icmp eq i64 %1, -1
  %a3_8 = and i1 %a3_6, %a3_7
  %a3_9 = or i1 %a3_5, %a3_8
  br i1 %a3_9, label %3, label %2

2:
  %a3_10 = sdiv i64 %0, %1
  br label %3

3:
  %.0 = phi i64 [ 0, %arm_tv_entry ],
              [ %a3_10, %2 ]
  ret i64 %.0
}
```

```
f:
    sdiv x0, x0, x1
    ret
```

Fig. 1. How ARM-TV performs translation validation. The top-level correctness property that we check is that the compiler's output refines its input. Thin arrows indicate transformations that are intended, by their respective authors, to be refinements. If each of them is a refinement, then the double arrow at the top must also be a refinement, since refinement is compositional. If this final refinement relation—which we check using Alive2—does not hold, then at least one of the three other purported refinements must have been defective, in which case either we or the LLVM developers have a bug to fix. Section 2.2 explains why the lifted code is so much longer than both the original and AArch64 functions.

2.1 Building an Execution Environment in LLVM for Lifted AArch64 Code

Both within a function and across function boundaries, ARM instructions communicate with each other using the stack and the register file. Within a function, communication is free-form, whereas across functions the protocols are spelled out in a 60-page ABI document: the "Procedure Call Standard for the Arm 64-bit Architecture (AArch64)."[6]

When ARM-TV lifts a function, it emits code that allocates LLVM-level memory to hold:

- 29 64-bit general-purpose registers
- a 64-bit stack pointer
- a 64-bit frame pointer
- a 64-bit link register
- a 64-bit zero register
- four 1-bit condition code flags
- 32 128-bit floating-point / vector registers

[6]https://github.com/ARM-software/abi-aa/releases/download/2023Q3/aapcs64.pdf

- a 4 KB stack

Lifted instructions communicate with each other using only these memory regions. This means that ARM-TV does not need to generate non-trivial SSA code, significantly simplifying our job and reducing the likelihood of lifter bugs. Furthermore, since the lifted code has a clean separation between different instructions, debugging the lifter is made easier. The program counter is absent from the lifted representation; lifted LLVM instructions pass control to their successors in the same way that ARM instructions do: either by implicit fallthrough to a successor, or explicit branching to a new target.

Consider this simple LLVM function:

```
define i64 @f(<8 x i8> %0) {
  %conv = bitcast <8 x i8> %0 to i64
  ret i64 %conv
}
```

It takes, as a parameter, an 8-way vector of 8-bit elements called `%0`, type-casts it into a 64-bit integer `%conv`, and then returns this value. Compiled to AArch64, this function is:

```
f:
    fmov   x0, d0
    ret
```

This code moves 64 bits of data from `d0`—an alias for the bottom half of the first 128-bit vector register—into `x0`, a 64-bit general-purpose register. At this level, the positions of arguments and the return value are implicit, as defined by the ABI. ARM-TV emits LLVM instructions that emulate the ABI; the entire resulting lifted function—before optimization by LLVM—contains 320 instructions, most of which are creating registers not used in this trivial example. If we manually remove the unneeded instructions, the lifted code is:

```
define i64 @f(<8 x i8> %0) {
  ; allocate memory for register X0, initialize it with non-deterministic contents
  %X0 = alloca i64, i64 1, align 8
  %a0_0 = freeze i64 poison
  store i64 %a0_0, ptr %X0, align 1

  ; allocate memory for register Q0, initialize it with non-deterministic contents
  %Q0 = alloca i128, i64 1, align 8
  %a0_58 = freeze i128 poison
  store i128 %a0_58, ptr %Q0, align 1

  ; glue code: store the function parameter into the bottom of Q0,
  ; the high bits remain non-deterministic
  %a0_99 = bitcast <8 x i8> %0 to i64
  store i64 %a0_99, ptr %Q0, align 1

  ; lifted implementation of "fmov x0, d0"
  %a3_0 = load i128, ptr %Q0, align 1
  %a3_1 = trunc i128 %a3_0 to i64
  store i64 %a3_1, ptr %X0, align 1

  ; lifted implementation of "ret"
  %a4_22 = load i64, ptr %X0, align 1
  ret i64 %a4_22
}
```

By initializing memory with "freeze poison," we leverage the underlying SMT solver to act adversarially with respect to elements of the AArch64 machine state that a compiled function is not supposed to rely upon. In contrast, if we merely zeroed out the machine state, or even initialized it to random values, we might miss bugs.

The important property of the glue code is that when it is combined with lifted ARM instructions, the result is an LLVM function that (assuming correct compilation) refines the original LLVM IR. And, indeed, after the lifted code is optimized by the LLVM middle-end, the resulting function is:

```
define i64 @f(<8 x i8> %0) {
  %a0_99 = bitcast <8 x i8> %0 to i64
  ret i64 %a0_99
}
```

Of course, this was a very simple example. In practice, we are rarely lucky enough to end up with the same code that we started with.

An earlier attempt. Rather than relying on LLVM to optimize lifted code, the first version of ARM-TV generated optimized Alive2 IR directly. For example, it generated high-quality SSA code, elided the computation of condition code flags that were never used, collapsed redundant extend and truncate instructions, etc. We did this because we wanted to avoid trusting the LLVM middle-end optimizer. However, our design choice resulted in several problems. First, generating optimized SSA code incurred algorithmic complexity in our lifter and also made the resulting code tricky to read and debug. Second, to improve the quality of the lifted code, we had to implement additional optimizations on Alive2 IR, creating still more complexity. Even so, our optimizer was much less powerful than LLVM's optimizer. Finally, since we do not have a serializer/deserializer for Alive2 IR, it was difficult to determine whether a given bug was in our lifter or in Alive2 itself. Generating LLVM IR instead of Alive2 IR works around all of these problems, but adds a massive amount of code to our trusted computing base. We have actually run into cases where ARM-TV detected a refinement failure, where the root cause was an LLVM middle-end bug, instead of the kind of LLVM backend bug that we are interested in, here. But this has not happened very many times. Our current set of implementation tradeoffs appears to be workable.

2.2 Preserving Refinement During Lifting

At first glance, LLVM IR is an attractive target for lifting: it provides a wide variety of instructions that are, overall, a reasonably close semantic match for AArch64 (and, of course, other ISAs). But there's a catch! In several important respects, the semantics of LLVM IR are specified more loosely than are the semantics of native ISAs. This is a deliberate design choice; the intent is to retain optimization power by avoiding early commitment to overly-tight semantics. In the normal compilation flow, when LLVM IR is lowered to an actual ISA, its loose semantics admit efficient refinements—this is the point. On the other hand, maintaining refinement while lifting AArch64 to the LLVM level is less straightforward: we must be extremely careful to preserve the tightly-specified meanings of the ARM instructions. As a concrete example, while each of LLVM's three basic shift instructions (`shl`, `ashr`, `lshr`) is refined by a single shift instruction in every popular modern ISA, the converse does not hold: none of these three LLVM instructions refines any machine-level shift instruction that we are aware of. We explain this situation in more detail below.

Handling undefined behaviors in individual LLVM instructions. Consider the example in Fig. 1. LLVM's signed integer division instruction—`sdiv`—has undefined behavior when the divisor is zero, or when the divisor is -1 and the dividend is `INT_MIN`. This LLVM instruction is refined by the AArch64 `sdiv` instruction, which returns the same result as the LLVM instruction for all of its defined cases, and returns zero for its undefined cases. However, when lifting the AArch64 instruction to LLVM, refinement can only be achieved by explicitly checking for the inputs that would otherwise trigger undefined behavior. This requires nine LLVM instructions!

When a value is shifted past its bitwidth (e.g., $x \ll 100$, where x is a 64-bit value), LLVM specifies that the result is a poison value. On the other hand, AArch64 specifies that the shift exponent is reduced modulo 64, requiring ARM-TV to emulate that behavior by adding extra LLVM instructions to the lifted code. Similar, but not necessarily identical, code would be required when lifting other ISAs.[7]

We also encountered refinement issues when lifting instructions such as `fcvtzu`: "floating-point convert to unsigned fixed-point, rounding toward zero." For floating point values that exceed the range of the target integer type, this AArch64 instruction saturates the result to the largest unsigned integer value. In contrast, the corresponding LLVM `fptoui` instruction produces a poison value when the result is not representable (after rounding) in the target type. Luckily, since 2020 LLVM has supported intrinsic functions that provide saturating floating-point to integer conversions, and these are what we emit in ARM-TV.

Early in the course of this project, we tested several publicly available AArch64 to LLVM lifters, in hopes of avoiding creating our own lifter. Alas, none of the lifters that we tried was correct without respect to refinement.

Memory model issues. In LLVM IR, pointers carry *provenance*: additional information that acts something like a capability, telling the compiler which memory regions that pointer is allowed to be used to access [7]. On the other hand, at the assembly level, any pointer can be used to access any part of the address space. During the development of ARM-TV, we had hoped that representing values that are stored in ARM registers as integers, and then converting them to pointers just before using them in a memory operation (which gives them wild-card provenance) would suffice to avoid provenance-related problems, but this turned out to be over-optimistic. We ended up needing to add a new assembly-level memory model to Alive2 that enforces an alternate set of memory rules for lifted code. We describe it in detail in Section 2.8.

Preventing the spread of poison values. For the most part, LLVM-level poison values are all-or-nothing; a value cannot be, for example, poisonous only in its sign bit. LLVM-level vectors are an exception: each lane can independently be poisonous or not. If a vector containing any poison lanes is type-cast to an integer type, the entire integer becomes poisonous. Early in ARM-TV's development, we were insufficiently cautious in casting between integers and vectors, leading to a handful of spurious failures of refinement. In general, this is a difficult problem to solve because, of course, at the assembly level there is no distinction between integers and vectors. Since it does not seem likely to us that a fully general solution to this problem exists (such as recovering a well-typed program from the assembly instructions emitted by LLVM), we simply try to keep the number of vector-to-integer casts to a minimum while lifting code.

2.3 Checking ABI Rules

Although many lifters from various assembly languages to LLVM IR have been created, ours is different in that it is designed to specifically check for translation failures that might otherwise go unnoticed. These include:

- As mentioned in the previous section, we store non-deterministic values into all elements of the ARM machine state that are not guaranteed by the ABI to hold specific values.
- We verify that the stack pointer is always 16-byte aligned, as required by the ABI.
- We verify that the lifted function restores the value of all callee-saved registers.

[7] https://devblogs.microsoft.com/oldnewthing/20230904-00/?p=108704

- We render the contents of all caller-saved registers non-deterministic before returning from a function call made by the lifted code.
- At the LLVM level, function parameters and the function return value may possess the "signext" or "zeroext" attributes. Although neither attribute has any semantics at the IR level, at the ARM level they allow the function being lifted to assume that non-register-sized parameters have been sign or zero-extended. Correspondingly, the function being lifted is obligated to sign- or zero-extend its return value, when the relevant return attribute is present. Also, of course, when the lifted function itself issues calls, obligations are placed on it by parameter attributes, and it may assume that the return value attributes have been implemented correctly.

A bug found using these checks—described in Section 3.5—was particularly interesting.

Where the ABI falls short. The AArch64 ABI document is somewhat limited in scope and fails to specify behaviors for a number of situations that come up in practice for us. For example, the AArch64 ISA can only load or store one, two, four, eight, or sixteen bytes at a time. However, at the LLVM level, it is possible to load and store values of any bitwidth. When an LLVM load or store does not match the bitwidth of an available memory operation, is it the responsibility of the store side to zero out enough extra bits to make the store fit into an available memory operation, or is it the responsibility of the load side to ignore garbage in these extra bits? The de facto answer appears to be the former, and this is what we have supported in ARM-TV, but it is not documented anywhere.

A second example of a gap in the ABI is in the calling convention for vectors: it only specifies how to pass vectors whose total size is 64 or 128 bits, and whose elements are 8, 16, 32, or 64 bits wide. Not only is the calling convention for other vector shapes unspecified, but we found that in practice the LLVM AArch64 backend behaves erratically. For example, a 1-way vector of Boolean values, which has type `<1 x i1>`, is passed in register `w0`, the bottom half of the first general-purpose register. A 2-way vector of Boolean values (`<2 x i1>`) is passed in `v0`, the first vector register. A 3-way vector of Boolean values (`<3 x i1>`) is split across three general-purpose registers: `w0`, `w1`, and `w2`. We inquired about this issue, and it does not appear to be the case that the LLVM community is prepared to guarantee a stable ABI for these situations. Thus, we have chosen to not support them in ARM-TV: we instead reject inputs that rely on under-specified vector behaviors.

2.4 Lifting Instructions by Hand

Every AArch64 instruction inspects a subset of the machine state and then makes changes to another subset. Some of these are quite simple, requiring only a few LLVM instructions to emulate their effect. On the other hand, complex vector instructions can require a lot of code: one variant of the `tbl` instruction—vectorized table lookup—turns into 157 LLVM instructions.

Based on descriptions found in the AArch64 ISA reference, we implemented around 1,400 instructions (using the somewhat fine-grained decomposition that LLVM uses internally) by hand. Although the individual instructions were not difficult, the large surface area of this instruction set made this into a time-consuming task.

2.5 Lifting Instructions Using ASLp

The ASLp partial evaluator [5] is well-suited for use in ARM-TV; it has been designed to process ASL, resulting in simplified instruction semantics that can be easily integrated with software reasoning tools such as ARM-TV, providing its output in a minimal, imperative language. Moreover, it is designed to do this while maintaining semantic equivalence with the original specification.

```
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
   bits(esize) element2 = Elem[operand2, e, esize];
   if sub_op then
      Elem[result, e, esize] = element1 - element2;
   else
      Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

(a) Relevant portion of ASL specification, handling vector addition and subtraction for combinations of element size and element count

```
V[0] = ZeroExtend(add_vec(
     V[0][0 +: 64], V[1][0 +: 64], 8), 128);
```

(b) Reduced ASL after partial evaluation

```
%a4_5 = load i128, ptr %Q0, align 1
%a4_6 = trunc i128 %a4_5 to i64
%a4_7 = load i128, ptr %Q1, align 1
%a4_8 = trunc i128 %a4_7 to i64
%a4_9 = bitcast i64 %a4_6 to <8 x i8>
%a4_10 = bitcast i64 %a4_8 to <8 x i8>
%a4_11 = add <8 x i8> %a4_9, %a4_10
%a4_12 = bitcast <8 x i8> %a4_11 to i64
%a4_14 = zext i64 %a4_12 to i128
store i64 %a4_14, ptr %Q0, align 1
```

(c) LLVM IR after translation

Fig. 2. Code snippets of the ASL, reduced ASL, and LLVM IR for the instruction `add v0.8b, v0.8b, v1.8b` (simplified)

Given an instruction encoding, ASLp [5] will partially evaluate the architecture specification and return the instruction's semantics with a configurable level of detail. This configuration is implemented through a variety of means, such as enforced invariants over the architecture's system registers, and it allowed us to customize the output of ASLp to suit the context of ARM-TV. For instance, we configured ASLp to produce instructions for AArch64's least privileged execution mode—informally its user-mode—by fixing certain bits in a processor state register. Partial evaluation fails where it cannot statically show that this invariant is preserved for an instruction. Additionally, ASLp's configurable inliner enables the extraction of certain common abstractions from the specification. For example, memory operations can be extracted without the internal semantic details of alignment and address translation by configuring the inlining logic to stop at high-level load and store operations. Getting abstractions from ASL to match up with the internal structure of our lifter was critical to the successful integration of the existing formal semantics into ARM-TV.

After successful partial evaluation, ASLp represents the resulting instruction semantics in a textual encoding of the reduced ASL abstract syntax tree. To aid translation, this subset of ASL uses very few fundamental operations: the basic scalar/vector integer and bitwise operations, along with a number of intrinsics for floating-point. Then, on the ARM-TV side, we parse this and translate it into equivalent LLVM IR. Alas, we had to implement this translator manually; there is not a formal specification for the meaning of ASL constructs. Thus, the work required in developing a working lifter is redirected from implementing instructions to implementing (a much smaller number of) reduced ASL structures.

Generally, these reduced ASL structures closely correspond to primitive operations in LLVM IR, with the caveat that care needs to be taken to ensure refinement is preserved (Section 2.2). Within the ASLp lifter, vector operations were given special attention because of the particulars of their specification in the ASL—this is discussed below. In total, the reduced ASL to LLVM translator is implemented in just under 2,000 lines of C++ code. This process can be seen in Fig. 2, showing the ASL, reduced ASL, and LLVM IR for a vector add instruction.

Optimizing for verification performance. Given the high degree of automation in ARM-TV, verification performance is crucial. We modified ASLp to make its semantics more suitable for automated verification. Initially, vector instructions from ASL resulted in an inefficient representation; the ASL specification describes vector instructions as loops, with the loop body slicing and concatenating 128-bit operands to perform a single vector element operation per iteration. ASLp would fully unroll these loops, resulting in a series of operations extracting each vector element, performing the operation, and then concatenating the results into a final bitvector. This undesired scalarization led to complex LLVM IR and also poor verification performance.

We remedied this situation by extending ASLp to emit vectorized variants of its primitive operations. Such an extension was non-trivial as ASL lacks a notion of vectors—they are represented as bitvectors with the element count/width specified on vector accesses—and ASLp relied heavily on the assumption that all loops would be fully unrolled after early analysis stages. We modified ASLp to retain loops from the original ASL specification through its various stages and introduced a late vectorization pass. This pass is capable of recognizing various idioms common to the specification such as element-wise vector operations and reductions, as well as performing induction variable substitutions. However, this is a best-effort approach, and ASLp defaults to unrolling if the vectorization fails. We implemented this new pass in about 1,000 lines of OCaml code.

We identified and corrected various other deficiencies by comparing the two lifters' outputs. For instance, ARM-TV appears to produce more reliable verification outcomes when instruction semantics are specified in terms of bitvectors with widths that are powers of two. We made minor changes to ASLp to ensure the majority of produced operations conform to this property.

2.6 Lifting Data

ARM-TV must lift data as well as code. Consider this function that takes a vector of two 32-bit integers, adds three to the first element and four to the second element, and returns the resulting vector:

```
define <2 x i32> @f(<2 x i32>) {
  %2 = add <2 x i32> %0, <i32 3, i32 4>
  ret <2 x i32> %2
}
```

LLVM's AArch64 backend lowers it to:

```
    .section    __TEXT,__literal8,8byte_literals
lCPI0_0:
    .long       3
    .long       4
    .section    __TEXT,__text,regular,pure_instructions
    .globl      _f
    .p2align    2
_f:
    adrp        x8, lCPI0_0@PAGE
    ldr         d1, [x8, lCPI0_0@PAGEOFF]
    add.2s      v0, v0, v1
    ret
```

Instead of materializing the literal constants three and four as part of the instruction stream, the backend has chosen to place them in real-only memory. Then, the generated code uses program-counter-relative addressing to access the constants, which are loaded into `d1`—the bottom 64 bits of the second vector register. To lift this assembly into LLVM IR that refines the original code, we must lift the constants as well as the instructions. ARM-TV's LLVM-level version of the constant data is:

```
%0 = type { i8, i8, i8, i8, i8, i8, i8, i8 }
@.LCPI0_0 = constant %0 { i8 3, i8 0, i8 0, i8 0, i8 4, i8 0, i8 0, i8 0 }, align 8
```

The first of these two lines declares an LLVM-level structure type containing eight 8-bit integers, that is isomorphic to a pair of adjacent `.long` directives in AArch64 assembly. The second line instantiates a read-only global variable of this type, containing the appropriate contents.

Despite being simple, this example shows that ARM-TV must be able to propagate symbolic address information for relocatable data items. It also needs to be able to handle:

(1) offsets from these data items,
(2) structures that contain a mixture of concrete and symbolic data, and
(3) data that contain cyclic symbolic references.

While none of these things is individually particularly complicated to support, we mention them to demonstrate the depth of implementation detail that is necessary to support lifting AArch64 code found in the wild.

Symbolic references and ASLp. The requirement to handle symbolic references stems from our design decision for ARM-TV to process "slightly symbolic" assembly code. In contrast, the ARM MRA, and also our ASLp-based lifter that is derived from it, have semantics for fully concrete machine code, where all symbolic references have been resolved by the assembler and linker: every instruction is no more and no less than 32 bits of data. To bridge this impedance mismatch, our ASLp-based lifter simply delegates symbolic instructions to the non-ASLp lifter. In most cases, their semantics are straightforward and easily tested, so this is an acceptable compromise.

2.7 Lifting Function Calls

From assembly, it is not possible to precisely determine the types of the arguments to a function call, nor is it even possible to precisely determine the number of arguments. Rather than using heuristics, we exploit LLVM-level information from the original function we were given. Then, lifting a function call becomes simply a matter of respecting the ABI, as discussed in Section 2.3, although now we are implementing the caller side, not the callee side.

Lifting indirect function calls is trickier: since the function being called is only an address that resides in a register, the name of the called function is unavailable, and thus we cannot look up the function's signature. To handle this case, we use LLVM's debug information to find the LLVM-level indirect call from which the ARM-level indirect call came from, and we retrieve its signature that way. Similarly, we use this mapping to determine the values of several call-site-specific function attributes that, when they were present in the original LLVM code, will trigger a failure of refinement if they are not also present in the lifted code.

As the first example in Section 1 showed, some function calls exist only on the LLVM side, and are expanded inline on the ARM side. It can also be the case that the LLVM backend generates a library call on the ARM side, where there is no corresponding function call on the LLVM side; memset() and memcpy() are the most common examples. As long as Alive2 has models for the behavior of the functions involved, neither of these conditions is problematic for ARM-TV.

One final awkward case that we have encountered is LLVM-level functions that require "immediate" arguments: these must be literal constants, and they have no representation at the ARM level. For example, one of LLVM's memcpy() intrinsics has this signature:

```
declare void @llvm.memcpy(ptr %dst, ptr %src, i64 %len, i1 %isVolatile)
```

The last argument, `%isVolatile`, is used by the LLVM backend but is not actually present on the ARM side. When lifting a memcpy() call, it is necessary to add this argument to the lifted code. There can be no principled mechanism for inferring the existence of this kind of argument; we simply recognize (by name) that such a function is being called and add the necessary argument(s).

2.8 Supporting an Assembly-Level Memory Model in Alive2

LLVM IR has high-level features not present in assembly. For example, LLVM uses a logical memory model, where pointers are represented as a pair: an object identifier and an offset into the object. A pointer can only be used to access the object that it explicitly references: even if the program learns or guesses its memory layout, it is not allowed to use a pointer derived from one object to access a different object [7]. These semantics simplify alias analysis and allow LLVM to exploit undefined behaviors in languages like C/C++ that expose a logical memory model to program developers. In contrast, a pointer in assembly is just an integer that can be used to index the entire address space.

Consider this C code:

```
void fn(int *p, int *q) {
  if (p == q)
    *p = 3;
  else
    *q = 3;
}
```

It can be (and, in fact, is by LLVM[8]) compiled into the following assembly, that has an unconditional store to the second argument:

```
fn:
        mov     w8, #3
        str     w8, [x1]   ; store though q, which is in x1 as specified by the ABI
        ret
```

This assembly code is correct: when the pointers are equal, we can store to either `p` or `q`. Since in the else branch we have to store to `q`, we can remove the branch altogether and do a single store to `q`. While this reasoning is valid in the assembly world, lifting it to LLVM IR *is not sound*. The main reason is that two pointers comparing equal does not imply they are equivalent since pointer equality only compares addresses and not provenance. Hence, pointer `q` may be out-of-bounds but have the same address as `p`, making it undefined behavior to dereference `q`. Thus, not only is the obvious lifting of this AArch64 into LLVM incorrect, but also there is not enough information, even in principle, at the assembly level to reconstruct something that respects LLVM-level memory rules.

Beyond provenance, there are other mismatches between the semantics of LLVM IR and assembly that make it impossible to represent all the relevant details of assembly code for verification in LLVM IR. Therefore, we decided to extend Alive2 by adding a new assembly mode. When a function is tagged as having been lifted from assembly, Alive2 changes its semantics as follows:

- The result of function calls is never undef or poison.
- The value of the function arguments is never undef or poison.
- Bytes loaded from memory are never undef or poison.
- Sub-byte accesses trigger undefined behavior (since ARM assembly does not support them).
- All pointers are treated as if they were physical (i.e., resulting from an integer-to-pointer cast) and have full provenance (i.e., they can access any valid memory region).

Furthermore, assembly mode changes Alive2's overall refinement criteria that are used to decide whether an assembly function correctly implements the original function in LLVM IR. Two changes were required:

- Byte refinement: allow type punning between integers and pointers to support cases where, for example, the IR stores a pointer to memory but the assembly program stores an integer.

[8] https://gcc.godbolt.org/z/snsbvvqMK

- Pointer refinement: only the addresses contained in the pointers are checked for equality. Alive2 does not consider whether the pointers are both logical or physical during the refinement check.

Prior to our work, Alive2 did not support physical pointers at all. Our implementation follows Lee et al.'s description [8], except that we do not support the `inbounds` attribute for the `getelementptr` pointer arithmetic instruction, since it never appears in lifted code.

We additionally changed how Alive2 treats sub-byte memory accesses. Alive2 originally implemented sub-byte stores by extending them to a byte boundary, padding out the value with poison bits. Similarly, sub-byte load operations would mask out the padding bits. This was mostly sufficient to emulate the semantics of LLVM IR where the loaded value is poison if it was stored with a different size (e.g., storing a 5-bit integer and then loading a 4-bit integer yields a poison value). This encoding, however, did not capture the fact that the position of the padding bits should be non-deterministic since different ABIs may implement padding differently. It also did not allow us to capture the intricacies of ARM's ABI.

We changed the encoding of a byte in Alive2 to record two additional pieces of information: (1) the number of stored bits, and (2) the byte number (applicable to stores of integers larger than eight bits). It then becomes obvious when a load should be poison. For efficiency, these fields are only added to the SMT encoding for programs that do any sub-byte accesses.

The undocumented, LLVM-specific AArch64 ABI assumes that every sub-byte store is zero-extended to the next byte boundary. Since LLVM takes advantage of this ABI contract when loading sub-byte data (it does not mask out the padding bits, since they are assumed to contain zeroes), we need to verify the store side of this contract. We extended Alive2's memory refinement criteria to check that a sub-byte store in the original LLVM IR is refined by a byte on the assembly side if and only if the assembly byte contains zeros in its padding bits. Moreover, we added an extra axiom to ensure that both the initial memory and the memory after a function returns respect this part of the ABI. Together, these changes enable modular verification of refinement; that is, verifying each function separately implies correctness of compilation of the whole program.

Unfortunately, the changes described in this section made Alive2 very slow: it was sometimes incapable of validating functions containing two memory operations. The next section describes subsequent optimization work we did to recover verification performance.

2.9 Optimizing the Assembly Memory Model

SMT solvers are not magic: unless problems posed to them are encoded carefully and efficiently, they use an unacceptable amount of CPU time or memory. In particular, the memory model is a bottleneck for Alive2, and more so when dealing with physical pointers (i.e., all pointers for code in assembly language). The core problem is that Alive2's previously existing memory optimizations were based on LLVM's logical (object+offset) model. For example, if Alive2 knows that a pointer is somewhere within a 30-byte memory object, it can be represented using just five bits. Physical pointers, by their nature, require the full 64-bit width. More importantly, physical pointers have full provenance. That is, they can potentially point anywhere in memory, whereas logical pointers can usually only access a small number of blocks. This makes alias analysis much more difficult for the assembly memory model.

This section describes two new memory optimizations that we implemented within Alive2, that reduce the size of the encoding of memory operations by an order of magnitude in many cases. Together, they make the encoding of physical pointers almost as efficient as the encoding of logical pointers. We have pushed these optimizations to upstream Alive2.

Synchronizing alignment. AArch64, like x86-64, does not require most memory operations to be aligned. For example, it is legal to load a 64-bit value from memory address 0x3. Even so, LLVM IR usually comes with ABI-mandated alignment requirements on memory operations. Hence, at the LLVM level, 64-bit loads are usually required to be given pointers that are eight-byte aligned.

Prior to our work, Alive2 had an optimization that increased the size of memory words in the SMT encoding when this was justified by the alignments and access sizes in the code being analyzed. For example, if a function contains no memory accesses smaller than 32 bits, and all loads and stores are aligned on four-byte boundaries, then instead of representing memory as an array from offsets to bytes, it would represent memory as an array from offset/4 to 32-bit words. This cuts the size of the encoding by a factor of four, since a four-byte load is now a single SMT expression instead of a concatenation of four one-byte loads. It also makes offsets smaller by two bits.

Alas, ARM-TV defeated this important optimization because it lifts all ARM-level memory operations to LLVM instructions with one-byte alignment.[9] To recover this optimization, we implemented a preprocessor that attempts to align the control flow of an LLVM function and its lifted twin, in order to propagate the alignment of the original memory operations to the lifted ones. This is sound: if addresses were not aligned, the original function would trigger UB (undefined behavior). Hence, the lifted code can assume that any address accessed in the original IR is properly aligned.

To align the functions, we use a simple algorithm that first marks the entry basic blocks (BBs) of the two functions as twins and follows the control flow in tandem. If a source BB conditionally jumps to BB1 or BB2, and a twin assembly BB jumps to BB1' or BB3', we mark BB1/BB1' and BB2/BB3' as twins. This step is repeated until no more BBs can be matched. The names of the BBs are not meaningful: we match only the control flow graph (CFG) structure. In each BB and its twin, we check if they have the same memory operations. If so, we copy the alignment from the original BB, otherwise we give up on that BB and all successors.

This relatively simple, dataflow-insensitive algorithm is usually sufficient, but we have found some cases where the compiler backend changes the order of load operations, leading to false alarms. However, it is always possible to disable the alignment inference heuristic and check if the bug is still reported.

Inbounds alignment. The second significant inefficiency in Alive2's memory encoding came from pointers with full provenance—which is all pointers in code lifted from assembly. The encoding gets substantially larger than with logical pointers because 1) pointers with full provenance can alias any memory, and 2) the offset of a physical pointer within a memory block must be computed using a full 64-bit subtraction between the pointer and the address of the beginning of the block. These features, together, are necessary to account for the worst-case scenario where a full-provenance pointer is out of bounds and can point into another block. However, this is rare, and only compiler optimizations that propagate pointer equalities need to deal with this case.

We improved Alive2's encoding by first propagating UB conditions from the original to the lifted function. When we start doing symbolic execution of a new basic block, we check if the original function has a BB with the same path from the function entry and the same number of function calls for each function (to account for possibly non-returning functions). If so, we copy all the UB triggered in that BB up to the first possibly non-returning function. This is sound because LLVM uses time-traveling UB semantics, in which UB executed in the future can be brought earlier. We use function calls as barriers, since if they do not return we would not execute UB that occurs after them.

[9]We cannot simply assume that, for example, a 4-byte load in ARM assembly is aligned on a 4-byte boundary. LLVM's AArch64 backend freely inserts misaligned memory operations when this is convenient. The memcmp() example in Section 1 illustrates this: the `ldr` instructions that perform 8-byte loads will be misaligned whenever the function arguments are.

Consider this example:

```
void fn(int *p, int *q, int *r) {
  *p = 3;
  *q = 4;
  g();
  *r = 5;
}
```

Here, we would add as assumptions that p and q must be inbounds of a valid memory block (if they are not, then the original LLVM IR is undefined, and anything can refine it). We cannot assume that r is inbounds because it may not be dereferenced in the original function if g does not terminate.

When generating the SMT encoding for memory operations in the lifted code, we issue an SMT call to check if the assumptions imply that the pointer is inbounds. In our experiments, these SMT calls returned quickly because the constraints are simple bit-vector inequalities over offsets and memory block sizes plus no-overflow constraints.

2.10 Limitations of ARM-TV

ARM-TV focuses on sequential user-mode code. It does not support processor mode changes, privileged instructions, inline assembly, thread-local storage, C-style variadic functions, float types other than IEEE 754 single and double precision, non-default rounding modes for floating point instructions, or structures in function arguments and return values. We have also inherited several limitations from Alive2, such as not supporting unbounded loops, exception handling, or type-based aliasing information.

3 Evaluation

We return to the research questions posed in Section 1, answering them based on a collection of experiments and on our experience using ARM-TV in anger, to find bugs in upstream LLVM.

3.1 Experimental Setup

For the experiments in Sections 3.2–3.4, we gathered two collections of functions in LLVM IR. First, we took all of the functions in the LLVM unit test suite and filtered out the ones that are unsupported by upstream Alive2 and also those that cannot be lowered to AArch64 (because, for example, they contain inline assembly for a non-ARM architecture). This left us with 232,981 functions. We chose these functions for our evaluation because the LLVM community has implicitly decided (by including these functions in the test suite) that it is important to compile them correctly. The second collection of functions that we gathered came from compiling the C and C++ programs in SPEC CPU 2017 to LLVM IR on an ARM-based MacBook Pro, using the recommended baseline optimization flags.[10] This produced an additional 87,738 functions. For this collection of functions, we did not filter out those that are unsupported by upstream Alive2, but rather simply used every function that could be turned into ARM instructions. We chose the SPEC benchmarks because they have been selected, independently of our work, to be a representative collection of software.

We ran our experiments on an AMD EPYC 7502 machine (Zen 2 microarchitecture) with 64 hardware threads and 256 GB of RAM, running Ubuntu Linux version 24.04. We limited each invocation of ARM-TV to 10 minutes of CPU time and 4 GB of virtual memory. ARM-TV's run time is dominated by Z3's satisfiability checks—the rest of its tasks are relatively fast.

[10] `clang -O3 -march=native -fno-strict-aliasing -fno-unsafe-math-optimizations`

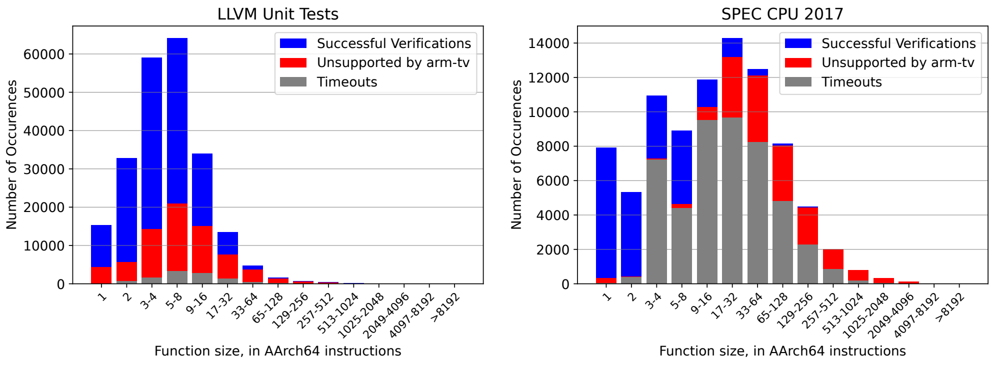


Fig. 3. Translation validation outcomes for functions from the LLVM unit test suite and from SPEC CPU 2017, binned according to the number of ARM instructions in each function

3.2 RQ1: Can We Create a Practical Translation Validation Tool for LLVM's AArch64 Backend, Without Modifying LLVM?

One sense in which ARM-TV is practical is that it is easy to use: one runs it on an LLVM module (a container for functions, variables, and data in LLVM IR, that is analogous to a translation unit in C or C++), telling it which function's translation should be validated. ARM-TV loads the module, lowers the selected function to AArch64 assembly, validates the translation, and reports its results. This is all completely automated. In the common case, the translation validates as correct and there is nothing more to do. When ARM-TV signals a failure of refinement, it provides a counterexample—a collection of memory states and function arguments—that causes the miscompilation to manifest itself during a concrete execution.

Fig. 3 shows the outcomes of running ARM-TV on the LLVM unit tests and on the functions from SPEC CPU 2017, binning the results by function size as measured by the number of ARM instructions in each function. These results are all from the ASLp version of ARM-TV; the results from the non-ASLp version of ARM-TV are very similar. We omit them here but explicitly compare the two versions of ARM-TV in Section 3.4.

These graphs show that, as expected, functions from SPEC tend to be larger than those from the LLVM unit tests. Moreover, code from SPEC is more difficult to verify: it generates many more timeouts than does the LLVM unit test suite, even for functions of comparable size. This is because SPEC code uses pointers heavily and commonly uses features such as function pointers and multiple levels of indirection that are difficult for Alive2 and Z3 to handle.

For the LLVM unit tests, the most common unsupported features blocking ARM-TV were:

- vector parameter type with unstable ABI (55% of unsupported functions)
- float type other than single or double (14%)
- function argument type other than integer, float, pointer, or vector (4%)
- variadic function argument (4%)
- structure in return value (3%)

In contrast, for SPEC, the most commonly seen unsupported features blocking ARM-TV were:

- LLVM instruction unsupported by Alive2, usually related to exception handling (47% of unsupported functions)
- variadic function argument (16%)
- LLVM attribute unsupported by Alive2, usually `noalias` (10%)

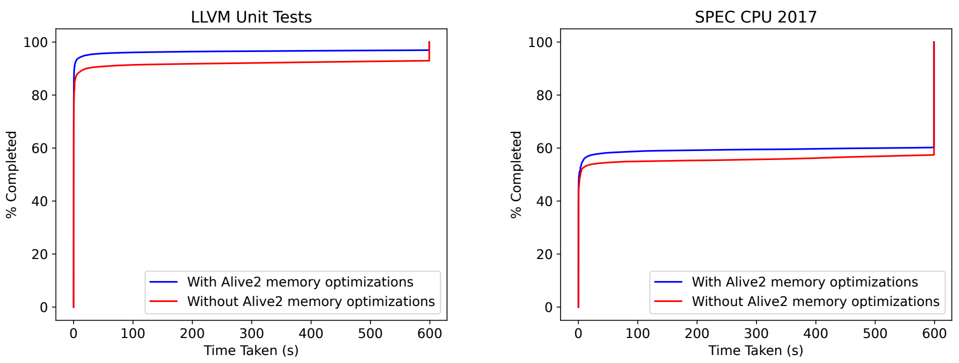


Fig. 4. CDFs of completion times for ARM-TV invocations before and after our Alive2 memory subsystem optimizations. Each invocation timed out after 10 minutes (600 seconds).

- personality function not supported by Alive2 (9%)
- aliasing-related LLVM metadata unsupported by Alive2 (7%)

3.3 RQ2: Can We Optimize Alive2 for Assembly-Style Code that Freely Intermixes Integers and Pointers, that Defeat Many of Its Existing Memory Verification Optimizations?

Fig. 4 shows CDFs for ARM-TV completion times before and after we implemented the assembly-focused memory model optimizations from Section 2.9. For the LLVM unit test suite, these were highly effective, reducing the percentage of translation validation calls that time out from 7.0% to 3.1%. However, for functions from SPEC CPU 2017, our optimizations had less of an effect: timeouts were reduced from 43% to 40%. It appears that the significantly larger, more pointer-intensive functions from SPEC are simply less amenable to our optimizations than are the relatively simple functions from the LLVM unit test suite.

3.4 RQ3: What Are the Tradeoffs Between Writing Semantics for AArch64 Instructions by Hand and Deriving Semantics from ARM's Machine-Readable Architecture Description?

Our hand-written code for lifting individual AArch64 instructions is 8.1 kloc, out of ARM-TV's total size of 14.4 kloc. We support about 1,400 instructions (using LLVM's internal decomposition of this instruction set) but there is considerable overlap between these, that we have exploited to make our implementation as concise as possible. In contrast, our ASLp-based lifter required about 2 kloc of C++, and again this code was not particularly complicated or difficult. In fact, since this code was able to take advantage of subroutines that we had created to support our hand-written lifter, it was generally quite pleasant to write.

Correctness. 8,100 lines of fiddly, bit-twiddling C++ are bound to contain bugs and, in fact, we discovered numerous bugs in our hand-written lifter by testing it. All of these have been fixed, but focused testing of our semantics was definitely necessary. In contrast, our ASLp-derived lifter was (empirically) almost completely correct by construction. We did have a very small number of bugs in cases where an ARM instruction maps to a complex LLVM operation, such as a floating point rounding operation—these mappings had to be done by hand, creating opportunities for error.

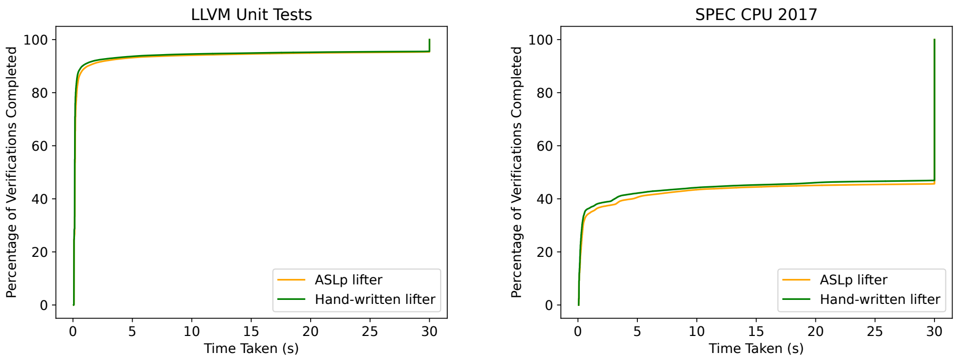


Fig. 5. CDFs of completion times for ARM-TV invocations using the hand-written and ASLp-derived instruction lifters. Only one line is apparent in the left-hand graph because the lines almost totally overlap.

Completeness. Although our hand-written lifter is not missing support for any instruction generated by LLVM's AArch64 backend when compiling SPEC CPU 2017, running ARM-TV on LLVM's unit test suite reveals that it is missing 641 instructions. These are predominantly vector and floating point operations. Counting the same way, our ASLp-based lifter is missing only 11 instructions, all relating to thread synchronization, pointer authentication, and processor feature reporting. All of these things are purposefully out of scope for ARM-TV.

We looked at how our ASLp-derived lifter works for the LLVM unit tests that cannot be processed using ARM-TV's hand-written lifter, due to an unsupported instruction. 54% of the time, the original LLVM IR contains an intrinsic not yet supported by Alive2, such as `@llvm.aarch64.cls64` or `@llvm.aarch64.neon.urshl.v2i32`. 17% of the time, the code generated by the ASLp lifter times out, for example due to a floating point operation or a complicated vector operation. Finally, in just four cases containing an ARM instruction missing from our hand-written lifter, code produced by the ASLp-based lifter can be validated as a correct translation of the original LLVM IR.

Verification performance. One reason we implemented two different lifters is that we wanted to enable a comparison between them in terms of verification performance. The specific question we wanted to answer is: Does generating idiomatic LLVM IR in a hand-written lifter result in better verification performance than we would get by mechanically following the ARM MRA?

Indeed, we initially noticed that code lifted using ASLp resulted in significantly more timeouts, compared to code produced by our hand-written lifter. However, after implementing the optimizations described in Section 2.5, much of the performance gap between the two lifters disappeared. Fig. 5 illustrates our current results where the number of timeouts for LLVM unit tests is virtually unchanged by the choice of lifter. For functions from SPEC CPU 2017, the ASLp lifter causes an additional 2% of ARM-TV invocations to time out.

3.5 RQ4: Does LLVM's AArch64 Backend Have Latent Miscompilation Bugs that We Can Discover Using Translation Validation?

We did not discover any miscompilations by running ARM-TV on code from SPEC CPU 2017. We did, however, find a small number of LLVM bugs using functions that are already part of the LLVM unit test suite. This is possible because ARM-TV is a very strong correctness oracle, compared to the syntax- and crash-based oracles that are used in the test suite. Mostly, however, we found bugs by using ARM-TV together with fuzzers. Our bug-finding effort was conducted during April 2022–July

2025 and was not a controlled experiment. Rather, we opportunistically found and reported bugs as we had time, and when we wanted a top-level sanity check that ARM-TV was working as intended.

To find bugs, we primarily used existing randomized compiler testing tools: YARPgen [12], which generates random C code, and also alive-mutate [2] and IRFuzzer [18]; both mutation-based fuzzers for LLVM IR. For the mutation-based tools, our seed corpus was the functions in the LLVM unit test suite. We also wrote a few simple test-case generators to ensure that, for example, memory and arithmetic operations of every bitwidth in 1..64 were correctly supported by the AArch64 backend. These custom tools were invaluable for gaining confidence in ARM-TV but did not lead to the discovery of new LLVM bugs.

Table 1 summarizes the results of our testing campaign; it only includes miscompilation errors, we ignored compiler crashes while doing this work. The "categories" column contains our (somewhat subjective) judgments about which aspects of the semantics of LLVM IR one would need to understand, in order to understand the bug. Three of the categories are abbreviations: FP (floating point), UB (undefined behavior), and ABI (application binary interface). Most of the bugs that we found were fixed by changing source files that are shared across multiple LLVM backends. Hence, many of these bugs could have affected multiple backends (in some cases, the developers' discussion at the Github issue explicitly indicates that this was the case). Out of these 45 bugs, 40 have been fixed. Many of the bugs (including all of the unfixed ones) involve undefined behavior—a historically problematic area for the LLVM optimizer [9]. During the course of this work we also found and reported several miscompilation bugs in LLVM's middle-end optimizer; we did not include these in our table.

Some of the bugs that we found were straightforward. For example, bug #55284 led the "global instruction selection" mode of LLVM's AArch64 backend[11] to miscompile this function:

```
define i32 @f3(i32 %0) {
  %2 = or i32 %0, 65536
  %3 = and i32 1520220788, %2
  ret i32 %3
}
```

as:

```
_f3:
    orr    w8, w0, #0x10000
    and    w0, w0, w8
    ret
```

This ARM code, for example, returns 1 instead of 0 when passed 1 as an argument. In contrast, a correct compilation of this function is:

```
_f3:
    mov    w8, #47732
    movk   w8, #23196, lsl #16
    and    w0, w0, w8
    ret
```

Other bugs were more involved. Consider this function:

```
define i32 @f(i1 signext %0, i32 %1) {
  %3 = zext i1 %0 to i32
  %4 = sub i32 %1, %3
  ret i32 %4
}
```

The first function parameter, `%0`, has type `i1`: a one-bit integer. This corresponds to a Boolean parameter in C++. The `signext` attribute attached to `%0` has no semantics at the level of LLVM

[11]Enabled by default in Apple's clang at some optimization levels.

Table 1. Miscompilation bugs that we found using ARM-TV and reported to LLVM developers. The Github IDs are links into the LLVM project's issue tracker. Bugs marked with an asterisk (*) are discussed in Section 1.3 or 3.5.

Github ID	Affected Operation(s)	Categories	Fixed?
55003	funnel shift	Integer, UB	Yes
55129	right shift	Integer	Yes
55150	integer/FP conversions	Integer, FP	Yes
55178	left shift + comparison	Integer, UB	Yes
55201	rotate-like code	Integer	Yes
55271	unsigned maximum + remainder	Integer, UB	No
55284*	or + and	Integer	Yes
55287	urem + udiv	Integer	Yes
55296	funnel shift	Integer	Yes
55342	arithmetic on constants	Integer	Yes
55484	arithmetic	Integer	Yes
55490	arithmetic on constants	Integer	Yes
55627	arithmetic on constants	Integer	Yes
55644	multiplication with overflow check	Integer	Yes
56664	funnel shift	Integer	Yes
57181*	sign extension	Integer, ABI	Yes
57256	funnel shift	Integer, UB	No
58109	saturating subtraction	Integer	Yes
58321	arithmetic	Integer, UB	No
59898	funnel shift	Integer	Yes
59902	signed maximum	Integer	Yes
72475	"select" instruction	Integer, UB	Yes
74248*	vector insertion	Integer, Vector, Memory, UB	Yes
75557	vector insertion	Integer, Vector, Memory, UB	Yes
76769	vector reduction	Integer, Vector	Yes
77169	vector signed remainder	Integer, Vector	Yes
78383	vector extraction	Integer, Vector, Memory, UB	Yes
78477	vector inspection	Integer, Vector, Memory	Yes
84718	arithmetic	Integer	Yes
88959	vector extraction	Integer, Vector, Memory, UB	No
88784	vector absolute difference	Integer, Vector	Yes
88950	saturating left shift	Integer	Yes
90242	array writes	Integer, Memory	Yes
90245	subtraction	Integer, UB	No
90532	FP compare	FP	Yes
90936	store coalescing	Integer, Memory	Yes
96366	arithmetic	Integer, UB	Yes
121372	arithmetic	Integer, Vector	Yes
125989	vector shuffle	Integer, Vector	Yes
133928	arithmetic	Integer	Yes
136746	funnel shift	Integer	Yes
137254	arithmetic	Integer, UB	Yes
137998	FP compare	FP	Yes
145360	sdiv + srem	Integer, UB	Yes
145363	loads with range metadata	Integer, Memory, UB	No

IR, but during code generation it permits the compiler to assume that this function's caller has sign-extended the first function argument up to 32 bits. The problem is that this LLVM-level requirement interacts with ARM's ABI document, which specifies that Boolean function arguments are zero-extended to eight bits. Lacking documented guidance about the relative precedence of these rules:

- The default "selection DAG" mode of LLVM's AArch64 backend assumed that `%0` was sign extended all the way from one bit to 32 bits.
- The global instruction selection mode of the AArch64 backend—and also ARM-TV—assumed that `%0` was zero extended to eight bits before being sign extended to 32 bits.

We detected the issue because ARM-TV signaled a miscompilation of the function above when it was translated by the selection DAG backend. After some discussion among LLVM stakeholders, it was agreed that sign extension from one bit to 32 bits was the desired behavior, and the global instruction selection backend's behavior, as well as ARM-TV's, were changed (#57181).

At present, our fuzzing campaign has reached a stable state where it is no longer finding very many old bugs in LLVM's AArch64 backend. This indicates that—within the limits of the available tools and resources—we are starting to mine out the bugs in this code base. This is the desired endgame for a randomized testing campaign!

4 Discussion and Future Work

Is ARM-TV trustworthy? We wrote more than 7,500 test cases for ARM-TV, and we also look carefully at the alarms that it produces when we run it on fuzzer-generated test cases. Even so, we worry about missed alarms regarding cases where we neglected to check, or only incompletely check, some ABI property of interest. We do not even have a list of these properties that we are certain is complete. What we want is an ABI specification that is tight in the sense that we can prove that a callee will function correctly if its callers meet their obligations, and vice versa. We would love to see more work along the lines of Wagner et al. [25].

Can we improve scalability? As Figure 3 shows, ARM-TV has trouble validating the translation of medium to large functions. There is little that we can do about this at the level of ARM-TV—improvements will have to come from Alive2 or Z3. Although Alive2 has already been subjected to extensive optimization effort, many opportunities remain. For example, better program alignment, including speculative alignment, would allow us to break functions into many small pieces, making vcgen and obligation discharge much faster. Also, Alive2's memory model is still far from optimal. Alas, there is no single optimal encoding for all programs, but rather this depends on details such as the ratio of loads to stores in a particular program. Being able to dynamically switch between encodings would (we believe) improve performance significantly.

Is lifting the right strategy? We don't know. Implementing ARM-TV as a lifter was convenient: by reusing Alive2's LLVM semantics and refinement checking machinery, we were able to be up and running rapidly. Moreover, lifting to LLVM IR allowed us to reuse LLVM's powerful middle-end optimizer. On the other hand, the issues we discussed in Sections 2.2 and 2.8 were awkward and created some implementation complexity for us. An alternative to lifting would be lowering both LLVM IR and AArch64 code into a common semantic framework and establishing refinement conditions there. This would avoid the impedance mismatches that we encountered, at the cost of writing a significant amount of new code.

If we wanted to formally verify ARM-TV, then our current implementation strategy would be a hopelessly bad choice. In that case, we would have built it inside of a proof assistant.

Can ARM-TV support other LLVM backends? We are in the process of adding support for LLVM's 64-bit RISC-V backend. So far, we support RV64I—the core integer ISA—in addition to its standard B (bit manipulation) and M (multiplication and division) extensions. RISC-V is extremely simple compared to AArch64, although we have not yet started on its floating point or vector extensions. The modifications to Alive2 that we made to support AArch64 are sufficient to handle scalar RISC-V code, and we believe they would also be sufficient for other general-purpose ISAs, such as x86-64. On the other hand, Alive2 does not yet support the scalable vectors that RISC-V's vector ISA uses, nor does it support multiple address spaces—so we cannot yet support either GPUs or Harvard architecture microcontrollers such as AVR.

Can we support inline assembly? Alive2 already has rudimentary support for inline assembly, but we have not yet attempted to make use of it in ARM-TV. The inline instructions themselves can easily be assigned semantics using our lifter, but we would still need to formalize the interface between LLVM IR and assembly. Recoules et al. [16] describe an approach that seems like it would work for us, as well.

Can we support concurrent programs? We do not envision extending ARM-TV to support reasoning about multiple threads at once. On the other hand, it should be possible to validate the modular rules for concurrency-related transformations. For example, if an LLVM backend moved a load or store around a synchronization instruction in an illegal fashion, we could signal an error.

5 Related Work

Since translation validation [13–15, 19, 21–23] requires minimal, if any, support from the compiler being validated, it is an attractive technology for working with existing production compilers like LLVM and GCC that are not obviously amenable to direct application of formal methods. In this section we look at prior research that relates to, and in some cases inspired, our current work.

Numerous lifters from assembly languages to LLVM IR have been developed, primarily by the security community; Liu et al. [11] describe and evaluate several of them. Here we focus on previous work that performed lifting for purposes of verification. Lam and Coughlin created ASLp, which generates a lifter for AArch64 instructions from ARM's MRA [5]. We adapted and improved ASLp so it could be used from ARM-TV. Verbeek et al. [24] have developed a lifter for x86-64 code that provably respects not only the overt computation that it performs, but also covert behaviors that would be of interest to security researchers.

In terms of finding bugs using translation validation, Ruffy et al. [20] found 34 miscompilation defects in P4C, a compiler for the P4 packet processing DSL, using a translation validation tool that they developed. Dagsupta et al. [1] used an existing formal semantics for x86-64 to derive their own reference lifter, and then used translation validation to find numerous bugs in McSema, a security-oriented binary lifter for LLVM IR. Kwon et al. [4] used translation validation to find a previously unknown bug in TurboFan (V8's JIT compiler for JavaScript), and also showed that they could reproduce a number of known bugs. Optimuzz [3] couples targeted fuzzing—where mutations are selected for their ability to execute a specific part of the compiler—with translation validation, discovering a number of bugs in LLVM and TurboFan.

6 Conclusion

Our original goal—a pushbutton tool that people developing safety-critical, embedded AArch64 systems can use to ensure that an LLVM upgrade did not introduce a backend-related miscompilation into their software—is not yet fully realized. We still need to be able to reliably validate the translation of large functions that are created by heavy inlining, and also to support concurrency and system-level features such as volatile memory accesses, inline assembly, and interrupt handlers.

Nevertheless, ARM-TV is useful and advances the state of the art. It checks not only functional properties of code—such as its return value and memory effects—but also ABI-level properties, such as clobbering a callee-saved register. We have implemented novel verification optimizations: a vectorizer for ASLp that allows it to produce idiomatic LLVM vector operations instead of the open-coded equivalent versions that would be produced by a straightforward reading of ARM's ASL (Architecture Specification Language). Also, we implemented several new memory subsystem optimizations in Alive2 that substantially improve its performance when dealing with physical pointers that are found in code lifted from assembly. We performed the first (as far as we're aware) apples-to-apples comparison of a hand-written instruction set semantics against a semantics mechanically derived from ASL, finding that after significant optimization work, the mechanically derived semantics perform almost as well as the hand-written ones. Finally, we used ARM-TV to find 45 previously unknown miscompilation bugs in LLVM, which we reported, and 40 of which have been fixed. ARM-TV is open source software and runs on Linux and macOS. It can be found at: https://github.com/regehr/alive2/tree/arm-tv

Acknowledgments

This work was supported in part by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), and unrestricted gifts from Google and from Woven by Toyota, Inc.

References

[1] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable validation of binary lifters. In *PLDI*. doi:10.1145/3385412.3385964

[2] Yuyou Fan and John Regehr. 2024. High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM. In *CGO*. doi:10.1109/CGO57630.2024.10444854

[3] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. *Proc. ACM Program. Lang.* 9, PLDI, Article 172 (June 2025). doi:10.1145/3729275

[4] Seungwan Kwon, Jaeseong Kwon, Wooseok Kang, Juneyoung Lee, and Kihong Heo. 2024. Translation Validation for JIT Compiler in the V8 JavaScript Engine. In *ICSE*. doi:10.1145/3597503.3639189

[5] Kait Lam and Nicholas Coughlin. 2023. Lift-off: Trustworthy ARMv8 semantics from formal specifications. In *FMCAD*. doi:10.34727/2023/isbn.978-3-85448-060-0_36

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. doi:10.1145/2594291.2594334

[7] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (Oct. 2018). doi:10.1145/3276495

[8] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *CAV*. doi:10.1007/978-3-030-81688-9_35

[9] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. doi:10.1145/3062341.3062343

[10] Arm Limited. 2025. Arm A64 Instruction Set for A-profile architecture. https://www.arm.com/architecture/cpu/a-profile

[11] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In *IEEE SP*. doi:10.1109/SP46214.2022.9833799

[12] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020). doi:10.1145/3428264

[13] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. doi:10.1145/3453483.3454030

[14] George C. Necula. 2000. Translation validation for an optimizing compiler. In *PLDI*. doi:10.1145/349299.349314

[15] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS*. doi:10.1007/BFb0054170

[16] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2020. Get rid of inline assembly through verification-oriented lifting. In *ASE*. doi:10.1109/ASE.2019.00060

[17] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *FMCAD*. doi:10.1109/FMCAD.2016.7886675

[18] Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. 2024. IRFuzzer: Specialized Fuzzing for LLVM Backend Code Generation. arXiv:2402.05256

[19] Abhishek Rose and Sorav Bansal. 2024. Modeling Dynamic (De)Allocations of Local Memory for Translation Validation. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 146 (April 2024). doi:10.1145/3649863

[20] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: finding bugs in compilers for programmable packet processing. In *OSDI*. https://www.usenix.org/conference/osdi20/presentation/ruffy

[21] Hanan Samet. 1975. *Automatically proving the correctness of translations involving optimized code.* Ph. D. Dissertation. Stanford University.

[22] Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *PLDI*. doi:10.1145/2491956.2462183

[23] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *PLDI*. doi:10.1145/1993498.1993533

[24] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally verified lifting of C-compiled x86-64 binaries. In *PLDI*. doi:10.1145/3519939.3523702

[25] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024). doi:10.1145/3689755

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. doi:10.1145/1993498.1993532

[27] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *PLDI*. doi:10.1145/2491956.2462164

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