High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM

Yuyou Fan
University of Utah
USA
yuyou.fan@utah.edu

John Regehr
University of Utah
USA
regehr@cs.utah.edu

***Abstract*—It is very difficult to thoroughly test a compiler, and as a consequence it is common for released versions of production compilers to contain bugs that cause them to crash and to emit incorrect object code. We created alive-mutate, a mutation-based fuzzing tool that takes test cases written by humans and randomly modifies them, based on the hypothesis that while compiler developers are fundamentally good at writing tests, they also tend to miss corner cases. Alive-mutate is integrated with the Alive2 translation validation tool for LLVM, which is useful because it checks the behavior of optimizations for all possible values of input variables. Alive-mutate is also integrated with the LLVM middle-end, allowing it to perform mutations, optimizations, and formal verification of the optimizations all within a single program—avoiding numerous sources of overhead. Alive-mutate's fuzzing throughput is 12x higher, on average, than a fuzzing workflow that runs mutation, optimization, and formal verification in separate processes. So far we have used alive-mutate to find and report 33 previously unknown bugs in LLVM.**

I Introduction

In principle, compilers are highly amenable to formal verification: they translate a semantically unambiguous source language into a semantically unambiguous target language. In practice, production-grade compilers are large, complex, and are often implemented in unsafe, imperative programming languages. Full formal verification of multi-million line artifacts like GCC and LLVM remains beyond the state of the art. An alternative verification technology, translation validation, can be used to prove that a single execution of a compiler functioned correctly. Translation validation can be applied to existing production compilers because the verification tool only has to reason about the original and compiled code: the compiler implementation can be entirely ignored. However, when translation validation is used as part of a compiler bug-finding campaign, it must be coupled with a source of inputs such as a collection of application code or a fuzzer. In this paper, we present alive-mutate: a new mutation-based fuzzer for code in the LLVM intermediate representation (IR) that, coupled with the Alive2 translation validation tool [9], is effective at finding optimization bugs; it has helped us find 33 previously unknown bugs so far.

Our new mutation engine is based on two basic ideas. First, we have observed that it is a fairly common occurrence for an existing test case to come close to triggering a bug, but to miss the mark somehow. In other words, while compiler developers are very good at writing tests that stress-test different facets of an optimization, they sometimes miss corner cases. Therefore, our strategy is to randomly modify previously-written unit tests in order to explore a neighborhood of the program space near each test, in order to cast a wider net and hopefully trigger subtle errors. Unlike a metamorphic testing technique such as EMI [7], which is required to preserve some or all of the original code's semantics, alive-mutate does not need to be semantics-preserving. Rather, it is the LLVM optimization passes that must have this property.

Our second motivating idea is that high throughput is important for a mutation-based fuzzing campaign. This is because, while some bugs reveal themselves easily, other bugs are extremely difficult to trigger during testing, and these bugs tend to be the ones that remain in a software system once the easy bugs have been discovered and fixed. A high-throughput testing tool can find more of these bugs than a slow tool can. Figure 2 shows a traditional fuzzing workflow with associated overheads including context switching and file I/O. To this end, alive-mutate runs in the same process as the LLVM optimizers and also Alive2, allowing the mutate-optimize-verify loop to amortize away almost all sources of overhead such as parsing, printing, file I/O, process creation and destruction, and context switches. Alive-mutate's testing throughput is about 12 times as fast, on average, when compared to a workflow that runs mutation, optimization, and verification in separate UNIX processes.

To see how alive-mutate can reveal a compiler bug, consider Figure 1, which shows three versions of a function from LLVM's unit test suite. The first, Listing 1, is the original code; of course it is not

Listing 1. One of LLVM's unit tests

```
define i32 @t1_ult_slt_0(i32 %x,
                         i32 %low,
                         i32 %high) {
    %t0 = icmp slt i32 %x, -16
    %t1 = select i1 %t0, i32 %low,
                         i32 %high
    %t2 = add i32 %x, 16
    %t3 = icmp ult i32 %t2, 144
    %r = select i1 %t3, i32 %x, i32 %t1
    ret i32 %r
}
```

Listing 2. The test, after mutation by alive-mutate. Changes are shown in bold.

```
define i32 @t1_ult_slt_0(i32 %x,
                         i32 %low,
                         i32 %high) {
    %t0 = icmp slt i32 %x, 0
    %t1 = select i1 %t0, i32 %low,
                         i32 %high
    %t2 = icmp ult i32 %x, 65536
    %1 = xor i1 %t2, true
    %r = select i1 %1, i32 %x, i32 %t1
    ret i32 %r
}
```

Listing 3. The mutated function, after being optimized by the then-current version of LLVM's InstCombine pass in January 2022. This optimization was unsound. We reported this bug and it was fixed by the LLVM developers.

```
define i32 @t1_ult_slt_0(i32 %x,
                            i32 %low,
                            i32 %high) {
    %1 = icmp slt i32 %x, 0
    %2 = icmp sgt i32 %x, 65535
    %3 = select i1 %1, i32 %low, i32 %x
    %4 = select i1 %2, i32 %high, i32 %3
    ret i32 %4
}
```

Fig. 1. An example of a bug discovered with the help of alive-mutate

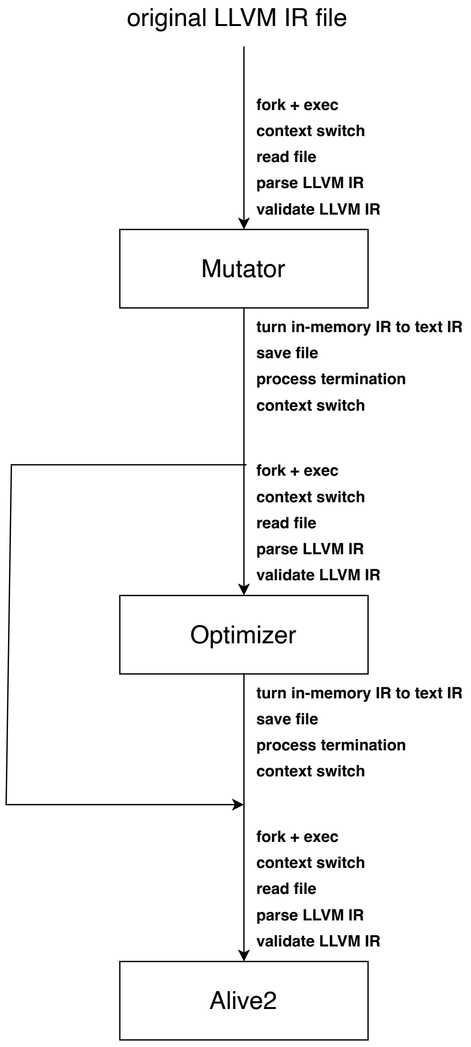

Fig. 2. A mutate-optimize-verify workflow based on discrete programs encounters numerous sources of overhead, shown in bold. alive-mutate achieves high throughput by removing these overheads from its critical path.

miscompiled by LLVM, or else this would have been flagged as a regression during normal testing. The second, Listing 2, has been mutated by alive-mutate; a literal constant has been changed, instruction `%t2` has been moved after `%t3` (updating the definition used in `%t3`), and the `and` instruction has been changed to an `xor`. The third function, Listing 3, was produced by optimizing the mutated function using LLVM's InstCombine optimization pass. InstCombine performs a wide variety of peephole-style optimizations. This optimization is incorrect: given the inputs `x=2`, `low=1`, and `high=1`, the mutated function returns 1 while the optimized function returns 2. We reported this bug to the LLVM developers and it was rapidly fixed.

II Rationale for a New Fuzzing Tool

Given the availability of powerful, coverage-guided, mutation-based fuzzers, one might ask why we decided to implement a new mutator. The short answer is that off-the-shelf tools are structure-agnostic: they mutate file formats that they do not understand, relying on heuristics that have worked well in the past. This works remarkably well in many cases, but we found that it did not work well for LLVM IR.

Going into more detail, we conducted a preliminary study using Radamsa [6]: a standalone, state-of-the-art, open-source mutation engine. We found that

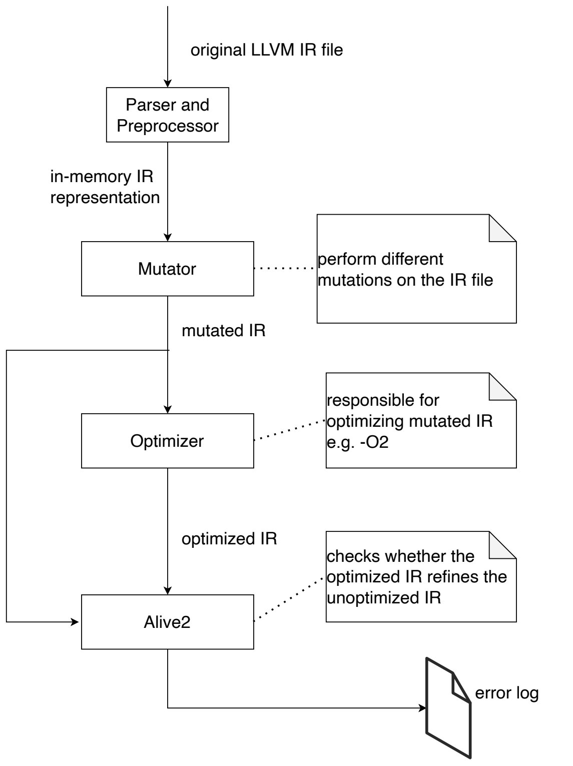
original LLVM IR file
Parser and Preprocessor
in-memory IR representation
Mutator
perform different mutations on the IR file
mutated IR
Optimizer
responsible for optimizing mutated IR e.g. -O2
optimized IR
checks whether the optimized IR refines the unoptimized IR
Alive2
error log

Fig. 3. Alive-mutate's workflow. The "mutator" component is new; the other components shown here existed previously. All components run inside a single UNIX process.

mutating LLVM IR using Radamsa has two problems. First, the vast majority of mutated LLVM IR files were invalid and could not be loaded by the compiler. Second, the mutants that could be loaded by the compiler were almost all boring, with something like a variable name or debug metadata being changed. In short, this approach was almost a complete waste of CPU time. We have no reason to think any other structure-blind binary mutator (such as AFL++'s [4]) would do better. LLVM IR files have extremely strict constraints on validity, and this appears to necessitate a domain-specific mutation approach. Moreover, LLVM's internal APIs provide a broad collection of functionality for inspecting and rewriting its IR. Using them, alive-mutate can create valid LLVM IR 100% of the time.

III Design and Workflow

Figure 3 gives a high-level overview of alive-mutate's workflow. This section describes how it works.

A. Parsing and Preprocessing

After inspecting its command line arguments, alive-mutate reads in a file of LLVM IR, which may be in either the human-readable text format or the compact binary bitcode format. Alive-mutate then checks that Alive2 can process each function in the input file without encountering errors; any function that cannot be handled by Alive2 (for example, because it contains an unknown form of metadata) is removed from the internal list of functions that are used in subsequent steps. Additionally, any function whose un-mutated form would cause Alive2 to signal a translation validation error is dropped: there is no point mutating these.

Following parsing, alive-mutate preprocesses each function by computing its dominance tree and scanning it to build a list of literal constants found in the code, that will be randomly changed later, during mutation. These steps are done early to avoid slowing down the main mutation loop.

B. Mutation

Alive-mutate makes a copy of the in-memory IR, and then selects and applies one or more mutation operators on each function in the IR, in order to create a mutated IR module. During the course of mutating a function, some of the cached information about it, such as its dominance tree, might be invalidated. To deal with this, we maintain a two-level data structure: it maintains a set of information that is specific to the current mutant, and then the cached information about the original mutation is considered immutable. Thus, queries first go to the mutant-specific information, falling back to the original version when the initial lookup fails. This arrangement allows us to always have correct dominance (and other) information for mutated functions, but also supports high performance by avoiding repeated dominance tree computations. The specific mutations supported by alive-mutate are described in Section IV.

C. Optimization

Once a module of mutated LLVM IR is available, alive-mutate invokes one or more LLVM optimization passes, as directed by its command line arguments. This can be a sequence of built-in passes, an out-of-tree pass loaded from a shared library, or a canned sequence of passes such as `-O1` or `-O3`.

D. Refinement Check

We invoke Alive2 to see if the optimized IR refines the unoptimized (but mutated) IR. Whenever the refinement relation does not hold, a bug has been found, and we log it into an external log file.

E. Looping and Repeatability

Finally, alive-mutate discards its current mutant and jumps back to its mutation stage. This entire loop repeats until LLVM crashes, until our tool has executed as many iterations as were requested, or else until a predetermined amount of time has elapsed.

Listing 4. An LLVM function that will serve as a running example for mutations

```
define i32 @test9(i32* %p, i32* %q){
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %q
    %c = sub i32 %a, %b
    ret i32 %c
}
```

Listing 5. One function-level attribute and one parameter-level attribute have been randomly added to the function from Listing 4

```
define i32 @test9(
            i32* dereferenceable(2) %p,
            i32* %q) #0 {
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %q
    %c = sub i32 %a, %b
    ret i32 %c
}

attributes #0 = { nofree }
```

Listing 6. The inlining mutation takes the function from Listing 4 and inlines a function other than the intended callee

```
define void @f(i32* %ptr){
    store i32 42, i32* ptr
    ret void
}

define i32 @test9(i32* %p, i32* %q) {
  %a = load i32, i32* %q, align 4
  store i32 42, i32* %p, align 4
  %b = load i32, i32* %q, align 4
  %c = sub i32 %a, %b
  ret i32 %c
}
```

Listing 7. The function from Listing 4 with a function call removed

```
define i32 @test9(i32* %p, i32* %q){
    %a = load i32, i32* %q
    ~~call void clobber(i32* %p)~~
    %b = load i32, i32* %q
    %c = sub i32 %a, %b
    ret i32 %c
}
```

These behaviors are specified using command line options.

Alive-mutate ensures that its runs are repeatable by logging an individual PRNG seed that led to the creation of each specific mutant. Also, it has a command-line argument for saving mutated IR files to disk where they can be subsequently analyzed. In a typical workflow, we run alive-mutate without saving files, to make fuzzing as fast as possible. Then, when an error is discovered, we re-run with the same seed but with file-saving turned on, in order to capture the IR file that triggers whatever bug had been previously encountered.

IV Supported Mutations

Alive-mutate supports mutations at the level of functions, basic blocks, and instructions. This section describes them in detail. Throughout, we will use `@test9` in Listing 4 as a running example.

A. Mutating Attributes

LLVM functions and parameters may be annotated with attributes that either force or permit the compiler to treat them specially. For example, in LLVM "a pointer is captured by the call if it makes a copy of any part of the pointer that outlives the call."[1] The `nocapture` parameter attribute asserts to the compiler that that parameter is not captured. At the function level, the `nofree` attribute asserts to the compiler that the function "does not, directly or transitively, call a memory-deallocation function (free, for example) on a memory allocation which existed before the call."[2] Attributes are a fruitful source of compiler bugs because it is easy for compiler developers to forget to consistently enforce their special semantics. Alive-mutate randomly toggles these attributes. Listing 5 shows an example; the first argument now guarantees that at least two bytes may be accessed by dereferencing it, and also the function promises not to free any previously allocated memory cells.

B. Inlining

LLVM, like other optimizing compilers, relies heavily on function inlining. We abuse its inliner for mutation testing by asking it to inline functions other than the intended inlining target, based on the hypothesis that this will perhaps create interesting results, when a function with a compatible signature is available. Listing 6 shows an example.

C. Removing Function Calls

Alive-mutate randomly removes "void" function calls; Listing 7 shows an example.

D. Shuffling Instructions

When a sequence of consecutive instructions lacks mutual internal dependencies, the instructions can be shuffled without breaking LLVM's SSA invariants. Alive-mutate randomly performs this shuffling. Alive-mutate precomputes maximal ranges of shufflable

[1]https://llvm.org/docs/LangRef.html#parameter-attributes

[2]https://llvm.org/docs/LangRef.html#function-attributes

Listing 8. The original version of @test9 had three non-dependent instructions in the order `a, call, b`; in this mutant they have been shuffled to be `b, call, a`

```
define i32 @test9(i32* %p, i32* %q){
    %b = load i32, i32* %q
    call void @clobber(i32* %p)
    %a = load i32, i32* %q
    %c = sub i32 %a, %b
    ret i32 %c
}
```

Listing 9. In this mutant, `%c` has been replaced with a new instruction and its usage is updated as well

```
define i32 @test9(i32* %p, i32* %q){
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %q
    %1 = mul nuw nsw i32 %b, %a
    ret i32 %1
}
```

Listing 10. An example of replacing a use of `%c` with a new random generated binary instruction

```
define i32 @test9(i32* %p, i32* %q){
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %q
    %c = sub i32 %a, %b
    %1 = ashr i32 %b, 10691696680
    ret i32 %1
}
```

Listing 11. An example of replacing one of `%b`'s arguments with a random SSA value, which ends up being a fresh function parameter

```
define i32 @test9(i32* %p,
                  i32* %q, i32* %0){
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %0
    %c = sub i32 %a, %b
    ret i32 %c
}
```

instructions during its initialization phase so that this mutation can be performed rapidly, when it is randomly selected. Listing 8 shows an example of shuffling. Because the first three instructions do not depend on each other, they could be rearranged freely.

E. Mutations on Arithmetic Instructions

Arithmetic operations make up a substantial fraction of many programs. Since LLVM optimizes them heavily, we mutate them fairly aggressively. Alive-mutate randomly:

- Changes the operation, for example turning an addition into a left-shift
- Swaps the two operands, for binary instructions
- Toggles any flags associated with an operation, such as the "no signed wrap" and "no unsigned wrap" flags that are present on the addition, subtraction, and multiplication instructions
- Replaces literal constants with randomly chosen values

Listing 9 shows an example; the original `%c` is replaced with a `mul` instruction with both math flags turned on, and the original operands are swapped. We consider LLVM's GetElementPointer (GEP) instruction to be arithmetic, even though it does pointer arithmetic.

F. Mutating Uses

A primitive that alive-mutate makes heavy use of is "for a given program point, randomly generate a dominating SSA value with compatible type." These conditions are necessary and sufficient for replacing an arbitrary SSA value in an LLVM function with a different value, without breaking any SSA invariants. This value might be one that already exists within the function (e.g., a function argument, or the result of some instruction), it might be a fresh literal constant, or it might be a fresh randomly generated instruction—whose operands are chosen by recursively invoking this same primitive. Alive-mutate randomly replaces SSA uses with values chosen by this primitive. Listings 10 and 11 contain examples.

G. Moving an Instruction

Moving an instruction around requires handling various conditions. For example, if B uses A, and we try to move B before A, then we must change A to some other available SSA value. Similarly, if we try to move A after B, we will have to update B's use of A. We handle both of these cases using the previously-mentioned primitive for randomly choosing (or creating) a dominating, type-compatible SSA value.

Listing 12 shows an example of moving an instruction forwards. Because `%c` uses both `%a` and `%b`, they are inaccessible after moving `%c`, and alive-mutate must find substitutes for them. In this example, one use is replaced with `%0`, which comes from a fresh function parameter; the other use is replaced with randomly generated constant.

H. Changing Bitwidths

Some of LLVM's optimizations are sensitive to bitwidth, and we wanted to stress-test these. However, randomly changing bitwidths is fairly tricky, because LLVM's type system insists, for example, that both of the arguments to, and the result of, a binary arithmetic instruction all have the same width. Thus, changing

Listing 12. %c is moved to the top and its both usages are replaced with a function parameter and a random constant

```
define i32 @test9(i32* %p,
                  i32* %q, i32* %0){
    %c = sub i32 1280583335, %0
    %a = load i32, i32* %q
    call void @clobber(i32* %p)
    %b = load i32, i32* %q
    ret i32 %c
}
```

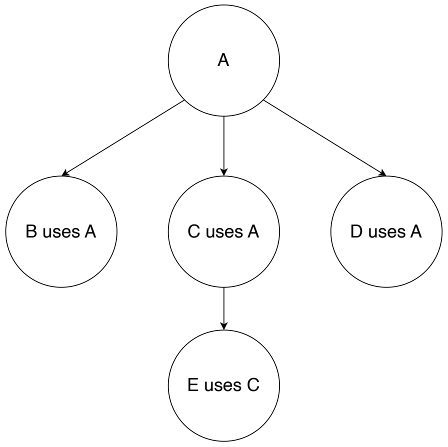

Fig. 4. Example: SSA use tree of a variable A before changing bitwidth

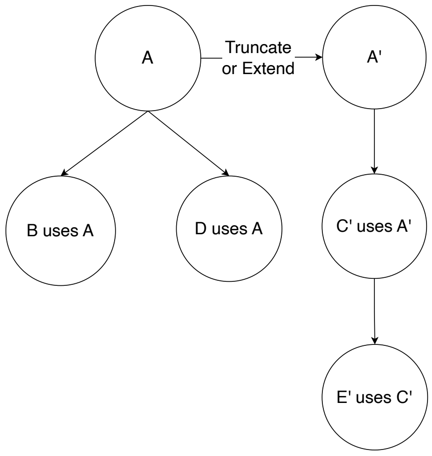

Fig. 5. Example: SSA use tree of a variable A after updating A, C and E

Listing 13. %c is replaced with %new0, the same operation but on size i26 (a 26-bit integer) instead of i32. Both operands are truncated.

```
define i32 @test9(ptr %p, ptr %q) {
    %a = load i32, ptr %q, align 4
    call void @clobber(ptr %p)
    %b = load i32, ptr %q, align 4
    %old0 = sub i32 %a, %b
    %1 = trunc i32 %a to i26
    %2 = trunc i32 %b to i26
    %new0 = sub i26 %1, %2
    %last = zext i26 %new0 to i32
    ret i32 %last
}
```

the width of a single SSA value tends to have a contagious effect on a function, necessitating updates to the widths of several, or many, other values. In the worst case, we would need to resize all children in a use tree, and we wanted to avoid that. Therefore, we only change the bitwidth of a selected path from a root node to a random leaf node.

For example, Figure 4 shows a hypothetical SSA use tree for a variable A, where all of A, B, C, D and E have the same bitwidth. Assume that we randomly choose a path A, C, E where we want to change the bitwidth starting from A. To do this, we create a new version of A that is either truncated or sign extended or zero extended to some other width, and then propagate the new widths along the path to E. Figure 5 shows the use tree after these updates have been performed.

Listings 13 is a concrete example on definition %c from i32 to i26. It uses two values %a and %b and its only user is a terminator instruction, %ret, without introducing new definitions. As a result, we perform two truncations, %1 and %2, on both operands and an extension %last back to i32 when it reaches the terminator. Value %c is replaced by %new0 with the same operation but on truncated values.

A complication is that certain LLVM instructions only work for certain widths. For example, the bswap intrinsic only supports 16, 32, or 64 bit arguments. Similarly, the icmp instruction can only produce a 1-bit output. Therefore, we only consider fully bitwidth-polymorphic binary instructions in use paths to be eligible for bitwidth changes.

I. Applying Multiple Mutations

So far, we introduced different kinds of mutations. When running Alive-mutate, we select a subset of applicable mutations and perform them sequentially.

Listings 14 shows an example of applying two mutations to the same function. It first moves %c before %b and updates the uses from %b to %1, a fresh generated smin function. Next, %c itself is changed

Listing 14. Multiple mutations are applied at once

```
define i32 @test9(ptr %p, ptr %q) {
  %a = load i32, ptr %q, align 4
  call void @clobber(ptr %p)
  %1 = call i32
        @llvm.smin.i32(i32 375689115,
                       i32 %a)
  %2 = mul nsw i32 %a, %1
  %b = load i32, ptr %q, align 4
  ret i32 %2
}
```

from a `sub` instruction to a `mul` associated with `nuw` and `nsw` flag.

V Experimental Results

We evaluate alive-mutate in two ways. First, we look at its ability to find previously unknown defects in the LLVM compiler. Second, we evaluate our claim that it is "high throughput."

A. Fuzzing the LLVM Compiler

LLVM has an extensive unit test suite containing 29,243 files in the LLVM intermediate representation. This suite includes regression tests that are added as compiler bugs are fixed, and also tests written alongside optimizations that are designed to increase developers' confidence in their new code. Although this test suite is extensive and useful, buggy commits still escape it, and make it into the source tree, and into released versions of LLVM.

We conducted a testing campaign where, over a period of about a year, we would build alive-mutate against the then-top-of-tree version of LLVM, run it for a while, and then inspect its results and report any bugs that it had found. In some cases, we also ran across bugs or other shortcomings in alive-mutate that we then fixed. During this testing campaign we focused both on LLVM's "middle-end" optimizations passes, using the set of passes implied by the `-O2` command line flag, and we also tested LLVM's AArch64 (64-bit ARM) backend. To test the AArch64 backend, we used an experimental branch of Alive2 that lifts 64-bit ARM assembly code back to LLVM IR, before performing a refinement check between the original (mutated) LLVM IR and the lifted LLVM IR. We found 33 bugs that can be divided into two categories:

- 19 bugs leading to incorrect code generation—i.e., flagged by Alive2 as refinement failures, and
- 14 bugs leading to abnormal termination of the optimizer—i.e., segmentation faults or assertion violations.

Table I summarizes the bugs that we found. LLVM's AArch64 backend was a fruitful source of bugs, but several of the bugs that we found in it ended up being defects in architecture-independent parts of LLVM's code generation infrastructure. In other words, these bugs could have affected architectures other than 64-bit ARM code; we simply happened to find them using the AArch64 backend. Additionally, InstCombine, the pass that contains many of LLVM's middle-end peephole optimizations, contained a number of bugs. This finding echoes an earlier one; in 2011, InstCombine was the single buggiest LLVM component, according to Csmith [12].

Listing 15. Code that triggered a crash bug in InstCombine

```
define i8 @smax_offset(i8 %x) {
  %1 = add nuw nsw i8 50, %x
  %m = call i8 @llvm.smax.i8(i8 %1,
                              i8 -124)
  ret i8 %m
}
```

Listing 16. Code that triggered a crash bug in the AlignmentFromAssumptions pass

```
declare void @llvm.assume(i1 noundef)

define i8 @align_non_pow2(ptr %p){
  call void @llvm.assume(i1 true)
    [ "align"(ptr %p, i64 123) ]
  %v = load i8, ptr %p
  ret i8 %v
}
```

Listing 15 corresponds to bug 52884 in the table; it shows a function that caused the InstCombine pass to crash. A compiler developer mentioned that "InstCombine is expecting InstSimplify to squash the pattern before it gets too far, but the analysis got thwarted by having both `nuw` and `nsw` on the add."

Listing 16 corresponds to bug 64687. In this case, an `align` in the operand bundle to the `assume` call specifies 123-byte alignment for `%p`. According to the LLVM Language Reference,[3] alignments that are not powers of two are allowed in certain situations. However, an optimization pass incorrectly assumed that all alignments are powers-of-two, leading to a crash.

Listing 17, which corresponds to bug 59836 in the table, triggered a miscompilation in InstCombine. A developer assumed that when two zero-extended values were multiplied together, the result could not overflow, resulting in an optimization that caused this function to return false. However, Alive2 found a counterexample when `%x` is 3363831808, in which case an overflow occurs. Thus, the version of the code optimized by InstCombine was wrong.

Listing 18 corresponds to bug 55129 in Table I. This bug happened because the backend attempted to

[3]https://llvm.org/docs/LangRef.html#assume-operand-bundles

TABLE I
LLVM BUGS FOUND USING ALIVE-MUTATE

Issue ID	Component	Status	Type	Description
53252	InstCombine	fixed	miscompilation	didn't update predicate in function 'canonicalizeClampLike'
50693	InstCombine	fixed	miscompilation	missing a simplification of the opposite shifts of -1
53218	NewGVN	fixed	miscompilation	need to merge IR flags of the removed instruction into the leader
55003	AArch64 backend	fixed	miscompilation	need to combine GSHL, GASHR, GSHL of undef shifts to undef
55201	AArch64 backend	fixed	miscompilation	when matching a disguised rotate by constant should apply LHSMask/RHSMask
55129	AArch64 backend	fixed	miscompilation	zero-width bitfield extracts to emit 0
55271	multiple backends	fixed	miscompilation	missing a freeze to ISD::ABS expansion
55284	AArch64 backend	fixed	miscompilation	an or+and miscompile within GlobalISel
55287	AArch64 backend	fixed	miscompilation	an urem+udiv miscompilation within GlobalISel
55296	multiple backends	fixed	miscompilation	didn't clear promoted bits before urem on shift amount
55342	AArch64 backend	fixed	miscompilation	sext and zext selection in promoted constant
55484	multiple backends	fixed	miscompilation	wrong match in in MatchBSwapHWordLow
55490	AArch64 backend	fixed	miscompilation	another sext and zext selection in promoted constant
55627	AArch64 backend	fixed	miscompilation	refine sext and zext selection
55833	AArch64 backend	fixed	miscompilation	conflict between the selection code in tryBitfieldExtractOp and isDef32
58109	AArch64 backend	fixed	miscompilation	wrong code generation in usub.sat
58321	AArch64 backend	open	miscompilation	miscompilation of a frozen poison
58431	AArch64 backend	fixed	miscompilation	wrong GZEXT selection GISel
59836	InstCombine	fixed	miscompilation	precondition of a peephole optimization is too weak
52884	InstCombine	fixed	crash	analysis got thwarted by having both "nuw" and "nsw" on the add
51618	newGVN	open	crash	PHI nodes with undef input
56377	VectorCombine	fixed	crash	created shuffle for extract-extract pattern on scalable vector
56463	InstCombine	fixed	crash	calling a function with a bad signature
56945	ConstantFolding	fixed	crash	the dyn_cast to a ConstantInt would fail with a poison input
56968	InstSimplify	fixed	crash	uncovered condition in detecting a poison shift
56981	ConstantFolding	fixed	crash	assertion is too strong
58423	AArch64 backend	fixed	crash	CSEMIRBuilder reuse removed instructions
58425	AArch64 backend	fixed	crash	udiv did not reach the legalizer
59757	TargetLibraryInfo	fixed	crash	signature for fprintf is wrong
64687	AlignmentFromAssumptions	fixed	crash	missing a corner case
64661	MoveAutoInit	fixed	crash	the assertion is too strong
72035	SROA	open	crash	wrong code in AllocaSliceRewriter
72034	VectorCombine	fixed	crash	wrong code in scalarizeVPIntrinsic

Listing 17. A miscompilation in pattern (zext a) * (zext b)

```
define i1 @pr4917_4(i32 %x) {
entry:
  %r = zext i32 %x to i64
  %0 = trunc i64 %r to i34
  %new0 = mul i34 %0, %0
  %last = zext i34 %new0 to i64
  %res = icmp ule i64 %last,
                     4294967295
  ret i1 %res
}
```

coalesce a logical "and" operation and a shift into a single AArch64 `ubfx` instruction. However, in this particular situation, that coalescing was incorrect.

Finally, Listing 19, which corresponds to Bug 55342, was caused when compiler developers missed a case when performing a type promotion.

Listing 18. Code that triggered a miscompilation bug in LLVM's AArch64 backend

```
define i64 @lsr_zext_i1_i64(i1 %b) {
  %1 = zext i1 %b to i64
  %2 = lshr i64 %1, 1
  ret i64 %2
}
```

Listing 19. Another function that triggered a miscompilation in LLVM's AArch64 backend

```
define i32 @f() {
  %1 = sub i8 -66, 0
  %2 = icmp ugt i8 -31, %1
  %3 = select i1 %2, i32 1, i32 0
  ret i32 %3
}
```

B. Throughput Experiment

A primary design goal for alive-mutate was for it to be fast. To check if it is, we compared it against the baseline fuzzing approach (depicted in Figure 2), where mutation, optimization, and translation validation are performed separately, instead of being integrated into the same program.

We randomly selected 200 LLVM IR files, each of them smaller than 2 KB, from the unit tests for LLVM's InstCombine pass. Out of these, we discarded six that triggered Alive2 errors, leaving 194 files to be used in the experiment. Then, for each of these files, we measured how long it took for alive-mutate and for the baseline approach to perform the same amount of mutation testing. For the alive-mutate case, we simply asked it to generate, optimize, and perform translation validation for 1000 mutated versions of the code in the file. For the baseline case, we wrote a loop in Python that repeated the following operations 1000 times:

1) mutate the file using a standalone version of alive-mutate
2) optimize the file using LLVM's standalone `opt` tool
3) perform translation validation using the standalone `alive-tv` tool

We ensured that the actual work performed under both conditions were exactly the same by seeding the PRNG in alive-mutate appropriately. We found that, on average, alive-mutate is about 12x faster than performing the same tasks using standalone tools.

In the best case (a file that contained very small functions and required Alive2 to do very little work), alive-mutate was 786 times faster than the combination of standalone tools. On the other hand, in the worst case (a file that caused Alive2 to spend a large amount of time doing SMT solving), alive-mutate was only 1% faster than the standalone tools—in this case the overheads due to file and process management did not constitute a significant fraction of the overall execution time.

VI Related Work

Domain-independent fuzzers, such as AFL [13] and AFL++ [4], have been highly successful in finding security-related defects in software systems. Similarly, Radamsa [6] is a format-independent mutation tool that has been successful in finding security-related defects. Tools like this, however, have not been nearly as successful in finding bugs in compilers.

Because LLVM's correctness is critically important to projects such as Android, iOS, and macOS, it has been specifically targeted by a number of previous testing tools. For example, LLVM has an instruction selection fuzzer [1] that is assisted by a random LLVM IR generator llvm-mutate [11], which supports four mutations including those that insert and remove instructions. Another structured fuzzer by Rong [10] found a number of crash bugs in LLVM. SRCIROR [5] is a mutation tool for LLVM that supports modifying arithmetic operators, integer constants, integer comparisons, and bitwise operators. Mull [2] supports six mutations on integer-typed values including negating conditions, changing arithmetic operators, and replacing a function call with an integer constant. FLUX [8] is the only mutation-based tool for LLVM that we are aware of that uses Alive2; it focuses on crossover mutations such as inlining one function into another, and connecting a collection of un-mutated functions into a sequence. In contrast with these existing tools, alive-mutate has nine distinct mutations that it performs, including several—changing the bitwidth of instructions, toggling undefined behavior flags, and replacing an SSA value with a randomly chosen dominating value—that are not implemented in any of the other tools. As far as we know, alive-mutate's design point, combining aggressive mutations with formal methods support and a throughput-oriented design, in unique.

VII Conclusion

We designed and implemented alive-mutate, a mutation-based fuzzing tool for the LLVM compiler that builds upon the Alive2 translation validation tool. Alive-mutate takes test cases written by humans and rapidly generates many similar test cases, in hopes of finding corner-case bugs in compiler optimization passes. So far it has found 33 previously unknown bugs in LLVM, 19 of which caused it to silently miscompile code.

Appendix

A. Abstract

This appendix describes how to build alive-mutate, how to use it to mutate any valid LLVM IR file, and how to reproduce the throughput experiment in our paper.

B. Artifact Check-list (Meta-information)

- **Program:** alive-mutate
- **Compilation:** g++ $\geq$ 10
- **Run-time environment:** Ubuntu 22.04, re2c, Z3, LLVM
- **Hardware:** Reasonably modern x86-64 machine running Ubuntu 22.04
- **Output:** Possible bugs, performance results
- **Experiments:** Running benchmarking scripts
- **Disk space required:** about 20 GB (including LLVM dependency)
- **Time needed to prepare workflow:** about 2 hours

Listing 20. An example output from a throughput experiment

```
Total: 1
Alive-mutate lst:[(0.9584662914276123,
  'test.ll')]
Discrete tools lst:[(8.755630254745483,
  'test.ll')]
perf lst:[(9.135042445472115, 'test.ll')]
Avg perf:9.135042445472115
Total not-verified:0
Not-verified files:[]
Total invalid file:0
Invalid files:[]
```

- **Time needed to complete experiments:** about 4 hours
- **Availability:** https://github.com/Hatsunespica/alive2/tree/CGO-2024-artifact
- **Code licenses:** MIT
- **Workflow framework used:** Python scripts
- **Archived:** 10.5281/zenodo.10205321 [3]

C. Description

1) How delivered

Our source code is available on Github: https://github.com/Hatsunespica/alive2/tree/CGO-2024-artifact

2) Hardware Dependencies

An x86-64 machine running Ubuntu Linux 22.04 is required

3) Software Dependencies

The Ubuntu package manager should be used to install Z3 and re2c. LLVM is also a dependency, but pre-compiled versions will not work; LLVM must be compiled following specific instructions in the Alive2 repository.

D. Installation

Please see the README at https://github.com/Hatsunespica/alive2/tree/CGO-2024-artifact for detailed installation instructions. Additionally, this blog post contains detailed instructions that may be helpful.

E. Experiment Workflow

1) Fuzzing LLVM IR Files

- Run the script *run.sh* *repo_path/benchmark/fuzzing*
- Add any valid LLVM IR files to *repo_path/benchmark/fuzzing/tests* and execute *run.sh* again.
- All mutants will be written to *repo_path/benchmark/fuzzing/tmp*

2) Throughput Experiment

- Run *bench.py* under *repo_path/benchmark/throughput*
- The user can add any valid LLVM IR files to *repo_path/benchmark/throughput/tests* and execute *bench.py* again.
- All result will be written to *repo_path/benchmark/throughput/res.txt*

F. Evaluation and Expected Result

1) Fuzzing LLVM IR files

All mutants can be found at *repo_path/benchmark/fuzzing/tmp*. They are slightly different from the original tests. For example, if you generate ten mutants for *test.ll*, those mutants will be named as *test0.ll*, *test1.ll*, and until *test9.ll*.

2) Throughput Experiment

All results will be written to *repo_path/benchmark/throughput/res.txt*. It will includes the number of total input files, the performance improvement on each input and the average inputs. For example, if we run our throughput experiment with only one *test.ll*, a possible record is shown in Listing 20 in *res.txt*. For *Alive-mutate lst* and *Discrete tools lst*, it includes the test case and seconds consumed. The *perf lst* shows the ratio of performance improvement for that test file. Finally, it also includes a list of files failed to pass the verification.

G. Experiment Customization

1) Fuzzing LLVM IR Files

The script *run.sh* calls the same alive-mutate command for every files in *tests*. Consequently, the user can change the command to satisfy the requirement. For example, the user can add *–pass='instcombine'* to perform *instcombine* optimizations instead of generate *O2* pass. Or can remove *–saveAll* to let alive-mutate only saves non correct cases. The user can type *alive-mutate –help* to check all arguments it supports. In addition, the user can replace *-n 10* with *-n X* where *X* is the number of mutants the user wants to generate, or with *-t 1* for keeping alive-mutate running for 1 second. Finally, the user can add any number of LLVM IR files to *tests* folder to run alive-mutate on those files in a batch.

2) Throughput Experiment

The script *bench.py* executes two workflows mentioned in the paper with all files in *tests*. The user can add any number of LLVM IR files to see the comparison on throughput. Besides, the user can change the *COUNT*, a global variable to control how many mutants a test case should generate. In our experiment, we set the number to 1000. In addition, we randomly selected 200 IR files with file size less than 2KB and 200 files with size larger than 2KB. The user can use any files where they are interested.

H. Methodology

Submission, reviewing and bodging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] Justin Bogner. Adventures in Fuzzing Instruction Selection, 2017. https://youtu.be/UBbQ_s6hNgg?si=ceLpoqhWI21dJQaL.

[2] Alex Denisov and Stanislav Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, 2018.

[3] Yuyou Fan and John Regehr. Artifact for alive-mutate paper at cgo 2024, November 2023. https://doi.org/10.5281/zenodo.10205321.

[4] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[5] Farah Hariri and August Shi. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 860–863, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Aki Helin. A Crash Course to Radamsa, 2016. https://gitlab.com/akihe/radamsa.

[7] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, 2014.

[8] Eric Liu. FLUX: Finding Bugs with LLVM IR Based Unit Test Crossovers. Master's thesis, University of Toronto, 2023. https://security.csl.toronto.edu/wp-content/uploads/2023/06/eliu_ms_2023.pdf.

[9] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Peter Rong. Improved Fuzzing of Backend Code Generation in LLVM, 2022. https://youtu.be/LfpmUxIuKgo?si=kMOtZIyOTbstc79P.

[11] Eric Schulte. llvm-mutate – mutate LLVM IR, 2013. https://eschulte.github.io/llvm-mutate/.

[12] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, 2011.

[13] Michał Zalewski. American Fuzzy Lop - Whitepaper, 2016. https://lcamtuf.coredump.cx/afl/technical_details.txt.