ZigRight: Memory Tracking for Zig

John Berberian, Jr.
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
ccg3sr@virginia.edu

Deebakkarthi Chinnasame Rani
Department of Computer Science
University of Virginia
Charlottesville, VA
brg8ve@virginia.edu

***Abstract*—Manual Memory management, though a relic of the past in most domains due to the features introduced by modern languages, remains a necessary evil for high-performance, low-level applications. Zig, a modern language — surprisingly, without any memory management— is positioned as a successor for C. Though it eliminates many of C's pain points, the onus is still on the programmer to manage their own memory. To alleviate the difficulty of this, we propose a tool to statically detect five different types of memory bugs. While our tool is incomplete,[1] it manages to perform quite well on hand-generated CFG examples.**

***Index Terms*—Zig, Static analysis, Memory safety, Linting**

I. Problem

Memory bugs have long been a programmer's bane. Even today, with modern languages and techniques, memory errors remain relevant: the Chromium project reported that "around 70% of [its] serious security bugs are memory safety problems" [1] [2]. Over the decades, various approaches have developed to address this issue: new programming techniques, languages, static analysis techniques, and dynamic instrumentation frameworks.

All of these approaches come with their trade-offs: programming techniques like RAII are helpful when applied correctly, but require strict enforcement to be effective; most memory-safe languages have much lower performance than the unsafe languages they replace, making them unsuitable for many applications—the few that don't (Rust comes to mind) require completely different program structuring techniques and have long compile times; static analysis must tread a fine line between overzealous flagging of false positives and overly timid exclusion of true positives; dynamic methods require a corresponding test suite that exposes the faults.

Zig, a newer language positioned as a modern C replacement, takes a different approach: where C often has a single global allocator (`malloc`), Zig treats an allocator as an object to be passed around throughout the program. This allows multiple allocators with different allocation strategies to coexist, and forces functions which require allocation to explicitly take an allocator object as a parameter.[2]

Zig has enjoyed fast-growing popularity in recent years: the 2025 Stack Overflow survey rated it the fourth-most-admired language [3]. It follows a similar philosophy to C—giving the programmer complete control over memory management—and reaps the consequent performance benefits. Zig places a greater emphasis on explicitness of program properties: functions may not be implicitly called by operators (as is common in higher-level languages), memory allocations may not be hidden, and there is no preprocessor to insert implicit code. This explicitness is intended to simplify the process of reasoning about code, improving maintainability.

Zig's approach to memory management is innovative, but can introduce new challenges. When we surveyed users on Reddit [2], we got a few useful comments.[3] The most interesting one addressed problems with multiple allocators.

> Tracking which allocator you are using and making sure that [its] state is copied, because if it is, it could lead to double free or null reads.
> —u/Armagidon_MC

Put simply, it seems that pointers allocated with one allocator must be freed with the same one. Critically, allocators may be copied to multiple places, and some allocators might have actions that implicitly free all the objects associated with them (arena-style allocation comes to mind). Another comment brought up concerns familiar to any experienced C programmer:

> Anything that shouts at you when you accidentally pass a pointer to stack allocated memory out of the function stack would be nice.
> —u/SilvernClaws

We were shocked to find that pointers to stack memory could be passed out of functions—surely, we thought, this is something the compiler can prevent? But no, such programs compiled cleanly—at least, initially. It happens that during the course of this project, a new version of the Zig compiler (0.16.0) was released, which began issuing warnings for returning dangling stack pointers.

Because Zig is a relatively new language, not many tools exist to help programmers. There are a few basic linters [4] [5], but they seem more focused on code style than code correctness. Dynamic approaches that operate on binaries

[1] Generating a CFG is left as an exercise to the reader.

[2] Actually, a function could create its own locally-scoped allocator and use it to allocate whatever memory it desired. This, however, would be *very* poor form, and immediately obvious.

[3] Alas, the useful comments were strongly outnumbered by the useless ones. Quite a few folks seemed to think we were asking "how would you like us to redesign the language for you?" rather than "what tools can we make for this language?"

rather than code (Valgrind [6], for example) will always be effective, but (as with all dynamic tools) require some test case to expose the fault. Generating such test cases can be difficult. Furthermore, Valgrind requires the use of the C allocator under the hood, which severely limits its applicability to Zig projects—Zig intentionally allows multiple types of allocators to be used for different contexts.

Somewhat late in our search, we discovered a tool that does almost exactly what we want: Yonemoto [7] wrote a tool that performs static analysis, using a complex lifetime calculation algorithm. Though an excellent tool, it has several shortcoming. `clr` is only available on Linux, which goes against one of the key features of Zig: portability. It also requires the user to use a custom compiler which is pinned to version 0.15.2. Unfortunately, this makes the tool nearly obsolete: version 0.16.0 is one of the more significant changes to the language, with a complete overhaul of the I/O interface.

We believe we can architect a better and simpler solution without these drawbacks. We would like to solve the following bugs:

Double free When a pointer is `free`d twice.

Cross free A unique challenge in Zig: when a pointer is `free`d using a different allocator than it was allocated with.

Memory leak via drop When a `free()` is forgotten, and a pointer is dropped.

Memory leak via clobber When a new call to `malloc()` overwrites an existing pointer.

Free before allocate When an uninitialized pointer is `free`d.

We initially hoped to detect use-after-free as well, but were unable to finish implementing that within the time we had.

II. Approach

Our approach involves performing data-flow analysis on a control flow graph of the program, keeping in mind the peculiar properties that come with passing around allocators. We introduce two concepts that are fundamental to our tracking: a *source* and a *sink*. We annotate all the variables in the program with their appropriate sources and sinks. After this annotation process, we search through the annotations to find patterns that match the signatures of memory bugs. This approach will give some measure of memory safety, without introducing the full overhead of a Rust-like ownership system.

Before discussing the details of our implementation, it is wise to review how Zig handles memory allocation. Unlike C, which has a single global allocation interface through `malloc()`, Zig requires all the allocation be declared up-front in the function prototype. Each Zig program receives an allocation object from `std.process.Init` that is passed to `main()`. All memory that is to allocated dynamically requires access to an allocator object (more of which can be constructed from the starting allocator). If a function doesn't take an allocator object as one of its argument then it doesn't allocate any memory. Though this may seem tedious, it is an intentional feature of Zig: making allocations completely clear simplifies debugging substantially, especially in embedded contexts where allocating 4K of scratch space in a function might crash the system.

However, this isn't without its flaws. It is very easy to forget to `.deinit()` or `.free()` variables, mix up allocators, free a variable twice, allocate a variable twice, and (generally) make all the usual careless mistakes associated with manual memory management.

1) CFG Annotations: As we are just dealing with dynamic memory, most statements aren't relevant to our analysis. Due to Zig's insistence on explicit allocation and explicit control flow, the locations of allocation and de-allocation become quite apparent, and ideally should be straightforward to mechanically annotate.

2) Memory Operations: We have categorized the sorts of memory operations into the following categories:

Allocation Explicit calls to a allocation method of a `std.mem.Allocator` object.
An example would be the oft-used formula `var foo = gpa.alloc();`.

FunctionCall These are functions that take in an allocator and (optionally) return back an object or a pointer.
Examples: `var foo = bar(gpa);` `baz(foo, gpa);`

Deallocation These are calls to deinitialization methods of a `std.mem.Allocator` object.
Example: `gpa.free(foo);`

Deinit Sometimes it is possible to just call `.deinit()` on objects themselves. These objects store a pointer to the allocator that they were created with in some internal field, and can in turn use that to free their own memory.
Example: `foo.deinit();`

DeinitExplicit In similar vein to the last one, these are methods on objects that deallocate the space for them. But this sort is for objects that don't store their allocators: these take in the `std.mem.Allocator` object explicitly.
Example: `foo.deinit(gpa);`

These 5 types cover the vast majority of instances of memory allocation/deallocation.

3) Source States: We have introduced the concept of the source and sink earlier. The source is the location of allocation and the sink is the location of deallocation. In complex program there may be multiple possible sources or sinks (depending on branching) or none (if there were a bug). We need some sort of structure to capture this. We describe the `SourceState` of a variable as a tagged union:

Uninit This is the default state. This is applicable when a variable is set to be `undefined` or after the variable has been free.

Alloc (allocator) This is the state of being initialized. The variable has been allocated using the `allocator`.

Maybe (allocator) This is was put in place to handle mark the results of *opaque*[4] functions that take in `allocator`.

[4]That is, functions whose source code we have no access too, e.g. FFI calls.

We never ended up writing support for opaque functions, but here we are.

FnInput This is a placeholder for the sources of function parameters. We chose to analyze per-function for simplicity and then substitute the results in. The true sources of these are in the *caller's* scope. This placeholder allows us to analyze each function once and then reuse the results.

4) Sink States: Similarly for sinks we have,

Uninit No sink has been encountered.

Dealloc (allocator) An `allocator` freed the variable.

Maybe (allocator) To model opaque functions of the form `foo(bar, allocator);` to which we lack source code access.

FnInput For functions that take in some pointer `*foo` and call `foo.deinit()`, we need to know the `gpa` with which it was allocated in the caller's scope. `FnInput` is used to model this scenario, as a placeholder to remind us to substitute the caller's source set in here, with some translation.

5) CFG Nodes: Each CFG nodes stores the following information:

- Incoming edges `[]CFGNode`
- Outgoing edges `[]CFGNode`
- SourceDictIn `{Variable: Set(SourceState)}`
- SinkDictIn `{Variable: Set(SinkState)}`
- SourceDictOut `{Variable: Set(SourceState)}`
- SinkDictOut `{Variable: Set(SinkState)}`
- MemOp `enum MemoryOperations`

There are a few more fields to deal with the hairy details of graph traversal, but these are the important ones. The nodes encapsulate the global state at the point of their execution. We represent the variables' allocation state using a dictionary mapping from variables names[5] to a set of `SourceState/SinkState`. Because each node's allocation operation is able to change these dictionaries, we save separate copies for the combined state flowing in and the state flowing out.

We use a set of `SourceState/SinkState` for each variable because the different paths through the CFG might result in different allocation states, and we want to be able to represent that complexity. Whenever two paths merge in a node, their sets of allocation states for each variable are combined.

A bit of complexity is introduced by function calls: we must support all manner of functions, including those that free their arguments and/or allocate memory. We decided, as mentioned earlier, to describe them by setting all the arguments' source to `FnInput` and then taking the final source/sink state from the function's end node. This gives us a view of how the function changes its inputs' allocation state. There are, however, a few subtleties worth dwelling on: first, we only support tracking allocated memory that is returned as an argument—we don't support placing the output pointers through indirection into the inputs of the function; second, because we limit ourselves to one memory operation per syntax node, we do not support such constructions as `f(gpa.alloc(), gpa)`— this will need to be split up into two statements (this could be automatic, but isn't); third, special difficulty arrives when it comes to freeing arguments that store their own allocators—we must record the sink state as `FnInput` and substitute it at the call site later on.

The complexity of all this is increased at a higher level, as well: how do we schedule our visits of the CFG nodes? Should we perform one pass over each node, or accept multiple? We used the approach suggested in class: a repeated depth-first search over the node graph, recalculating the allocation properties on each visit, until none of the nodes' states change anymore. This is *almost* good enough. Unfortunately, it struggles with loops: if a variable is unconditionally allocated earlier in the program, and the allocation state is not modified within a loop, a blind application of this algorithm will result in an erroneous `Uninit` being introduced into the allocation state: the allocation state on the loop's return branch is uninitialized, which looks just like a branch from sometime earlier in the in which the variable was not allocated. We considered using some advanced loop-detection techniques to solve this, but decided that the simplest solution was to just track which nodes' allocation states have been visited at least once, and only accept implicit (that is, empty-dict) `Uninit` entries from those. With sufficient iterations of the graph search algorithm, this works itself out to be equivalent.

6) Rules: We detect memory bugs by checking for the violation of the following rules on all CFG nodes.

- Attempting to set the `source[var]` when its value is already `Alloc` and `sink[var]` only contains `Uninit` → `[MemoryLeakClobber]`
- `sink[var] = Uninit` and `source[var] = Alloc` at the end of the program → `[MemoryLeakDrop]`
- Overwriting `sink[var]` when it contains only `Dealloc` → `[DoubleFree]`
- `sink[var] = Dealloc(gpa)` but `source[var]` doesn't have `Alloc(gpa)` or `Maybe(gpa)` → `[CrossFree]`
- Accessing `var` when `sink[var]` only has `Dealloc` → `[UseAfterFree]`[6]

```
fn foo(gpa: std.mem.Allocator) void {
    var bar = gpa.alloc();
    gpa.free(bar);
    gpa.free(bar);
}
```

Listing 1: Double `free()`

[5]Actually, canonicalized token numbers: the first appearance of that token within that variable's scope.

[6]We didn't end up implementing this one, because we didn't include access tracking in our handwritten CFG.

```
start.in
        source: { }
        sink: { }
start.out
        source: { gpa: .FnInput }
        sink: { }
node1.in
        source: { gpa: .FnInput }
        sink: { }
node1.out
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { }
node2.in
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { }
node2.out
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { bar: .Dealloc(gpa) }
node3.in
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { bar: .Dealloc(gpa) }
node3.out
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { bar: .Dealloc(gpa) }
end.in
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { bar: .Dealloc(gpa) }
end.out
        source: { gpa: .FnInput, bar: .Alloc(gpa) }
        sink: { bar: .Dealloc(gpa) }
```

Listing 2: State of the nodes

up.

```
pub fn two_alloc() !void {
    const smp_alloc = std.heap.smp_allocator;
    // A: smp_bytes.source = smp_alloc
    const smp_bytes = try smp_alloc.alloc(u8, 100);

    var debug_alloc_generator:
    ↪  std.heap.DebugAllocator(.{}) = .init;
    const debug_alloc =
    ↪  debug_alloc_generator.allocator();
    // A: debug_bytes.source = debug_alloc
    const debug_bytes = try debug_alloc.alloc(u8,
    ↪  100);

    // Whoops, swapped allocators!
    // A: smp_bytes.sink = debug_alloc
    something_and_free(smp_bytes, debug_alloc);
    // A: debug_bytes.sink = smp_alloc
    something_and_free(debug_bytes, smp_alloc);
    // Error: source-sink mismatch: debug_bytes,
    ↪  smp_bytes
}

// A: sets thing.sink = alloc
pub fn something_and_free(thing: []u8, alloc:
↪  std.mem.Allocator) void {
    // A: thing.sink = alloc
    defer alloc.free(thing);
    // And do something interesting, maybe.
}
```

Listing 3: CrossFree

7) Examples: The code snippet in 1 represents a simple memory bug. `bar` is allocated but never free, leading to a memory leak.

We see the progression of the states in 2. Each statement is a basic block on its own. Since we don't know the source of the argument `gpa` it is annotated with `FnInput`. It can only be resolved in the caller scope.

We see that `node1`'s allocation is reflected in `bar`'s source being set to `Alloc(gpa)`.

In the next node we see the deallocation being reflected as `bar`'s sink being set to `Dealloc(gpa)`.

When we are evaluating `node3`, we can detect a `DoubleFree` using rule 3 from above. We see that `sink[node3]` only contains `Dealloc`.

A memory bug unique to Zig is `CrossFree` illustrated in 3. The two variables are freed with the wrong allocators. Usage to multiple allocators is a very common design pattern in high performance applications. Imagine, for instance, a game where thousands of entities are spawned and destroyed in a second. Issuing a syscall for each and every one of them is a asinine. A more efficient approach is to allocate a big block of memory first and then use a secondary allocator to manage that chunk ourselves. This way we only incur two syscalls: one to initially ask for the large chunk and one at the end to clean everything

A. Limitations

As with most static analysis tools, we expect that our approach will struggle with loops that have conditional calls to `free()`. We have not yet decided whether we would prefer to emit a warning or remain silent in these cases; in any case, such code is very poor style.

Because we use the raw tokens to track a variable throughout the program, we are not able to support any sort of containers: any pointer trickery, whether aliasing or complex indirection, will fool our tool. Similarly, using different allocators for two fields of a `struct` (or storing variables packed in a `struct` to begin with) or using the reference to an allocator stored inside an object to allocate more objects are both unsupported.

We don't support derived allocators or arena allocation (an awesome feature of Zig in which one may allocate large chunks of memory at a time, packed efficiently with many tiny structs, so long as one is fine with only being able to free them all at the same time). We also don't support using `deinit()` to do things other than releasing memory; we assume it always releases memory, but it is a user-implemented method, so it need not follow any standards of decorum.

We also assume that the variables are only muted byt the single-threaded function currently running, with no outside

influence. As such, we don't support multithreading or global variables.

Because of the way that we annotate functions, we also don't support recursion or returning function pointers. In short, we don't think our tool ended up being useful for anything except toy programs. We hope the core ideas of this tool can be useful in some more advanced tool, but we find ourselves quite incapable of constructing one such.

III. Evaluation

Quantifying the usefulness of a linter tool is very hard. Such tools are not meant to catch bugs—rather, they prevent them from happening by providing rapid feedback to the programmer.

We were particularly limited by our inability to create a CFG generator. Despite many long nights, we found ourselves quite stumped. Usually, compilers generate CFGs on some sort of intermediate representation, but we were trying to generate one directly from the source code's AST. Because of the newness of the language, there was no documentation of the Zig compiler's IR, and very minimal (and somewhat outdated) documentation of the AST. In general, we felt that we were very much on our own.

The control flow of the Zig language is also somewhat more complex than most: it permits labeling of arbitrary blocks, and "breaking out" to those blocks from arbitrary points within those blocks, optionally with a value that the block should evaluate to. Such labels are particularly handy when attempting to `break` or `continue` within nested loops: you can specify precisely which loop you meant! Unfortunately, these also make construction of the CFG much more complex: we can no longer use a simple recursive method, as the break statements may reach up an arbitrary number of levels.

We were hoping to run our tool on a large codebase, but because we couldn't get that bit working, we were limited to programs whose CFGs we could construct by hand—that is, programs under 10 lines. While it is difficult to create complex behavior in such a small space, we were able to create programs that exhibited each of the flaws that our tool was intended to detect, along with a few tricky but correct programs.

In the 12 programs that we constructed, we were able to fully test all aspects of our tool, including the most complex feature, function call substitution with allocation and freeing. All that remains to be seen is how the runtime scales to larger program sizes; we are confident that the core features are working well.

IV. Conclusion

We were successful in creating a tool that can analyze Zig's unique semantics with regard to memory safety. Unfortunately, we were severely hampered by our inability to create a CFG generator, which was worsened by the poor state of Zig's internal compiler documentation. We believe that creating such a generator should be quite straightforward for someone with sufficient familiarity with the Zig compiler's internals, but we did not have time in this project to familiarize ourselves with the 455KLOC codebase.

The tool that we created, however, performs admirably on the hand-written CFG examples that we created, which were intentionally crafted to represent a diverse range of possible memory allocation strategies. We believe that the core techniques and dataflow rules described in this document could be quite useful when coupled with a complete CFG generator.

Our repository may be found on GitHub.

References

[1] T. C. Project, *Memory safety*, https://www.chromium.org/Home/chromium-security/memory-safety/, 2015.

[2] Reddit, *Pain points with zig?* https://www.reddit.com/r/Zig/comments/1s7xtjx/pain_points_with_zig/, 2026.

[3] S. Overflow, *2025 stack overflow developer survey*, https://survey.stackoverflow.co/2025/technology#2-programming-scripting-and-markup-languages, 2025.

[4] K. Wagner, *Zlinter - linter for zig*, https://github.com/KurtWagner/zlinter, 2026.

[5] S. Petunin, *Zwanzig: A static analyzer and linter for zig*, https://github.com/forketyfork/zwanzig, 2026.

[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007, ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. [Online]. Available: https://doi.org/10.1145/1273442.1250746.

[7] I. Yonemoto, *Clr: Checker for lifetimes and other refinement types*, https://github.com/ityonemo/clr.