GO2 TEAM MEETING #2

JAN 16, 2020

Weekly progress

1. Established a prototype of static race checker that can detect the false negatives missed by the reference tool
2. Ported NEO to Go runtime

Criteria for identifying *race pairs*

1. at least ***one*** instruction has to be write
2. cannot ***both*** be atomic
3. access the ***same*** memory location
4. operate in ***different*** goroutines
5. ***NO*** happens-before relation (union of PO and SO)

reference tool

Case 1 - shows that pointer aliases are out-of-scope for this static analysis algorithm.

Case 2 - shows that the algorithm ONLY considers data race between an anonymous function and its outside function, which in real-life could rarely be the case as data races often occur in named functions.

Case 3 - shows that global variables are out-of-scope for this algorithm.

GopherCon 2019 Tutorial Session - Tracking Inter-Process Dependencies

Expectation:

- To deploy ***small changes often***

Questions:

- What is the ideal system?
- What are the obstacles of implementing Nanoservices (serving only a ***single*** request)?

- How to keep track of new services added to the system?

Solution:

- Reveal dependency information of each service, and report any changes
  eg. If a pull request calls a new library that uses a new service -> change request could report: "new dependency added"
- Reveal dependencies between services across the system

Plan of Action

- find interesting function calls
- call destinations will be the dependencies of the service
- service dependencies (edges of the system graph) are put together to make a graph of how things are calling through the entire system

Go's `analysis` package has a term called `Pass`, which describes the action of running an Analyzer on a single package.

In the subpackages, there are reuseable passes. One of them being `buildssa`.

Plan of Action for each package Pass

- produce SSA results using `buildssa`
- build call graph using rta.Analyzer
- mark the paths along the call graph that lead to interesting calls
- report unmarked paths as linter errors (can be used as a PR status check)
- report destination markers as dependency data

SSA

- intermediate representation that the compiler can transform code into
- allows for better analysis of what the code is doing
- build an SSA program first and then analyze
- easy to tell where something is used and where it isn't, due to single assignment
- WARNING: the interface is experimental and likely to change

Fact for an analysis:

- the ***same*** analyzer storing information about a package it has already seen.

A Fact can be stored about an object or a package in general.

Result of an analysis:

- one analyzer can use the Result of another analyzer; however, an analyzer ***cannot*** read the Facts of another analyzer.

Potential Issues

- if no Facts are declared, then no dependency tree behaviors, and incomplete syntax analysis of dependent packages
- buildssa is ***not*** modular, each Pass builds a new ssa.Program.

`singlechecker` library, parses command line flags, so you dont have to figure out which packages are going to be analyzed

- `singlechecker.Main()` calls os.Exit, point of no return

Undesirable results

- multiple Passes, each with separate ssa program builds
- limitations on Rapid Type analysis package, less precise than pointer package even though the algorithm is faster.

-> replace RTA with Andersen's pointer analysis

-> use a simple driver
`packages.Load`
`ssautil.AllPackages` and `(*ssa.Program).Build`
`pointer.Analyze`

New Attempt

- False positives as call graph analysis does't precisely follow actual runtime execution path
- Pointer analysis -> not all reflection operations are supported. eg. addressable `reflect.Values` are not implemented

-> service-oriented architecture instead of monolithic adds flexibility but increases complexity.