# Interprocedural static analysis for Go with closure support

Alexey Borodin*, Varvara Dvortsova*†, and Alexander Volkov*
* *Ivannikov Institute for System Programming of the RAS*
† *Lomonosov Moscow State University*
Moscow, Russia
E-mail: {alexey.borodin, vvdvortsova, volkov}@ispras.ru

***Abstract*—We present an interprocedural static analysis to detect errors in the Go source code. The analysis supports most of the language features, while the main focus of the paper is closures and `defer` statements. The analysis we have developed demonstrates good scalability and performance. It is able to analyze a project of 1.1 million lines in 12 minutes.**

***Index Terms*—interprocedural static analysis, symbolic execution, data flow analysis, flow-sensitive analysis, golang, closures, defer, svace.**

## I. Introduction

The spread of Go language in the modern software development has been continuously increasing during the last years. Now it occupies 11th place in the TIOBE index [15]. Despite its popularity we are not aware of any interprocedural static analysis tools to detect errors in the Go source code. We have been able to find only linters for Go such as STATICCHECK [13], GO-CRITIC [9], ERRCHECK [7].

Functions in Go are first-class citizens, which means that a function can be assigned to a variable, passed as an argument to another function and returned from a function.

*Closure* is an anonymous function that refers (i.e. captures) free variables of its enclosing function scope.

One more Go-specific feature is `defer` statement. It allows to designate a function call to execute at the time, when the enclosing function exits (returns). In particular, the specified function to call can be an anonymous function or a closure. Use of `defer` statement may simplify significantly the logic to release acquired resources. Use of closures to define functions in `defer` statements is very typical and widespread in Go projects.

Listing 1 demonstrates using a closure within the `defer` statement. Variable `file` is the captured variable in this example:

```
func checkAll(files []string) int {
  for _, path := range files {
    file, err := os.Create(path)
    if err != nil {
      return 1
    }
    defer func() {
      file.Close()
    }()
  }
  return 0}
```

Listing 1. Closure with `defer`

The goal of our work is to create static analysis that is able to detect errors in Go programs and to track the semantics for most of its features, while the main focus of the current paper is Go closures and `defer` statements support.

For our analysis we use an approach similar to the one from PREfix [5]: it is a summary-based analysis, the analysis is heuristical: though it aims to cover most of the possible execution paths of a procedure, it may miss some cases.

We develop our analysis within SVACE static analyzer framework [10]. SVACE static analyzer is a tool for automatic software defect detection in programs written in C, C++, C#, Java, Kotlin, and Go.

## II. Build process

SVACE workflow design consists of two stages:

1) *build*, which builds low-level intermediate representation for a Go program;
2) *analysis*, which runs an interprocedural summary-based analysis to detect errors.

Fig. 1 gives an overview of SVACE analysis process. During the build stage SVACE BUILD CAPTURE tool runs the original project build, intercepts the `go` command invocations and uses the captured information about these invocations to launch the modified SSADUMP utility for each captured module. SSADUMP generates SSA-based [6] intermediate representation, and SVENG uses it as its input data.

SSADUMP provides for each function a list of anonymous functions and closures that it encloses. In addition, each function has a list of captured variables, which is not empty, if the function is an anonymous function and lexically captures these variables in a closure.

There is a special instruction `makeClosure` to create a closure in a function, which constructs a special functional object to store a pointer to function and pointers to the captured variables. The result of this instruction is intended for use in call instructions to call the constructed closure.

## III. Analysis

### A. Intermediate representation

We treat the program being analyzed as a set of procedures (functions). The intermediate representation (IR) for a procedure is a control flow graph (CFG). Instructions are the nodes of this CFG and are one of the following:



979-8-3503-9853-3/22/$31.00 

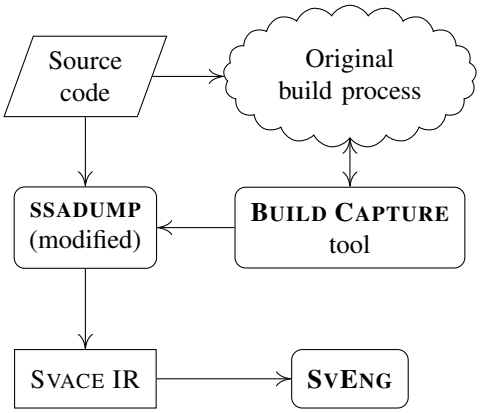


Figure 1. Analysis scheme

- $r$ `:= alloca()` — allocate memory on stack;
- `pmove` $val$`,` $ptr$ — store value $val$ into memory at address designated by $ptr$;
- $r$ `:= deref(`$ptr$`)` — read value $val$ from the memory at address designated by $ptr$;
- $r$ `:=` $val_1$ `+` $val_2$ — calculate sum of values $val_1$ and $val_2$;
- `return` $val$ — return value $val$ from the current function (only a single value);
- $r$ `:= makeClosure(`$func$`,` $bindings$`)` — create a closure object for the function referenced by $func$, where $bindings$ are the list of pointers to the captured variables;
- $r$ `:= call` $func$`(`$arg_1$`,` $arg_2 \dots arg_n$`)` — call the function or the closure object referenced by $func$, where $arg_1 \dots arg_n$ are its arguments.
- `defer` $func$`(`$arg_1$`,` $arg_2 \dots arg_n$`)` — defer a call to a function or a closure;

We represent global variables through the fields of the special parameter `global`, the analysis simulates its passing for each procedure call.

### B. Intraprocedural analysis

Our intraprocedural analysis is based on symbolic execution with state merging. Symbolic execution uses symbolic values to model data values. Symbolic values are symbolic variables that represent initial values of the procedure parameters, or symbolic expressions for the results of operations on other symbolic values and constants. State merging technique helps to prevent path explosion.

Let $R$ be the set of program SSA variables. Set $V$ is the set of symbolic values, and its subset $M \subset V$ is the set of references. References model memory locations in a program.

Let $B$ be the set of analyzed properties. The elements of this set have $join$ operation defined on them that for two input values calculates the result value describing both input values. This operation may result in the least upper element $\sqcup$ if $B$ is a semilattice. We do not require such restriction for $B$, but it is desirable in most cases.

Program execution states are defined via functions $val : R \cup M \to V$, $alias : V \to M \times M$, $attr : V \to B$. Program states are associated with CFG edges. A program state contains information about symbolic values for SSA variables and references (`val`), aliases for symbolic values (`alias`), and properties for all symbolic values (`attr`).

An analysis step calculates for each instruction a state on its output edge from the state on its input edge. For convenience all the nodes of CFG have one input edge and one output edge with the exception for split and join nodes. At split nodes the analysis simply copies states from their input edge to the output edges. At join nodes the analysis creates the state for the output edge that describe all the possible states from the input edges.

If there are no loops in an analyzed procedure, the analysis processes each instruction only once. For loop analysis we use a heuristic approach. First, we find strongly connected components (SCC) of CFG. For each SCC the analysis makes two iterations. For the output edges of an SCC the analysis joins program states on each iteration. See [2] for more details. Such analysis allows to traverse most of the paths in loops and at the same time the majority of the program instructions are visited only once. This features allows to achieve high speed and good scalability of our intraprocedural analysis.

Our intraprocedural analysis defines transfer functions for the IR instructions as follows:

- `a = alloca()`: $val[a \to v_a]$ .
  Here $v_a \in V$ is a fresh variable to represent the value of SSA variable $a$. This transfer function creates a state, where $a$ has a unique symbolic value.
- `a = deref(p)`: The result depends on the contents of $alias(val(p))$ set. There are three possible cases:
  - $\emptyset$ : $val[a \to v_a],\ alias[v_a \to m]$
  - $\{x\}$ : $val[a \to v_x],\ v_x = val(x)$
  - $X$ : $val[a \to v],\ v = joinval(X)$
    Function $joinval$ creates new symbolic variables that represent properties of all joined variables.
- `pmove p, a`: Again, the result depends on the contents of $alias(val(p))$ set. There are three possible cases:
  - $\emptyset$ : $val[m \to val(a)],\ alias[val(p) \to m]$
    Where $m \in M$ is a fresh variable to represent the memory reference for pointer $p$.
  - $\{x\} : val[x \to val(a)]$
  - $X$ : $val[x \to joinval(val(x), val(a)) \mid \forall\ x \in X]$

### C. Interprocedural analysis

Our interprocedural analysis uses summary-based approach, which we described in depth in [3]. Since the support for closures interferes closely with the interprocedural analysis, we highlight first its main features below to make the specific details of closure analysis clearer further.

The summary-based analysis creates a summary for each processed procedure. A summary models certain properties of the procedure's behavior. The subsequent analysis of the calls to a processed procedure uses its generated summary.

Each detector can propagate its specific properties. For interprocedural analysis it needs to define two summary-related

operations: $create\ summary$ to store the related properties in a summary while processing a procedure exit point and $apply\ summary$ to use them at call instructions.

Summary-based analysis is context-sensitive, since the result of applying a summary depends on the call context.

A summary may use symbolic values as a part of particular behavior properties it models. For operation $apply\ summary$ analysis framework translates these symbolic values to caller context and matches the symbols between callee and calling procedures.

When a summary is applied, formal arguments from the summary are matched to actual arguments from the context. After that other dependent values in the summary are matched to their corresponding elements in the caller context. The rules are the following.

Let $a$ be an actual argument, $m$ be a formal argument, $S$ be the program state at the input edge of a call instruction, $R$ be the summary for the called procedure, $val_S(a) = va$, $val_R(m) = vm$. Then element $va$ will be matched to the element $vm$. If $val_S(a)$ does not exist, then a fresh element will be created and updated at the output program state for instruction. For pointed elements the rules are more complex because $alias_S$ returns a set of aliases. Lets $val_S(a) = sa$, $val_R(m) = dm$. $dm$ will be matched to all elements from set $sa$.

Such matching is performed for all elements in the summary. For all the matched elements analysis framework notifies the detectors. Interprocedural detectors may transfer the properties they propagate from the summary to the caller context in accordance to their specific logic.

Note, that not all the symbolic values make sense outside an analyzed procedure and can be potentially translated to caller. For instance, values of local variables that are not either written to parameters or returned from the procedure make no sense to be translated.

As a part of summary creation implementation the analysis framework builds the set for visible symbolic values (visible set) and initializes it first with function parameters and return value. Then it recursively adds symbolic values when they are reachable by one *step* from any value in the set, where step is a dereference or a shift by an offset operation.

The analysis framework triggers $create\ summary$ operation at the exit point of an analyzed function and uses the program state for it, where it removes all the properties related to the symbolic variables that are not in visible set. Thus a procedure summary models program execution state at procedure's exit point omitting local variables and intraprocedural properties. At the same time a summary models side effects of the procedure, since $val$ we defined above denotes side effects at each CFG edge.

The main reason to use a summary-based analysis is its *scalability*, since in the case of recursive calls absence, each procedure is processed only once. In our approach we do not analyze call graph loops (which results from recursive calls) in a specific way, i.e. our analysis just breaks these loops.

Nevertheless summaries can grow up to huge sizes on real projects. At the same time a great part of the information collected in summaries might remain unused by the subsequent analysis, since at the moment of summary generation it is not yet known, if a particular summary will be required in any context. It is a downside of the summary-based approach.

We have an observation that developers tend to make their programs as simple as possible, while the summaries of big sizes correspond to functions with very complicated logic. Thus, exceeding a threshold for a summary size may indicate itself a potential problem of the corresponding function source code. This allows us to use such a threshold without a significant decrease of the analysis precision. The current implementation limits summary size to 300 symbolic values.

### D. Closures

Our approach for captured variables resembles the one we use for global variables. Globals are represented through the fields of an artificial parameter, which we name `global` in our implementation. For captured variables modeling we add one more artificial parameter to the procedure. Unlike globals, a captured variable comes from a particular function scope, so the fields of this new parameter differ for each closure, since they depend on the list of the variables that are captured in the closure's body.

Let's consider a simple example with a closure use:

```
func foo() int {
  x := 10
  y := 20
  z := func(a int) int {
     return a + x + y
  } (5)
  return z
}
```

Listing 2. Closure use

The following IR will be generated:

```
func foo() int {
  addr_x := alloca()
  addr_y := alloca()
  pmove 10, addr_x
  pmove 20, addr_y
  closure_1 := makeClosure(anon_1, addr_x, addr_y)
  z := call closure_1(5)
  return z
}

func anon_1(a int, outer) int {
  x := deref(outer.addr_x)
  y := deref(outer.addr_y)
  r := a + x + y
  return r
}
```

Listing 3. `makeClosure` and call

SSA-variables `addr_x` and `addr_y` represent addresses of local variables `x` and `y` correspondingly. Note that variables themselves are not present in IR directly and are handled through their addresses. Instruction `makeClosure` creates a functional object `closure_1` that stores `addr_x`, `addr_y`

to reference the captured variables. The analysis propagates the `closure_1` value through the CFG, and when processing the `closure_1(5)` call instruction it will add these captured variables to the call parameter list.

In a summary-based analysis callee functions are analyzed prior to their callers. So, first, closure's anonymous function `anon_1` will be analyzed. The IR for the closure function provides special parameters `outer.addr_x` and `outer.addr_y`, which are handled similarly to globals. The analysis creates the summary that contains information that the result value `r` is: `r = deref(outer.addr_x) + deref(outer.addr_y) + a`, where `a` is the closure argument.

In order to apply a summary in the caller context, analysis matches symbolic variables of the summary with the corresponding expressions in the caller context:

| | |
|---|---|
| a | 5 |
| outer.addr_x | addr_x |
| outer.addr_y | addr_y |
| deref(outer.addr_x) | 10 |
| deref(outer.addr_y) | 20 |
| z | r |

Since all the argument values to calculate `r` are known in this context, the analysis calculates the value for the matching `z` as `z = 10 + 20 + 5`.

### *E. Analysis of `goroutines`*

Go provides goroutines, which are a kind of lightweight execution threads. They aim to write efficient concurrent code and are widely used. We haven't implemented any concurrent analysis and simply replace a run of goroutine by a call instruction. This approach allows to detect some errors in the case of goroutines use, but not any errors specific for concurrent behaviour.

### *F. Analysis of `defer` instruction*

Our analysis models `defer` IR instruction behavior close to the corresponding Go `defer` statement semantics. It uses an additional stack for its support. While processing a `defer` instruction, the deferred function, its arguments and the related source code location are put onto this `defer`-related stack. The additional `defer` location is required to produce more user-friendly messages, since the reported deferred calls may be located far from their `defer` point, so it can be quite complicated to a user to understand why a function is called there. While visiting exit points of a procedure the analysis extracts deferred procedures and their arguments from the defer stack and processes the corresponding calls. The join operation at join CFG nodes tracks `defer`-related stacks as follows: first, the common bottom part of the incoming stacks is added to the result stack, then the rest of the tops of each incoming stack is added, while all the value duplicates are skipped.

Our initial approach to represent the deferred call argument values was to use the corresponding SSA variables. This solution is not sufficient in the cases when `defer` appears in a loop body. The problem is that the same SSA variable may correspond to different symbolic values at different loop analysis traversals. When the analysis unwinds defer stack in this approach, it uses just the last symbolic value for each argument, and these values are the same for different calls deferred by the same `defer` statement. This fact leads to a false positive for the code in listing 4. SVACE assigns different symbol values to variable `f` and puts a call to `f.Close` on the defer stack on different loop iterations. At the function exit point two calls of `f.Close` are extracted from the stack and processed. Since only one symbolic value is assigned to every symbol SVACE emits an error on double close of a resource.

```
func foo(files ...string) int {
    for _, fn := range files {
        f, err := os.Open(fn)

        if err != nil {
            return 0
        }

        defer f.Close() // false DOUBLE_CLOSE
    }

    return 1
}
```

Listing 4. False positive with `defer` in loop

To fix this issue we modeled the `defer` semantic more precisely. The improved version adds to the defer stack symbolic values for parameters as well. These values are used during defer stack unwinding.

The described implementation does not emit a false positive in the previous example and is able to find an error in the following example, where the same value is used:

```
func foo(files ...string) int {
    for _, fn := range files {
        f, err := os.Open(fn)

        if err != nil {
            return 0
        }
        f.Close()
        defer f.Close() // error DOUBLE_CLOSE
    }

    return 1
}
```

Listing 5. Error with `defer` in loop

Using closures in `defer` statements inside loops brings issues with the captured variables similar to those with parameter values: the values of captured variables may differ at different loop iterations. In order to support this case we put symbolic values of captured variables to the defer stack as well.

However even this implementation is not precise in all the cases, in particular because it does not collect path conditions for `defer` instructions. Listing 6 illustrates it. The defer stack at the end of the function will contain two elements, both are

for deferred close of the same channel, so this leads to the false positive report on double close `c`.

```
func bad(a int, c chan int) {
    if a < 0 {
        defer close(c) //false DOUBLE_CLOSE
    }

    if a > 0 {
        defer close(c)
    }
}
```

Listing 6. Unsupported case for `defer`

An obvious solution is to put the conditions on defer stack too and to process these conditions while extracting deferred calls from the stack. Nevertheless we have decided not to complicate the current implementation, since we haven't observed any related issues while validating our analysis on real-world projects. Moreover, our exploration of the Go projects source code shows that `defer` statements rarely occur under conditions and never twice with mutually exclusive conditions.

## IV. Other works

Paper [1] describes interprocedural taint analysis for Go based on the Monotone framework [11]. The authors consider most of Go language features including `defer` statement and channel operator, but the analysis is limited only to simple function calls. They also use an SSA-based representation. Unfortunately the paper does not contain any results on real projects including analysis quality and analysis time.

We found several linters for Go: STATICCHECK [13], GO-CRITIC [9], ERRCHECK [7], VET [8]. VET incorporates LOOP-CLOSURE, an AST-based detector, which checks whether an access to a loop variable is known to escape the current loop iteration in a call within `go` or `defer` statement. Our analyzer has a similar detector, which is designed to detect even more complex cases. , see [4]. VET is able to checks errors inside closure bodies, but it has no support for interprocedural errors.

## V. Results

In order to evaluate the described approach we used SVACE to analyze 8 open source projects and reviewed the produced warning reports. Table I shows build and analysis stage time[1] for these projects as well as the project sizes in lines of code (LOC).

We did not observe any change in project-specific metrics related to the closure usage on these projects. The improvements we introduced for closure support affected analysis time almost insignificantly. There are new true positive warnings, as well as false positive ones; the latter are caused not by the implementation of closure support itself, but by some inaccuracies in the checker algorithms. In other words the checkers are able to report true and false positive warnings in anonymous functions and closures, which are similar to those that these checkers were able to report within regular named functions previously.

Below is an example of a true positive issue detected by SVACE in the project `tidb` [14] at file `session/session.go`:

```
se := tmp.(*session)
...
defer func() {
    if !execOption.IgnoreWarning {
        if se != nil && se.GetSessionVars()
        .StmtCtx.WarningCount() > 0 {
            warnings := se.GetSessionVars()
                .SmtCtx.GetWarnings()
                s.GetSessionVars()
                .StmtCtx.AppendWarnings(warnings)
        }
    }
    /* error DEREF_AFTER_NULL */
    se.sessionVars.PertitionPruneMode.
    Store(prePruneMode)
}()
...
```

Listing 7. Error in `tidb`

Variable `se` is compared to `nil` and dereferenced after that without any appropriate `nil` check.

We have compared the results against the previous SVACE version, which did not have closure support [4]. Table II shows that on average the number of warnings has slightly increased (2.31%) and the number of analyzed functions increased on 5.45%.

Table I
EVALUATION OF BUILD AND ANALYSIS TIME

| Project (https://github.com/*) | LOC of the project (.go) | SVACE build time | SVENG analysis time (s) |
|---|---|---|---|
| pdfcpu/pdfcpu | 53076 | 20.4 | 67 |
| prometheus/prometheus | 109681 | 244 | 742 |
| ovh/cds | 208697 | 120 | 740 |
| etcd-io/etcd | 183098 | 100 | 130 |
| nanovms/ops | 24103 | 53 | 248 |
| pingcap/tidb | 555626 | 76 | 780 |
| percona/percona-server -mongodb-operator | 1144191 | 151 | 562 |
| jesseduffield/lazygit | 28714 | 52 | 73 |
| Average | | 91 | 378 |

## VI. Conclusion

We described a scalable interprocedural context- and flow-sensitive static analysis for the Go language, which tracks most of the language features and provides detectors for a variety of program error kinds. We provided the details of our approach to closures and `defer` statements support and analyzed its impact on the analysis results. The implementation we have developed is based on SVACE analysis framework and is able to analyze projects like `percona` [12] of 1144 KLOC in about 562 seconds.

[1]The environment we used: Ubuntu 20.04, RAM: 32 GB, CPU: Intel Core i7-7700 3.60GHZ.

Table II
THE NUMBER OF WARNINGS AND ANALYZED FUNCTIONS BEFORE AND AFTER THE CLOSURE SUPPORT

| Project (https://github.com/*) | Count of warnings before | Count of warnings after | Count of functions before | Count of functions after |
|---|---|---|---|---|
| pdfcpu/pdfcpu | 757 | 760 | 2826 | 3117 |
| prometheus/ prometheus | 9584 | 9705 | 80408 | 84142 |
| ovh/cds | 11521 | 11817 | 73887 | 77459 |
| etcd-io/etcd | 3421 | 3644 | 19994 | 21920 |
| nanovms/ops | 3477 | 3603 | 50734 | 53093 |
| pingcap/tidb | 10638 | 10918 | 61712 | 66172 |
| percona/percona-server -mongodb-operator | 8565 | 8633 | 43966 | 45580 |
| jesseduffield/lazygit | 1306 | 1320 | 7932 | 8593 |
| Average | 5540 | 5668 | 38326 | 40416 |

## REFERENCES


[1] Eric Bodden et al. “Information flow analysis for go”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 431–445.

[2] A.E. Borodin and I.A. Dudina. “Intraprocedural Analysis Based on Symbolic Execution for Bug Detection”. In: *Programming and Computer Software* 47.8 (2021), pp. 858–865.

[3] A.E. Borodin et al. “Searching for tainted vulnerabilities in static analysis tool Svace”. In: *Proceedings of the Institute for System Programming of the RAS* 33.1 (2021), pp. 7–32.

[4] Alexey Borodin et al. “Static analyzer for Go”. In: *2021 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE. 2021, pp. 17–25.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. “A static analyzer for finding dynamic programming errors”. In: *Software-Practice and Experience* 30.7 (2000).

[6] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[7] *errcheck main page*. https://github.com/kisielk/errcheck. Accessed: 2022-07-01.

[8] *Go vet main page*. https : / / golang . org / cmd / vet/. Accessed: 2022-07-01.

[9] *go-critic main page*. https://github.com/go-critic/go-critic. Accessed: 2022-07-11.

[10] V.P. Ivannikov et al. “Static analyzer Svace for finding defects in a source program code”. In: *Programming and Computer Software* 40.5 (2014), pp. 265–275.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. “Type and effect systems”. In: *Principles of Program Analysis*. Springer, 1999, pp. 283–363.

[12] *percona*. https://github.com/percona/percona-server-mongodb-operator. Accessed: 2022-10-01.

[13] *StaticCheck main page*. https://staticcheck.io. Accessed: 2022-07-10.

[14] *tidb*. https://github.com/pingcap/tidb/releases/tag/v6.4.0-alpha. Accessed: 2022-10-01.

[15] *TIOBE Index*. 2022. URL: https://www.tiobe.com/tiobe-index.