Static analyzer for Go

1st Alexey Borodin
ISP RAS
Moscow, Russia
alexey.borodin@ispras.ru

2nd Varvara Dvortsova
ISP RAS
Moscow, Russia
vvdvortsova@ispras.ru

3rd Sergey Vartanov
ISP RAS
Moscow, Russia
svartanov@ispras.ru

4th Alexander Volkov
ISP RAS
Moscow, Russia
volkov@ispras.ru

***Abstract*—This paper describes a static analysis tool for software defects detection in source code written in Go language. We developed a fast lightweight AST-based analyzer (GOA) to support detection of syntactic-level issues (linter) and a powerful interprocedural summary-based analyzer (SVENG) with its own intermediate representation.**

***Index Terms*—interprocedural static analysis, golang, symbolic execution, data flow analysis, path sensitive analysis**

I. Introduction

Go is an efficient and popular programming language. According to the TIOBE index [1], it is one of the top 20 most popular languages. However, the language is relatively young (the initial release dates 2012) and as far as we know there are still no tools based on deep interprocedural static analysis to detect common software error types in Go programs.

SVACE static analyzer was initially developed as a tool for automatic software defect detection in programs written in C/C++ and subsequently was extended to support Java and Kotlin. In this paper we present the new functionality, that has been recently introduced to SVACE to analyze programs written in Go. We will describe SVACE solely from the Go analysis perspective. All the specific features related to other languages support are beyond the scope of the present paper.

SVACE toolchain incorporates several tools and modules to intercept the original build of a software project, produce its models (representations) for the subsequent analysis and detect program defects of various kinds and levels of complexity.

SVACE combines two analysis frameworks (for the particular analyses and checkers implemented in SVACE):

- GOA[1] performs a lightweight analysis, which uses an abstract syntax tree (AST) as a representation of the source code under analysis and is intended for syntactic-level defect detection.
- SVENG[2] tool is the main part of SVACE. It performs a significantly more powerful and heavy control-flow and data flow based analyses with its own specific intermediate representation SVACE IR (we describe it in section IV) of the program under analysis; it performs not only intra-, but also interprocedural analysis.

In order to build AST and SVACE IR we use modified SSADUMP, a tool to produce a representation in single static assignment (SSA) form for a Go program. The original tool is a part of set of `golang.org/x/tools` packages. While parsing a program source code SSADUMP builds its native AST for Go and we use it in GOA. We patched SSADUMP SSA generation to produce the data for SVENG to build SVACE IR.

Fig. 1 gives an overview of SVACE analysis process. On the build stage SVACE BUILD CAPTURE tool runs an original build, intercepts the invocations of Go compiler and uses the captured information about these invocations to launch modified SSADUMP utility with GOA for each captured module. SSADUMP generates both AST and the input data for SVENG, passes the produced AST to GOA and stores the data for SVENG for further analysis. On the the subsequent analysis stage SVENG uses this data to build SVACE IR and he full call graph. SVACE history server imports the warnings produced by both analyzers and allows to browse them through the SVACE web interface.

More details on the specific changes were made to adapt SVACE for Go language can be found in [2]. The focus of this paper is the description of the full SVACE toolchain and the current state of the analyses implemented in SVACE and the way they work in the case of Go.

II. SVACE build process

As it was mentioned above, SVACE BUILD CAPTURE tool runs the original build, intercepts all the running processes and captures the important commands they execute (such as invocations of the Go compiler) and their arguments, since it is crucial to reproduce the original build structure for SVACE front-end. It is essential not to interfere with the original build process, so that its results are the same as of the original build process without any external interception. More information about the interception of the build process can be found in [3].

A compilation unit in the case of Go language is a package, not a single file and the original SSADUMP operates with the data at the package level. We preserved this approach in the modified version of it we use in SVACE.

For each Go package the modified SSADUMP produces a file, where it stores the data collected for the source code of this package. This data includes information about types, global variables, constants, and functions. The data for each

[1] short for **Go a**nalyzer
[2] short for **Tool Eng**ine

2767-9535/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPRAS53967.2021.00008

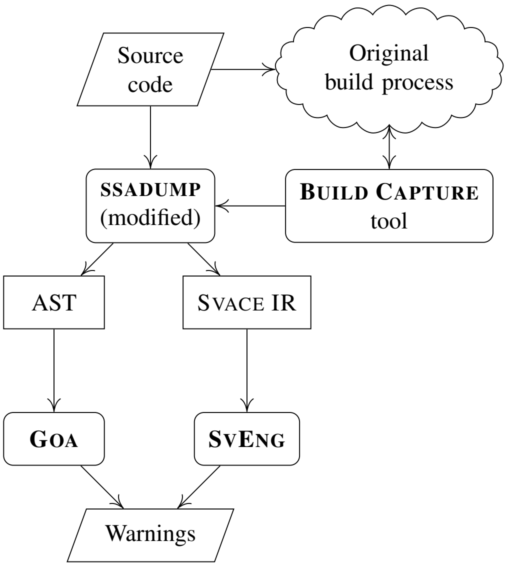

Fig. 1. Analysis diagram

function contains information on local types, symbol table and instructions, adapted from the original SSA representation.

In addition for each Go package the modified SSADUMP generates a file with the extracted information on function calls. It allows to speed up the construction of the full call graph for SvEng, required by the summary-based analysis.

We use JSON format for both these file kinds.

When Build Capture encounters a use of single compilation command for several packages, it identify package dependencies, filters this set, and runs modified version of SSADUMP for the resulting set of packages to produce intermediate representation. Svace Build Capture filters out the standard Go library and the previously built packages. It uses lock files to avoid processing package more than one time. These lock files are unique for package name and working directory name (last one is essential if `replace` directive is used).

An important performance issue is analysis of 3rd party code. Since 1.6 Go provides vendoring feature to manage dependencies. It uses `vendor` directories to collect 3rd party packages code. We provide an option to ignore the source code placed in them. Use of this option may significantly speed up Svace build process and the subsequent analysis. This however limits the depth and quality of the analysis and reduces the precision of its results (the reported warnings).

III. Goa

Goa AST analyzer was implemented on top of SSADUMP. Goa gets as input a list of AST representations (each AST) for each file of Go package. Next, detectors are launched in parallel over the AST and SSA representation. All AST detectors are intraprocedural.

Below we list some of the detectors we implemented in Goa:

- `UNSAFE_TYPE_ASSERTION` detects potential runtime errors, which can be trigger by type assertion expressions.
- `UNSAFE_TYPE_CONVERSION` detects possible integer overflow in type conversion operations.
- `LOOPVAR_IN_CLOSURE` detects the free variables captured in closures, which could be captured by reference. `go vet` [4] is able to detect a similar warning, but our version supports a more tricky case: it reports a warning for a method with pointer receiver, when it is called in a goroutine.

`LOOPVAR_IN_CLOSURE` detector performs a depth-first traversal of the analyzed AST.

1) When the detector reaches a loop, it begins to collect free variables of the loop
2) Finds a `go` or `defer` instruction node
3) There are two cases:
   a) if the called function is anonymous and how it captures the free variables (Listing 1.)
   b) if the called function is a method with pointer receiver (Listing 2.)

```
func(values []int) {
    var wait sync.WaitGroup
    wait.Add(len(values))
    for key, value := range values {
        go func() {
            /* LOOPVAR_IN_CLOSURE */
            fmt.Println(key, value)
            wait.Done()
        }()
    }
    wait.Wait()
}
```

Listing 1. The example for the anonymous function.

```
func (v *val) MyPtrMethod() {
    fmt.Println(v.String())
}

func test(values []val) {
    for _, val := range values {
        /* LOOPVAR_IN_CLOSURE */
        defer val.MyPtrMethod()
    }
}
```

Listing 2. The example for the method with pointer receiver.

IV. SvEng

SvEng provides a framework for deep interprocedural flow- and context-sensitive analysis and utilizes it in the detectors. In the case of Go analysis SvEng gets as input the files produced by modified SSADUMP and builds Svace IR—its own low-level intermediate representation format and it is the same for different languages. The latest Svace version supports C, C++, Java, Kotlin, and Go. More information on analysis for other languages can be found in [5–8].

SVACE IR supports the following types: integers, strings, floats, function pointers, structures, arrays, tuples. SVACE IR uses partial SSA form to represent a function. It has the following instruction classes: arithmetic, pointer handling (reading, writing, shifting), function calls by name or by pointer, assignments, control flow (goto and conditional goto), function manipulations (defer, make closure), specific built-in types support (channels, tuples, maps, slices).

SVENG analysis phases are the following:

1) Call graph construction
2) Preliminary phase
3) Main phase

First, SVENG reads a local call graph produced by modified SSADUMP for each analyzed Go package. The call graphs are read in parallel, since there is no need for any synchronization for that. Then the local call graphs are merged into a call graph for the whole project.

The merged call graph contains two types of vertices: function definitions and function calls. A function call is identified by the name of a called procedure. In the case of intra-package call the target (the callee) of a call can be defined easily.

Then SVENG builds the linked call graph. This graph contains only function definition vertices. The linking resolves function calls through their package-qualified function names.

Then a preliminary phase is run (IV-A). This phase is designed to be fast and it performs only simple intra-module (independent for each Go package) kinds of analysis. Its main goal is to collect information about global variables), array sizes, goroutines of each package for the subsequent phase.

Finally, SVENG runs the main phase of the analysis. It analyzes each procedure only once and creates a summary for it to support interprocedural analysis. The summary describes specific features of a procedure's behavior and the analysis will use it when it will track the calls to this procedure. The described summary-based approach has a good scalability, because there is no need to re-analyze the callee procedure for each call to it, since only its summary is used.

Section IV-B describes intraprocedural analysis. It is based on symbolic execution with state merging. In addition to it analysis of a particular function has data flow stage, which will be described at IV-D. Inter-procedural analysis consists of summary creation and applying it to caller context. Both operations will be described in section IV-C.

Our analysis is flow- and path-sensitive. It is sensitive to (is able to track) structure fields and array elements with constant indexes. SVENG performs global analysis for the whole program that means that calls to functions from other packages are also taken into account.

As far as we know only SVACE performs such full and deep analysis for Go programs.

A. Preliminary analysis

Preliminary analysis is intended to provide an additional information to the main phase, which can be collected through the simple traverse of all the program instructions.

Every module is analyzed independently in parallel. For every module SVENG analyzes global variables and then step by step every procedure in the module. Instructions of every procedure are analyzed in topological order. Every instruction is visited only once. We do not use any alias analysis at this phase.

Currently we implemented the following analyses:

- collect information on the call contexts for the main phase.
- Analysis for global constants. Here we collect information about global variables that assign only to the same constant values.
- Uninitialized global variables.
- Goroutines analysis. Collect function that are used as goroutines.
- Buffer size analysis. For buffers with constant size we collect information about its size and location where buffer is created.

SVENG is able to work without the preliminary phase, but it allows to get more information about the program at low cost[3].

B. Intraprocedural analysis

Procedure analysis during the main phase uses symbolic execution with state merging. Analysis gets control flow graph (CFG) of the function as an input. To simplify the analysis our CFG does not have any branching instructions. Instead of them it uses split nodes (vertex with one input edge and two output edges) and linear `assume` instructions at their outgoing edges. The semantics of it is that only the path where assume condition is hold is executed.

Abstract states are associated with the CFG edges. For each instruction, the analysis generates an abstract state associated with an output edge by using an abstract state associated with an input edge. For join points analysis generates the output state that describes all possible paths.

The loop analysis we use is not sound. During loop analysis several heuristics are used to model all the possible execution paths in strongly connected components (SCC) of CFG. It is possible that analysis incorrectly models properties for some paths if these heuristics work wrong. Several iterations of strongly connected SCC are performed. Current version uses two iteration for a loop body. After the end of loop analysis states for all iterations at output edges of SCC are merged.

All analyses and checkers are run simultaneously. It allows us to reduce both analysis time and memory usage. The analysis time is also reduced due to the fact that properties common to all checkers are analyzed only once. In most cases the state at the input edge of an instruction can be released immediately after its processing, this approach allows to significantly reduce memory consumption. The analysis does not need to store all states. It is enough to store current input state, the states at all input edges for not visited joins and out states for analyzing SCC.

[3] 5.1% of analysis time (VI)

SvEng uses an abstraction called "value identifier" to model the values stored in variables and other memory locations. We will refer to it as *value id* in this paper. Two variables are matched with one value identifier, if they have the same values during execution. Analysis builds *variable-to-value map* for every CFG edge to map variables to value ids.

SvEng models the properties of data it tracks through the so-called "attributes". Most of the attributes are associated with value ids. Attributes can also be described using value ids.

Associating attributes with value ids has advantage that in most cases attributes are not changed. For the following assign instruction:

```
a := b
```

Both variables have the same value id and and no attribute changes are required. Analysis needs only to update variable-to-value map.

Our alias analysis is also based on value ids. Its design is very simple. We use a heuristic that all the values of function input parameters are not aliases and, moreover, all the called functions return non-aliased results. Since it is usually true for the majority of the functions, it helps the analysis to produce precise alias results in most cases. Penalty for this is unsound results for the cases where these heuristics are not hold.

Value ids themselves have the following types:

- *Pre-values*—values for parameters, globals. Those values were created before function start.
- Constant value ids
- Join value ids—values that were created during path merging. They store information about values at every path.
- Shift value ids.
- Dereference value ids—results of dereferences. Those derefernced value ids are modeled as separate values stored in non-aliased original value ids.
- Internal value ids—other value ids that denote values created after function start.

SvEng runs alias analysis simultaneously with all its other analyses and checkers. Some of value ids are references (reference values). *Reference* is a value id for a pointer for which SvEng models memory. Current SvEng version models memory for pre-values and internal value ids and doesn't model for join value ids. For references SvEng makes strong updates. For other assignments through pointers SvEng makes only weak updates [4].

C. Inter-procedural analysis

We use a summary-based analysis, it implies implementations for the following two operations:

1) create summary—after an analysis of a function SvEng gets abstract state on the exit edge and uses it to create a summary
2) apply summary—for each call to a known function a corresponding summary is translated to the caller context

Summaries are designed to be small enough. That's why we use several thresholds to limit them. The main threshold is the number of value ids in a summary, this threshold in the current version is 250. For summary creation we use the following algorithm:

- Calculate a visible value ids set, add initially arguments and return value(s) to it.
- Find the values dependent on those that are in the visible set. Dependent value is a value that may be created from original value by dereference or shifting. Add those dependent values if result set size is less than the threshold.
- Repeat the algorithm until no new values are met.
- For all visible set run handle `annotate`. Every interprocedural checker must propagate values by implementing this handler. This handler is run for each value id and attribute pair. A checker gets an attribute value from the function exit state and must produce an attribute value in the summary.

Our analysis is context-sensitive. A summary for a function is applied while analyzing each call to it. If a function call is located in a loop, then a corresponding function summary will be applied at each analysis iteration.

While applying a summary at a function call point SvEng needs to translate the value ids used in the summary to the value ids of the caller context. It builds a value id multimap for that. This multimap is initialized with value ids for the formal and actual arguments of the called function. Then this map is filled for all the elements in summary using the dependency relation.

SvEng executes `apply` handler for all the corresponding value ids. Again, as in the case of `annotate`, it is the responsibility of every interprocedural checker to provide its specific implementation for this handler. Handler `apply` is called for every value id and attribute pair. Each checker has an attribute value of the state at the input edge of the processed call instruction and an attribute value from the summary and implements its specific algorithm to generate a new attribute to put it to the state at outgoing edge.

D. Data-flow analysis

Unreachable code detection was implemented as a part of conservative data-flow analysis of DFA stage [9]. SvEng runs this stage before the main analysis (symbolic execution) of each function. The key reason why it was implemented within this stage is the lack of accuracy of the main analysis. This analysis marks unreachable edges of the control flow graph. An unreachable edge dramatically affects all other analyses, therefore non-conservatism in this case would lead to significantly worse results.

In addition DFA analysis stage collects an auxiliary information for the symbolic execution stage (like inductive variables, live variables, loops with a constant iterations number).

DFA stage incorporates only a small number of analyses. It does not have alias analysis, it models only the variables

[4] Even for weak updates information may be recovered by use of definition graph, which will be described at IV-F

which addresses were not taken. The amount of the tracked and the collected data is much smaller than on the symbolic execution stage. As a result it is fast. In average DFA takes about 4.3% of total analysis time (VI).

Like the preliminary phase it is optional and may be removed.

E. Checkers

SvEng provide the detectors for the following classes of errors:

- `nil` pointer dereference
- integer overflow
- unreachable code
- buffer overflow
- division by zero
- improperly using data from external sources
- resource leaks

Most of these checkers work for other languages too. Svace core allows to implement source-sink checkers for variable values effectively. We describe a typical scheme for path-sensitive source-sink checker in section IV-F. Path-insensitive checkers can use binary, ternary attributes and interval attributes. Consider "and-boolean" attribute:

- for source point it is set up to `true`
- at join points result value is true if it is true for all input edges
- at sink point (which also may be modelled by other interprocedural attribute) checker emits warning if the value is true

In some cases source-sink scheme is not enough. It may be supplemented with information from preliminary phase (that current procedure is used as goroutine) or from DFA (about loop invariants, live variables).

F. Path-sensitive analysis

Path sensitivity of the analysis is based on value ids.

For every value id, SvEng stores its definition which contains information about how and where it was created. SvEng value id definition types are as the following:

- Constant definition—value is a constant,
- Expression definition—value is in the form of $a = b \bigotimes c$, where all a, b and c are value ids and $\bigotimes$ is operation (plus, minus, bit shift and etc.)

Those value id definitions form a value id definition graph for all created value ids.

Path-sensitive analysis uses definitions and conditional attributes.

Conditional attribute is a formula with trace information. Value ids are used as variables in those formulas. It has advantage that formulae remain the same even after assigning values to variables.

SvEng has special attribute $Ness$, which describes necessary conditions that some location is reachable. This attribute is filled by the following rules:

1) for path merging we use disjunction for attributes at input states
2) for assume-instruction we add conjuction with assume condition to value in input context

Below we describe a simple path-sensitive source-sink checker. It has a conditional attribute Src as an example. At a source instruction it sets attribute Src to value $True$ for all the corresponding value ids, which means that these properties for the modeled value are hold at this point.

Then attribute is changed only at path merging points using the rule:

$$Src_{out} = Ness_1 \wedge Src_1 \vee Ness_2 \wedge Src_2$$

At the sink location the checker creates a formulae $Src \wedge Ness$ for the error issue candidate and runs SMT-solver. If the solver returns `SAT` then this error is feasible and the checker emits a warnings.

Before adding formula to SMT solver SvEng collects all the used value ids and then for each used value id transitively finds definition. Then all the definitions are converted to a formulae and added to the resulting error formula as conjunctions. Use of definitions has an advantage that they do not depend on the current point and are true for the whole procedure.

Solver is run only if checker has a suspicion that an error is possible: source location is reached and variable has value id with non-default attribute Src. Solver is used only to suppress infeasible paths.

Checkers may use more than one conditional attribute: they may create attributes for sink too.

Below is a synthetic example of a `nil` dereference error:

```go
func foo(a int, p *int) {
    var isNil bool

    if a < 10 {
        if p != nil {
            isNil = false
        }
    }

    if a < 20 && !isNil {
        *p = 1 //error
    }
}
```

Listing 3. nil dereference

In this example an initialization of variable `isNil` is missing. Therefore the variable value is `false`. Then if pointer `p` is not `nil` than this variable is set up to `false`. value. Before dereferencing pointer `p` flag `isNil` is checked. But since it has only `nil` assignments it is always true. So if comparison for `p` is not redundant than `nil` pointer dereference is possible.

For the example, SvEng creates 6 value ids: 1 value ids for variables `a`, `p` and `isNil`, and 3 value id for location `*p`. Variables `a` and `p` do not have assignments. Variable `isNil` has 2 assignments to the same value. Because of that

SVACE generates only 1 value id for it. For location `*p` SVACE generates 3 values:

1) value before function start (”pre-value”)
2) constant `1`
3) join value id for the last join

Attribute **NilCond** is used to denote the values that were positively compared to `nil`. At `assume` instruction `p == nil` SVENG creates condition $true$ for this attribute for value id `p` (**NilCond**$(p) = true$). Path merging adds a conjunction with reachability attribute **Ness**. Reachability attribute denotes the conditions which are $true$, if the current location is reachable.

For the dereference of `p` SVENG checks attribute **NilCond**. Since it is not $false$, an error condition is created. In the current case it is $a < 10 \wedge p = 0 \wedge a < 20 \wedge isNil = \perp$. The resulting formula is passed to SMT-solver which establishes that the formula is satisfiable.

Path-insensitive checkers without conditional attributes also may use SMT-solver. First of all they may check attribute $Ness$ to filter out warnings at infeasible paths. Another option is to calculate error condition and check, if it is SAT at the current location. For example, for buffer overflow they may create condition that index is greater than a buffer size.

V. Other tools

We found only linter-like tools for static Go programs analysis: STATICCHECK [10], GO-CRITIC [11], ERRCHECK [12].

We found that famous analyzer COVERITY [13] supports Go, but we could not find any details.

Tool GOCATCH [14] performs the whole program analysis to find concurrency bugs. The tool has similar IR which is based on SSA package [15]. It has good false positive rate and can find many real bugs. It is hard to compare directly this tools with SVACE since scope of our warnings barely intersects. GOCATCH doesn’t have good scalability like SVACE. Analysis of big project Kubernetes takes 25.6 hours, while SVACE needs only 14 minutes.

VI. Results

For the estimation of SVACE results we used it to analyze 10 open-source projects. In total, SVACE detected 39066 warnings. A review of an unfamiliar source code and communications with its developers might take an excessive time, so we have chosen 25 bugs to describe in-depth, and reported them to the projects developers. 4 bugs have been fixed. 5 bugs have been rejected as minor. 1 bug was false positive.

Listing 1 demonstrates a previously unknown bug in PDFCPU. There is a pointer dereference after its comparison with `nil`. The variable `r` is used after a comparison with `nil`, and it means that if `r` is equal to `nil`, it will trigger panic.

```
func getR(d Dict) (int, error) {
    r := d.IntEntry("R")
    // comparison with nil
    if r == nil || *r < 2 || *r > 5 {
        if *r > 5 { // dereferenced
```

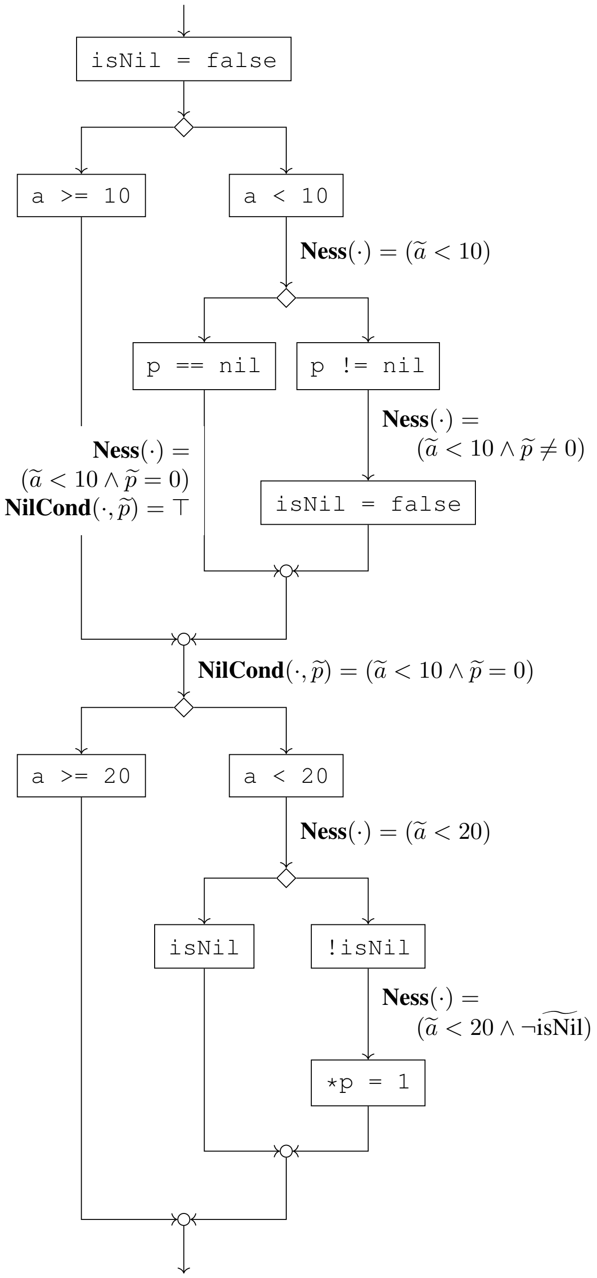
isNil = false
a >= 10
a < 10
Ness$(\cdot) = (\widetilde{a} < 10)$
p == nil
p != nil
Ness$(\cdot) =$ $(\widetilde{a} < 10 \wedge \widetilde{p} \neq 0)$
Ness$(\cdot) =$ $(\widetilde{a} < 10 \wedge \widetilde{p} = 0)$ **NilCond**$(\cdot, \widetilde{p}) = \top$
isNil = false
NilCond$(\cdot, \widetilde{p}) = (\widetilde{a} < 10 \wedge \widetilde{p} = 0)$
a >= 20
a < 20
Ness$(\cdot) = (\widetilde{a} < 20)$
isNil
!isNil
Ness$(\cdot) =$ $(\widetilde{a} < 20 \wedge \widetilde{\neg \text{isNil}})$
*p = 1

Fig. 2. Example control-flow graph

```
            return 0, errors.New("pdfcpu:
            PDF 2.0 encryption not supported")
        }
        return 0, errors.New("pdfcpu:
        encryption: \"R\" must be 2,3,4,5")
    }
    return *r, nil
}
```

Listing 4. The previously unknown PDFCPU bug.

PDFCPU team's patch is shown in Listing 2.

```
func getR(d Dict) (int, error) {
    r := d.IntEntry("R")
    if r == nil || *r < 2 || *r > 5 {
        if r != nil && *r > 5 {
            return 0, errors.New("pdfcpu: PDF
            2.0 encryption not supported")
        }
        return 0, errors.New("pdfcpu:
        encryption: \"R\" must be 2,3,4,5")
    }
    return *r, nil
}
```

Listing 5. A patch of pdfcpu bug.

Table I contains the data on build and analysis time for 10 open-source projects, their total source code size 1477 KLOC. Table II contains the data for these projects, including the project sizes in the lines of code and the size of the generated IR. In average SVACE build and analysis last 15 times longer than an original build.

Table III contains the data about different phases of SVENG analysis. Columns "Total function analysis time" and "DFA time" contain data about analysis time for all functions with thread summing. Other columns contain real time. DFA stage takes about 4.3% of function analysis time in average. Preliminary phase takes 5.1% of total analysis time. Most of this time is spent to reading IR.

Tables IV, V, and VI contain data for project KUBERNETES. It is the biggest open-source Go project we have found. The size of the project is nearly 2 million source lines and more than 3.7 million lines with its dependencies. SVACE build and analysis take 7.5 and 14.5 minutes correspondingly. An ability to analyze such big projects in a reasonable time proves a good scalability of our tool.

VII. Conclusion

We have described a way to implement static analyzers for the Go language. Our approach demonstrates good scalability and allows us to detect real errors in big Go projects.

The tool we have developed combines two types of analyzers and a build interception utility. Together they are able to detect a wide variety of software defects, ranging from typos and code-style problems to source-sink errors (like `nil` pointer dereference, division by zero) and more complex erroneous dataflow patterns, such as buffer overflows and resource leaks.

The tool has been tested on open-source Go projects and has demonstrated the ability to efficiently detect errors in an acceptable amount of time.

References

[1] *TIOBE Index*. 2021. URL: https://www.tiobe.com/tiobe-index.

[2] I.V. Bolotnikov and A.E. Borodin. "Interprocedural Static Analysis for Finding Bugs in Go Programs". In: *Programming and Computer Software* 47.5 (2021), pp. 344–352.

[3] A.A. Belevancev, A.O. Izbyshev, and D.M. Zhurikhin. "Monitoring program builds for Svace static analyzer". In: *System administrator* 7-8 (2017), pp. 135–139.

[4] *Go vet main page*. https://golang.org/cmd/vet/. Accessed: 2020-10-9.

[5] 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.

[6] A. Belevantsev et al. "Design and Development of Svace Static Analyzers". In: *In 2018 Ivannikov Memorial Workshop (IVMEM)* (2018), pp. 3–9.

[7] 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.

[8] A.E. Borodin and I.A. Dudina. "Symbolic execution based intra-procedural analysis for search for defects". In: *Proceedings of the Institute for System Programming of the RAS* 32.6 (2020), pp. 87–100.

[9] Mulyukov R.R. and Borodin A.E. "Using unreachable code analysis in static analysis tool for finding defects in source code". In: *Proceedings of the Institute for System Programming of the RAS* 28.5 (2016).

[10] *StaticCheck main page*. https://staticcheck.io. Accessed: 2021-09-01.

[11] *go-critic main page*. https://github.com/go-critic/go-critic. Accessed: 2021-09-01.

[12] *errcheck main page*. https://github.com/kisielk/errcheck. Accessed: 2021-09-01.

[13] *Coverity 2021.03: Supported Platforms*. 2021. URL: https://sig-docs.synopsys.com/polaris/topics/r_coverity-compatible-platforms_2021.03.html.

[14] Ziheng Liu et al. "Automatically detecting and fixing concurrency bugs in go software systems". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 616–629.

[15] *Package SSA*. https://godoc.org/golang.org/x/tools/go/ssa. Accessed: 2021-09-01.

Appendix A
Table of results

The data is for Ubuntu 20.04, RAM: 32 GB, CPU: Intel Core i7-7700 3.60GHZ.

TABLE I
Evaluation of build and analysis time of 10 projects

Project (https://github.com/*)	Original build time (s)	Svace build time with GOA (s)	Svace build time without GOA (s)	SvEng analysis time (s)
taskctl/taskctl	0.467	9.150	6.109	53.917
pdfcpu/pdfcpu	1.611	13.115	9.371	54.389
prometheus/prometheus	0.619	183.019	143.550	496.143
ovh/cds	57.591	236.837	165.837	572
etcd-io/etcd	22.177	99.512	78.148	164.286
nanovms/ops	0.482	219.287	185.887	185.213
pingcap/tidb	90.240	256.510	215.930	606.740
minio/minio-go	2.457	8.757	6.652	64.204
percona/percona-server-mongodb-operator	1.671	132.708	90.284	454.454
jesseduffield/lazygit	5.616	17.229	16.502	81
Average	18.293	117.612	91.827	270.789

TABLE II
Evaluation of sizes of 10 projects

Project (https://github.com/*)	Size of the project without vendor (LOC)	Size of the generated IR (MB)
taskctl/taskctl	4621	73
pdfcpu/pdfcpu	53076	65
prometheus/prometheus	109681	873
ovh/cds	208697	1126
etcd-io/etcd	183098	284
nanovms/ops	24103	541
pingcap/tidb	555626	2867
minio/minio-go	28973	59
percona/percona-server-mongodb-operator	15023	2764
jesseduffield/lazygit	294705	103

TABLE III
Time about different phases of analysis in SvEng

Project (https://github.com/*)	Call graph building time (ms)	Preliminary phase time (ms)	Main phase time (ms)	DFA time (ms)	Total functions analysis time (ms)
taskctl/taskctl	395	1787	46795	12508	336774
pdfcpu/pdfcpu	397	3109	47435	17180	344566
percona/percona-server-mongodb-operator	1787	20938	409832	97537	3091962
ovh/cds	2675	21748	538028	159094	4123749
prometheus/prometheus	2579	18005	469273	119428	3519939
etcd-io/etcd	1186	7449	151237	49282	1092232
nanovms/ops	1991	13191	164581	80513	1120748
pingcap/tidb	2493	46644	551050	221955	4183761
minio/minio-go	345	2494	58166	23398	265072
jesseduffield/lazygit	695	3541	61499	24625	556091
Average	1454	13891	249393	80552	1863489

Appendix B
Kubernetes

The data is for Ubuntu 20.04, RAM: 32 GB, CPU: Intel Core i7-7700 3.60GHZ.

TABLE IV
EVALUATION OF BUILD AND ANALYSIS TIME OF KUBERNETES

Project	Kubrnetes
Original build time (s)	54.684
SVACE build time with GOA (s)	455.436
SVACE build time without GOA (s)	429.435
SVENG analysis time (s)	870.127
Size of the project (LOC)	1 954 270
Size of the project with dependencies (LOC)	3 726 378
Size of the generated IR (MB)	3276

TABLE V
TIME ABOUT DIFFERENT PHASES OF ANALYSIS IN SVENG OF KUBERNETES

Project	Kubernetes
Call graph building time (s)	4.695
Preliminary phase time (s)	3.412
Main phase time (s)	845.127
DFA time (s)	252.659
Total functions analyzing time (s)	6733.555

TABLE VI
EVALUATION OF BUILD AND ANALYSIS TIME OF KUBERNETES WITH AND WITHOUT VENDOR

Project	Without vendor	With vendor
SVACE build time (s)	240	455.436
SVACE analysis time (s)	103	870.127
Count of warnings	1521	12425
Analysis		disappeared 2 FP disappeared 4 TP disappeared 6 WF