ISSN 0361-7688, Programming and Computer Software, 2021, Vol. 47, No. 5, pp. 344–352. © Pleiades Publishing, Ltd., 2021.
Russian Text © The Author(s), 2021, published in Programmirovanie, 2021, Vol. 47, No. 5.

Interprocedural Static Analysis for Finding Bugs in Go Programs

I. V. Bolotnikov[a,b,*] and A. E. Borodin[a,]**

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 119333 Russia*

[b] *Moscow State University, Moscow, 119991 Russia*

**e-mail: igor.bolotnikov@ispras.ru*

***e-mail: alexey.borodin@ispras.ru*

Received April 12, 2021; revised April 21, 2021; accepted May 11, 2021

Abstract—In recent years, the popularity of the Go programming language has been growing. However, currently, there are only lightweight static analyzers (linters) available for Go. We fill this gap by adapting the Svace static analyzer for Go programs. We implement an interprocedural and intermodular static analyzer that possesses both flow sensitivity and path sensitivity. To evaluate its performance, we use ten open source projects. The sixteen evaluated checkers emitted 6817 warnings with 76% true positive rate.

DOI: 10.1134/S0361768821050030

1. INTRODUCTION

Go is a compiled, strongly typed, multithreaded programming language created by Google in 2009. It is used mostly in the backend of web applications [1], which did not prevent it from getting into the top 20 of most popular programming languages at the time of writing this paper [2, 3].

The developers of this language tried to protect it, to the maximum extent possible, from common errors made by programmers. The language does not support implicit typecasting. All variables are initialized to zero by default, buffer overflow does not lead to vulnerabilities, and garbage collector is implemented to prevent the majority of memory leak cases.

The Go compiler searches for common trivial errors, e.g., declaration of an unused variable. An important advantage of Go is its high compilation speed because, when designing a compiler, the build speed and quality of optimizations, rather than bug detection, come first. Hence, it still requires additional static analysis.

In order not to overload the compiler, the language developers implemented an open static analyzer called *go vet*, the intermediate representation of which is the abstract syntax tree (AST) of the Go language. At the time of writing this paper, it supported warnings of 21 types. Below are some of them:

- copylocks: locks erroneously passed by value;
- nilfunc: useless comparisons against nil;
- printf: inconsistency of format strings and arguments;
- unusedresult: unused results of calls.

There are also several similar analyzers (linters): staticcheck [4], go-critic [5], and errcheck [6].

In this paper, we propose a static analyzer for Go that supports deep interprocedural semantic analysis. All Go analyzers mentioned above do not possess this property.

To solve this problem, we extend the capabilities of Svace, a static analyzer developed at the Ivannikov Institute for System Programming of the Russian Academy of Sciences [7–10]. This analyzer was originally created for C/C++ programs and then was extended to programs in Java [11] and Kotlin.

2. SVACE ANALYZER

This analyzer is designed to detect as many bugs as possible with an acceptable level of false positives. The interprocedural and intermodular analysis is summary-based and context-sensitive. The analysis takes into account aliases and values of variables, as well as models the contents of fields in structures and elements of arrays. The implemented checkers are aimed at detection of errors, including null pointer dereference, array overflow, resource leaks, infinite loops, and unsafe use of external data.

Below is a typical scheme of analysis.

1. As input, the analyzer receives source code and a build script.

2. A special build-capture component intercepts compilation commands.

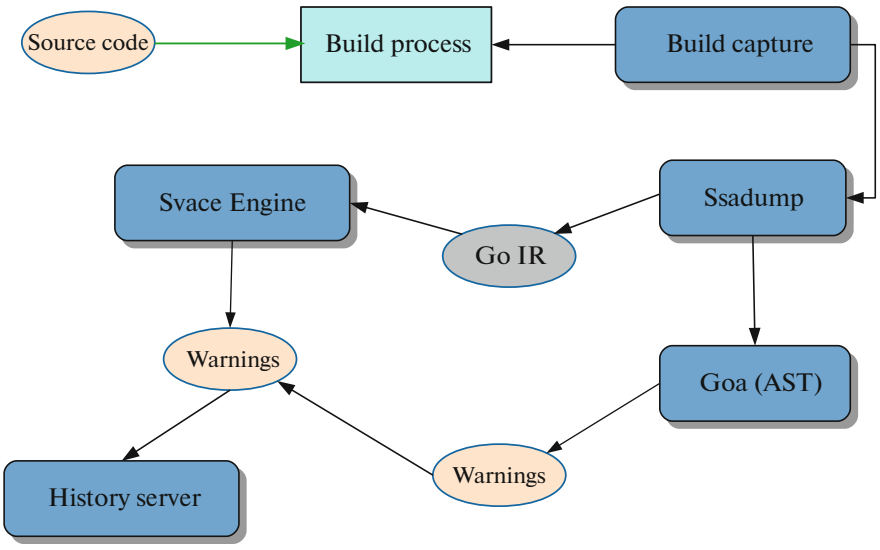

Fig. 1. Analysis scheme.

3. The modified compiler constructs an AST, which is input to the AST analyzer.

4. The modified compiler also generates an intermediate representation of the program for subsequent analysis.

5. The intermediate representation is fed to the SvEng analyzer (Svace Engine is the main Svace analyzer).

To make the analyzer capable of processing Go programs, the following steps were carried out.

• The build capture procedure was modified by extending the build-capture component. Only cases for direct compilation by Go compiler calls were implemented.

• A Go intermediate representation, quite similar to the intermediate representation of the analyzer, was selected. Its generator and parser were implemented. The intermediate representation of the analyzer was extended.

• In the framework of SvEng, some unique analyses for Go were written.

Figure 1 shows the analysis scheme developed for Go. Generally, to generate an intermediate representation, a compiler is used. For Go, we managed to avoid modifying the compiler, which is not only a complex procedure, but it also implies expensive support. Instead of the compiler, the ssadump utility was used, which is described in Section 3.2. This utility generates the intermediate representation and also runs the AST analyzer called *goa*. The SvEng utility carries out interprocedural analysis for the generated representation.

3. INTERMEDIATE REPRESENTATION

3.1. Svace IR

Svace IR is the intermediate representation of the SvEng parser, which allows one to analyze programs in different programming languages with the same analyzer. Svace IR has the following specific features: it is similar to the intermediate representation of LLVM and it is in the partial SSA form. Its more detailed description can be found in [12, pp. 32–41].

3.2. Go Intermediate Representation for Svace

As an intermediate representation, we chose the SSA representation of the ssadump tool, which is open source and included with the main auxiliary tools for Go in the set of packages `golang.org/x/tools`. This choice was due to the similarity of this intermediate representation to Svace IR. As the format for IR generation, we chose JSON due to the ease of its implementation, support, and (most importantly) debugging and readability. This format is not optimal in terms of generation/reading rate and memory consumption; if necessary, it can be changed.

The JSON IR generator was implemented in ssadump. The parser was implemented on the side of the analyzer. Instruction mapping from one intermediate representation to another is carried out in two steps. The instructions that have their counterparts in Svace IR are mapped directly. These are basic operations implemented in the majority of languages, e.g., logical operators, arithmetic operators, calls, jumps, etc. However, the intermediate representation of ssadump includes language-specific instructions:

• defer: implements deferred function calls;

• go: runs a goroutine with a specified function;

• typeassert: implements typecasting in a separate instruction with two variants:

– strict typecasting: the instruction returns a single result if successful, otherwise causing an execution error,

– nonstrict typecasting: the instruction returns two values (the first value is the typecasting result, while the second value indicates whether the typecasting was successful); in the case of a failure, the first value can be any garbage value;

• extract: extracts an element of a tuple;

• select and send: instructions for working with Go channels;

• makeClosure: creates a lambda object, based on a specified function and enumerated variables from the environment of a call point;

• calls of built-in functions that have special status in Go (append, len, copy, etc.);

• make*: creates various objects of built-in non-basic types;

• mapUpdate: inserts a pair into map;

• lookup: accesses an element of a string and map with key check;

• range and next: implement integration over Go collections (map, slice, array, string, and channel).

All ssadump instructions listed above were added as Svace IR instructions or as specifications (builtin and make*) (see Section 5.2).

The set of Svace IR types was also extended:

• tuple: this type implicitly occurs in Go when a function returns several values at once that can have different types;

• slice: built-in type for dynamic arrays;

• map: built-in type for mapping;

• chan: Go channel for communication between goroutines (it is most similar to pipe in C);

• Go interface: type that defines a set of functions to be implemented by another type; it does not require explicit description of *implements*; duck typing.

4. BUILD CAPTURE

The capture process executes the original build command while instrumenting it in such a way as to intercept all running processes and capture the desired build commands. In this case, these are calls of the Go compiler; however, the process can be extended to other build tools. In this case, it is important not to affect the original build: the results must coincide with the build carried out without external interference. A detailed description of the build capture process can be found in [13].

If, during the build, it turns out that a command for compiling some set of packages has been executed, then a script is run in parallel; this script finds dependency packages, filters the resulting collection, and runs a modified version of ssadump for the remaining packages, which yields information about the intermediate representation of the remaining packages. The filter has several settings. The packages for which the intermediate representation has already been built are discarded first. For this purpose, before running ssadump, a lock file is created in the file system for each package. It is unique for the next parameter set: the name of a package and the name of a working directory are created; the latter is required to take into account commands like replace in the case of using Go modules.

An additional filtering is carried out when Go vendoring is used, which is a compilation mode where some dependencies are fixed. This code is stored separately from the rest of the source code, it is rarely updated and is not a user-defined code. In this case, the build tool is supplemented with an option that allows one to ignore dependencies in *vendor*, which speeds up the build and analysis of the remaining code; however, this limits the depth of the analysis, which can affect the quality of warnings.

The unit of analysis is a package (rather than a file) because ssadump originally operates with data at the package level. All information is collected for the package, and there is no need for its additional splitting into files. In the case of the AST for Go, the tree is constructed for each file individually and, in its original form, it contains incomplete information even about types of used variables. In other analyzers, this problem is solved by a similar auxiliary analysis on a set of ASTs associated with the same package. For the same reasons, *goa* also analyzes an AST with additional information about packages (rather than a pure AST); the process of collecting this information does not reduce performance because it is used to generate the intermediate representation for the Svace analyzer.

As a result, depending on arguments, ssadump generates the following data for each package.

File with the intermediate representation. A JSON file with a type table, symbol table, and intermediate representation of functions for one package. It is required to construct Svace IR.

File with the call graph. It is required to construct the call graph in Svace. For the other programming languages, the call graph is constructed by standard reading of IR files. For Go, the intermediate representation was designed from scratch. With the analyzer implementing the call graph reading phase, a separate file with the call graph is created for optimization purposes.

Files that describe the location of declarations and uses of variables, types, and functions in CSV format, which are required for more convenient interaction

between the user and the web interface of issued warnings.

File with *goa* warnings. The analysis of the AST requires a certain amount of information collected during the build. Hence, it is more reasonable (in terms of time) to perform it immediately after the build in the same process because *goa* is implemented in the same language (Go) as ssadump.

5. EXTENSION OF THE ANALYZER

5.1. Brief Description

The SvEng tool uses summary-based interprocedural analysis. First, the call graph is constructed; then, the functions are traversed in such a way that the callee functions are analyzed before the caller functions. The loops in the call graph are forcedly broken. Once the call graph is constructed, a preliminary flow-insensitive phase is carried out, during which information about callee functions and constants used is collected.

Once the function is analyzed, its summary is created, which is then used to analyze function calls. For each function, an analysis based on symbolic execution with union of states at path merge points is carried out. The analysis models values of variables, structure fields, arrays elements, and memory cells that can be reached by dereferences and calls. To describe properties, attributes shared among different checkers are used. Currently, there are more than 350 attributes implemented.

To enable path sensitivity, formulas that describe error conditions are constructed; when the checker can issue a warning, an SMT solver is run to check the satisfiability of the formula.

5.2. Specifications

To model library functions, Svace uses specifications. A specification describes the behavior of a function: it is another definition of the function in a given language. Specifications can contain calls of special functions that are completely undefined but have special semantics for the analyzer. Specifications are analyzed in the same way as all other functions; as a result, a summary is created, which is then used to analyze function calls. Svace includes specifications for popular libraries; user specifications can also be added.

We wrote over 170 specifications for the following packages: bytes, crypto, database, encoding, errors, flag, fmt, io, log, os, strconv, strings, and zap.

Go has functions built in the compiler, e.g., *len* and *make*. For them, we created a file builtin.go, where one can write specifications for predefined functions. Thus, the behavior of these functions is not hard-coded in the analyzer; instead, it is defined in the configurable part.

Below is an example of a specification

```
func makemap(size interface{}) inter-
face{} {
    Sf_set_trusted_sink_int(size)

    var res interface{}
    Sf_overwrite(&res)
    return res
}

func len(v interface{}) int {
    var res int
    Sf_overwrite(&res)
    Sf_assert_cond(res, ">=", 0)

    if res != 0 {
       Sf_assert_cond_ptr(v, "!=", nil)
  }

  Sf_pure(res, v)
  return res
}
```

Here, special function *Sf_set_trusted_sink_int* tells the analyzer that the *size* parameter must not accept data from non-trusted sources. Special function *Sf_pure* means that the result of function *len* depends only on its argument. Special function *Sf_assert_cond* indicates that the return value is not negative.

5.3. Modeling Go-Specific Instructions

Go has the `defer` instruction that pushes a function call into a stack and executes functions from this stack when a caller function terminates. Go IR also includes the `defer` instruction; however, function calls themselves are more interesting for analysis. All necessary logic was implemented in a plugin, which memorizes arguments of `defer` and, for all memorized arguments, initiates processing of the function call statement at the exit points of a function. Thus, a representation is emulated where all defer calls are removed and instructions for calling the corresponding functions are added at the function exit points.

Go facilitates multithreaded programming. The goroutine allows function calls to be executed concurrently. Due to the built-in support, this instruction is widely used. Svace cannot analyze functions executed in parallel. Hence, we decided to simply flag the call executed in parallel and then analyze it as a common function call. This calling behavior of the goroutine is

permissible; however, it does not exhaust all the possibilities.

In Go, functions can return multiple values by using tuples. We extended Svace IR so that all functions can return multiple values. In addition, it was required to modify a lot of handlers for return values and summaries. A new data type—tuple—was introduced; values of tuples are modeled by the existing structured type with the corresponding fields.

5.4. Existing Checkers

For Go, we included some checkers designed for other languages. We did not include all checkers while focusing only on their certain subset. In most cases, they did not require adaptation to Go code and intermediate representation.

Below is the list of the included checkers.

• DEREF_AFTER_NULL: inconsistent work with pointers; the code contains null pointer check and dereference without null check.

• DEREF_AFTER_NULL.EX: an improved version of DEREF_AFTER_NULL that uses an SMT solver.

• DEREF_OF_NULL.RET.EX: a function returns null that is then dereferenced.

• OVERFLOW_UNDER_CHECK: access to an array by index for which there is a range check that does not exclude array overflow.

• TAINTED_INT: use of tainted data from external sources in critical operations.

• DIVISION_BY_ZERO.EX: "divide by zero" error; the source is either a null constant or a null comparison instruction.

• DIVISION_BY_ZERO.UNDER_CHECK: division by a figure the values of which were checked and zero value is not excluded.

• REDUNDANT_COMPARISON.ALWAYS_FALSE: truth check of an a priori false condition in a conditional statement.

Below is an example of DEREF_AFTER_NULL for the etcd project (etcd/embed/serve.go).

```
func (ac *accessController)
ServeHTTP(rw http.ResponseWriter, req *http.Request) {

        if req != nil && req.URL != nil
            && strings.HasPrefix(req.URL.Path, "/v3beta/") {
                req.URL.Path = strings.Replace(req.URL.Path,
                           "/v3beta/", "/v3/", 1)

        }

        if req.TLS == nil {
```

In the code, the *req* variable is compared with zero; then, the pointer is dereferenced when accessing the *req.TLS* field. Perhaps, *& &* should be replaced with ||. For this project, an interprocedural error involving three functions was found.

The DEREF_OF_NULL.RET.EX checker required significant modification mainly due to the use of tuples in Go, which are involved in error handling. A typical pattern is shown below.

```
func create(arg int) (*MyStruct, error) {
   if arg >= 0 {
       s := createImpl(arg)
       return s, nil
   }

   return nil, errors.New("Negative argument")
}
```

Thus, the pointer is null only if *error* is not null. We added values of other tuple elements to the error condition. In addition, a special procedure for handling the error part of the tuple was implemented. An attribute that has a reference to a potentially null pointer was created. When checking the error part by using this attribute, the information about the nullness of the pointer is deleted.

The array overflow checkers required modeling of the predefined function *len*, which was carried out using specifications (5.2).

5.5. Go-Specific Checkers

We implemented several checkers to find suspicious patterns in the source code that can be useful to Go programmers:

• UNCHECKED_TUPLE.RET: the error part of the tuple is not checked;

• UNCHECKED_TUPLE.CHAN: a version of UNCHECKED_TUPLE.RET for reading from a channel;

• DEREF_OF_NULL.GLOBAL: finds the situations of using a global variable that is completely uninitialized and, therefore, has zero value;

• INFINITE_LOOP.GOROUTINE: the use of a function with an infinite loop as a goroutine;

• DEREF_OF_NULL.RET.GO_INTERFACE: an incorrect null check of an interface.

In Go, tuples are often used to handle errors. The UNCHECKED_TUPLE.RET checker issues warnings in the cases where the error part of functions is ignored:

```
func parse(par string) (*Res, error) {
    if par == "" {
       return nil, fmt.Errorf("…")
    }
    return getRes(par), nil
}

func use(para string) {
    res, _ := parse(para)
    //error part is ignored
    res.handle() //error
}
```

It should be noted that this checker is interprocedural and path-sensitive. Hence, adding an incorrect check does not suppress the warning.

We additionally confined UNCHECKED_TUPLE.CHAN to the cases where reading occurs in a loop and a channel is transferred to the goroutine where it is closed. This pattern is dangerous because, if the channel is closed, then the loop becomes infinite.

The DEREF_OF_NULL.GLOBAL checker is flow-insensitive. The warning is issued only if a global variable is completely uninitialized. It is implemented at the preliminary analysis phase, which follows the construction of the call graph and precedes the main phase. The preliminary phase successively analyzes all instructions of all modules.

To implement INFINITE_LOOP.GOROUTINE, the existing infinite loop checker was extended. The preliminary phase collects information about the functions executed as goroutine. If infinite loops are found in these functions, then INFINITE_LOOP.GOROUTINE issues a warning.

In Go, the interface contains information about the value and type of a pointer. Null check of the interface does not imply that the captured value is not null and can be dereferenced. This can lead to unobvious errors:

```
var data *byte
var in interface{}

fmt.Println(data,data == nil)
//prints: <nil> true

fmt.Println(in,in == nil)
//prints: <nil> true

in = data
fmt.Println(in,in == nil)
//prints: <nil> false
```

For the last *Println*, the value of variable *in* is not zero, even though a null pointer is assigned to it.

To prevent such errors, we implemented two versions of DEREF_OF_NULL.RET.GO_INTERFACE. The first version issues warnings if the pointer that can be null is returned from a function of the "interface" type. The second version checks whether this returned value is then dereferenced. Both these checkers issue warnings only for return values of functions. We consider the return value pattern to be the most dangerous because, as a result of refactoring, the type of the return value can change from "pointer" to "interface."

6. AST-BASED ANALYSIS

We did not intend to replicate existing analyzers. However, any static analyzer only benefits from AST-based checkers. These checkers generally have a high true positive rate and can find a lot of typos.

Based on ssadump, the static analyzer called *goa*[1] was implemented. This analyzer is available in two versions:

• as part of ssadump: in this case, immediately after the generation of the intermediate representation, all collected information is transferred to *goa*;

• standalone: *goa* independently tries to collect the information provided by ssadump in the first version; in the case of a failure, only the checkers that work directly with the AST are run.

Goa implements warnings of the following types:

• INVARIANT_RESULT: warning about an expression the result of which is known at the compilation stage and can be replaced by a constant;

[1] Abbreviated form of Go analyzer.

• UNSAFE_TYPE_CONVERSION: warning about arithmetic operations that permit a safer explicit typecasting as compared to the current one, which makes it possible to avoid type overflow;

• UNSAFE_TYPE_SWITCH: warning about the absence of the default branch in a type-switch expression; it is desirable to explicitly declare default, even if it is empty, thus clearly stating that the function must not adapt to new implementations (if any) of this interface;

• UNSAFE_TYPE_ASSERTION: warning about the possibility of a runtime error in the typeassert instruction;

• LOOPVAR_IN_CLOSURE: warning about a reference—within a goroutine nested in a loop—to a variable that may not be constant on different iterations of this loop. In *go vet*, there is a similar warning. However, our version has a broader approach: in addition to analyzing the last instructions of the loop, warnings are issued for goroutine methods with objects passed by pointers.

The INVARIANT_RESULT checker performs the depth-first traversal of AST vertices and, for each vertex, tries to successively apply certain rules, the list of which can be quite large. For instance, a rule can check all vertices represented by binary expressions $\mathbf{a} \ll \mathbf{b}$. It is definitely known at the compilation stage if the size of type **a** is less than the value of operand **b**.

The UNSAFE_TYPE_CONVERSION checker traverses all integer type cast expressions. If, within the cast, there is an arithmetic expression over types of lower bitsize, then, during the arithmetic operation, type overflow is possible, which could not occur if type extension were carried out before the operation. Let us consider the following example.

```
func example(slice []byte) {
    length := len(slice)
    for {
        if int(slice[0])+1 > length {
        return
        }
        /* UNSAFE_TYPE_CONVERSION */
        length -= int(slice[0] + 1)
        if length == 0 {
           break
        }
        slice = slice[slice[0]+1:]
    }
}
```

If the value of slice[0] at the entry of the loop is 255 and len(slice) is greater than or equal to 256, then the loop is infinite because expression int(slice[0] + 1) is always zero due to type overflow.

The LOOPVAR_IN_CLOSURE checker performs the depth-first traversal of the AST. When encountering a loop, it starts collecting the variables (variants) the values of which depend on the iterators of the loop. When encountering a go or defer node, it checks whether a callee function is anonymous and whether it captures any external values that are variants of the current or outer loop. If it is a method for the "pointer" type (rather than an anonymous function), then it is also checked whether the object for which this method is called is a variant.

Below is an example of the former case

```
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()
}
```

An example of the latter case is as follows

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

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

7. RESULTS

To evaluate the results, we used ten open source projects. Table 1 shows the following data for these projects: size in lines of code (LOC), number of files, LOC size including dependencies, number of files including dependencies, and size of the intermediate representation generated.

Table 2 contains the build time and analysis time data for each project. The second column shows the build capture time with running *goa*, while the third column shows only the build capture time. The fourth column contains the main analysis (SvEng) time, while the fifth column contains the original build time. The instrumented build process slows down the main build. In the worst case, the build time is increased by a factor of 4.24 for the cds project. On average, the analysis time is longer than the build time. The maximum analysis time with respect to the original build time was for the unioffice project (27.5). On average, the build capture is 3.16 times slower than the original

Table 1. Parameters of the projects

Project (https://github.com/*)	Project LOC	Number of files	LOC with dependencies	Number of files with dependencies	Size of the IR generated (MB)
anacrolix/dht	3647	51	241540	531	25
taskctl/taskctl	4492	59	262344	562	54
unidoc/unioffice	9438	48	9438	48	289
quasilyte/go-ruleguard	11136	120	27362	203	20
percona/percona-server-mongodb-operator	12805	121	1080637	2923	889
nanovms/ops	17416	156	789421	2127	788
jesseduffield/lazygit	22714	153	248406	975	133
pdfcpu/pdfcpu	48703	186	58169	212	83
prometheus/prometheus	96971	331	1219480	4104	1066
ovh/cds	199302	1323	1286867	5846	1533

Table 2. Build and analysis times[2]

Project (https://github.com/*)	Build time with *goa* (s)	Build time without *goa* (s)	Main analysis time (s)	Original build time (s)
anacrolix/dht	26.077	21.469	16.335	13.097
taskctl/taskctl	18.398	17.987	61.524	4.432
unidoc/unioffice	38.718	33.815	329.054	11.959
quasilyte/go-ruleguard	10.139	9.745	15.675	6.477
percona/percona-server-mongodb-operator	207.338	197.267	393.428	68.578
nanovms/ops	160.655	159.805	172.399	47.506
jesseduffield/lazygit	33.314	30.287	74.635	13.753
pdfcpu/pdfcpu	15.73	14.615	50.001	5.838
prometheus/prometheus	255.099	240.48	404.897	75.85
ovh/cds	280.615	268.112	365.773	66.142
всего	1031.926	777.150	1883.721	245.367

[2]The experiments were carried out on Ubuntu 20.04, 32 Gb RAM, Intel Core i7-7700 3.60GH.

build, the capture with *goa* is 4.2 times slower, and the analysis is 7.67 times slower. The capture with the analysis is 11.88 times slower than the original build. We consider this an acceptable price for the opportunity of finding non-trivial errors.

For the projects under analysis, 6817 warnings were issued by 16 checkers. For each type, we manually labeled at least 20 warnings to assess the performance of the checkers. The results are shown in Table 3. On average, the true positive rate was 76%. This rate is quite high; however, we would like to improve it in the next version of the analyzer, as well as increase the number of covered errors.

The main source of false positives was the lack of specifications for library functions, especially for those that return tuples with dependent data.

For six warnings, the corresponding error reports were sent to the project developers. At the time of writing, three responses were received:

1. golang.org/x/text: array overrun (issue 42147), fixed;

2. github.com/pdfcpu/pdfcpu: null pointer dereference (issue 303), fixed;

3. github.com/minio/minio-go: redundant nil pointer comparison in one code segment or erroneous nil dereference in another segment (issue 1457), rejected as insignificant.

8. CONCLUSIONS

As far as we know, our static analysis tool for Go has no analogues. It is capable of finding interprocedural and intermodular defects with the average true

Table 3. Performance of the checkers[3]

Checker	Total	TP rate
DEREF_AFTER_NULL	34	70%
DEREF_AFTER_NULL.EX	141	20%
DEREF_OF_NULL.RET.EX	1212	50%
DIVISION_BY_ZERO.EX	12	45%
DIVISION_BY_ZERO.UNDER_CHECK	7	100%
UNSAFE_TYPE_ASSERTION	1286	75%
LOOPVAR_IN_CLOSURE	11	90.9%
UNSAFE_SWITCH	1915	75%
UNSAFE_TYPE_CONVERSION	90	80%
UNSAFE_TYPE_SWITCH	269	100%
INFINITE_LOOP.GOROUTINE	10	100%
INVARIANT_RESULT	14	85.7%
OVERFLOW_UNDER_CHECK	3	100%
REDUNDANT_COMPARISON.ALWAYS_FALSE	34	85%
TAINTED_INT	3	100%
UNCHECKED_TUPLE.RET	1776	40%

[3]For evaluating, we randomly selected 20 warnings (or evaluated all warnings if fewer were issued).

positive rate of 76%. Six error reports were sent to the developers of the corresponding open source projects. The responses to three of them were received, and two errors were corrected.

Any criticism and suggestions for improving the analyzer will be welcome.

REFERENCES

1. Golang-2019 survey. https://blog.golang.org/survey2019-results. Accessed October 3, 2020.
2. PYPL. http://pypl.github.io/PYPL. Accessed October 3, 2020.
3. IEEE Spectrum's programming languages top. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019. Accessed October 3, 2020.
4. StaticCheck main page. https://staticcheck.io. Accessed October 10, 2020.
5. go-critic main page. https://github.com/go-critic/go-critic. Accessed October 10, 2020.
6. errcheck main page. https://github.com/kisielk/errcheck. Accessed October 10, 2020.
7. Borodin, A.E. and Dudina, I.A., Intra-procedural analysis for error detection based on symbolic execution, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2020, vol. 32, no. 6, pp. 87–100. https://doi.org/10.15514/ISPRAS-2020-32(6)-7
8. Borodin, A.E. and Belevantsev, A.A., Svace static analyzer as a collection of analyzers of different levels of complexity, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2015, vol. 27, no. 2, pp. 111–134. https://doi.org/10.15514/ISPRAS-2015-27(6)-8
9. Belevantsev, A., et al., Design and development of Svace static analyzers, *Proc. Ivannikov Memorial Workshop* (*IVMEM*), 2018, pp. 3–9.
10. Ivannikov, V.P., Static analyzer Svace for finding defects in a source program code, *Program. Comput. Software,* 2014, vol. 40, no. 5, pp. 265–275.
11. Merkulov, A.P., Polyakov, S.A., and Belevantsev, A.A., Analyzing Java programs in the Svace tool, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2017, vol. 29, no. 3.
12. Borodin, A.E., Inter-procedural context-sensitive static analysis for error detection in the source code of programs in C and C++, *Cand. Sci. (Phys.-Math.) Dissertation,* Moscow: Inst. Syst. Program. Russ. Acad. Sci., 2016.
13. Belevantsev, A.A., Izbyshev, A.O., and Zhurikhin, D.M., Organization of controlled build in the Svace static analyzer, *Sist. Administrator,* 2017, nos. 7–8, pp. 135–139.

Translated by Yu. Kornienko