Go-Sanitizer: Bug-Oriented Assertion Generation for Golang

Cong Wang*, Hao Sun*, Yiwen Xu*, Yu Jiang*, Huafeng Zhang†, Ming Gu*
* *School of Software, Tsinghua University, China*
† *Software Quality Assurance Group, Huawei Technologies, China*

***Abstract*—Go programming language (Golang) is widely used, and the security issue becomes increasingly important because of its extensive applications. Most existing validation techniques, such as fuzz testing and unit testing, mainly focus on crashes detection and coverage improvements. However, it is challenging for test engines to perceive common program bugs such as loss of precision and integer overflow.**

In this paper, we propose *Go-Sanitizer*, an effective bug-oriented assertion generator for Golang, which is able to achieve a better performance in finding program bugs. Firstly, we manually analyze the Common Weakness Enumeration (CWE) and summarize the applicabilities on Golang. Secondly, we design a generator to automatically insert several bug-oriented assertions to the proper locations of the target program. Finally, we can utilize the traditional validation techniques such as fuzz and unit testing to test the programs with inserted assertions, and *Go-Sanitizer* reports bugs via the failures of assertions. For evaluation, we apply *Go-Sanitizer* to Badger, a widely-used database software, and successfully discovers 12 previously unreported program bugs, which can not be detected by pure fuzzer such as Go-Fuzz or unit testing methods.

***Index Terms*—Golang, program bug, assertion generation**

I. Introduction

The Go programming language (Golang) is widely used in different areas, such as micro-services for mobile applications [1], [2], blockchain systems [3], [4], etc. The security of Golang becomes increasingly important because bugs in those applications would result in failure of the whole system or even lead to huge loss of money.

Software testing methods, such as fuzz testing and unit testing, are regarded as effective techniques to find bugs in programs [5], [6]. Fuzz testing is to automatically generate different inputs to trigger the system crashes, and there are many fuzzers such as Go-Fuzz playing important roles in practical application [7]. Unit testing is also applied to ensure the program correctness with predefined test oracles and mainly focuses on the coverage improvements.

Those techniques work well in their context, but cannot deal with the program bugs [8] such as precision loss of float numbers and integer overflow. For example, as the function "main" presented in Listing 1, we use an equal comparison (Line.10) to decide whether to go inside the statements between Line.11-13. However, this comparison is based on two float numbers "a" and "b". Due to the precision loss, the two variables are misunderstood as unequal ones. Obviously, the behavior of the program can not match the expectations. Neither fuzz testing nor unit testing could detect this issue because the program would not crash during fuzzing or trigger oracle violation during unit testing.

```
1  // file: prec_loss.go
2  func main() {
3      x := 74.96
4      y := 20.48
5      b := x - y
6      log.Println(b) // 54.47999999999999
7      a := 54.48
8
9      log.Println(a == b) // false
10     if (a == b) {
11         // Intend to do something.
12         // Unfortunately, the branch can not be
13         // executed, due to precition loss issues
14     }
15 }
```

Listing 1: Bug of Precision Loss.

To fill this gap, we aim to design an approach to generate bug-oriented assertions for Golang. Let p denotes a user-input Golang package. Then our task is solved by working out $asserts(p) = \{assert_1, ..., assert_n\}$, which is a set of bug-oriented assertions. A bug-oriented assertion is a program assertion statement specially designed for bug detection (especially those can not trigger crashes). Once an assertion fails, it means that there could be a corresponding bug at the location of programs. However, assertion generation is challenging from two major perspectives:

- *Bug Categories.* Golang has its own syntax and semantics. Previous program bug or weakness classifications, such as Common Weakness Enumeration (CWE) [8], are not fully applicable.
- *Automatic Generation.* Identifying the assertion pattern and the location for the corresponding assertion are hard work, and it is time-consuming and error-prone to accomplish this task manually.

In this paper, we propose *Go-Sanitizer*, a bug-oriented assertion generator for Golang, which could be able to achieve a better performance in finding program bugs when accompanied with the traditional validation methods. For the above two challenges, firstly, we analyze program bugs from CWE and select the ones which are applicable in Golang. We further design different assertion patterns for these applicable bug types. Secondly, we implement a user-friendly assertion generator to generate bug-oriented assertions, including the pattern initialization and the location identification. *Go-Sanitizer* cur-

rently supports 9 types of program bugs. Finally, we utilize traditional validation techniques such as fuzz and unit testing techniques to validate the programs with assertions.

For evaluation, we apply *Go-Sanitizer* on Badger[9], a database software widely-used in industrial applications, such as PlayNet, Coyote, etc. *Go-Sanitizer* successfully discovers 12 unreported bugs in Badger's code base, which can hardly be discovered by pure fuzz or unit testing methods. All these bugs have been confirmed as real problems with manual validation. The main contributions of this paper are:

- We propose a bug-oriented assertion generator for Golang, named *Go-Sanitizer* [1], which are helpful to detect program bugs.
- We present a series of Golang bug categories and the corresponding test oracle related assertion patterns.
- We apply *Go-Sanitizer* to real-world application, and discover many previous unknown bugs [2].

The rest of this paper is organized as follows. Section.II presents the related work. Section.III elaborates on our approach of assertion generation, including the bug category and pattern definition, assertion initialization and insertion. Section.IV presents the experimental results. Section.V makes a conclusion of our work.

II. Related Work

Software Testing: In practice, software testing such as fuzz testing and unit testing has been widely used for security and quality assurance. Fuzz testing has been regarded as an effective technique to detect vulnerabilities of programs. American Fuzzy Lop (AFL) [10], considered as a leading fuzzing tool, has found a number of vulnerabilities in programs based on its coverage-guided method. To further improve the performance, researchers have proposed several optimizations based on symbolic execution and taint analysis [11], [12]. As for Golang fuzzers, Go-Fuzz [7] is a representative tool, and is effective in finding bugs, especially in a number of Golang's official packages. Unit testing is another common method to check the correctness of programs. For example, Golang has its unique grammar rules for unit testing programs.

Assertion Generation: For assertion generation, Wang et al. propose assertion recommendation for C programming language [13], [14]. They detect program weaknesses via well-defined assertions. Yamaguchi et al. propose an effective approach to discover missing checks in source code [15]. They statically scan source code to accurately identify real missing checks which are security-critical.

Main Differences: *Go-Sanitizer* mainly focuses on program bug categories which do not trigger execution crashes. To the best of our knowledge, *Go-Sanitizer* is the first bug-oriented assertion generator for Golang, and can be applied to improve the performance of traditional software testing such as fuzz testing and unit testing.

[1]We have made the tool open source. The source code of *Go-Sanitizer* can be accessed by https://github.com/wangcong15/Go-Sanitizer

[2]We also share the project inserted with assertions, which can be accessed by http://congwang92.cn/badger-with-asserts.zip

III. Go-Sanitizer Design

A. Overall Framework

Fig. 1 shows the overall framework of *Go-Sanitizer*. It takes the Golang source code as input. Firstly, in "Assertion Recommendation" phase, we do syntax check and static scan on the code. Then we utilize bug-oriented assertion patterns to identify the abstract syntax tree (AST) of the code. These assertion patterns are designed based on characteristics of program bugs. The bugs do not trigger code crashes but are able to be detected by specific assertions (like "precision loss" described above). Then our assertion generator works out a list of assertions, containing information of inserted positions, assertion expressions and bug categories. Secondly, in "Candidate Pick" phase, users can read assertions in detail (file, location, assertion expression, and reason). They can make decisions on which assertions to reserve before inerting them into programs. Finally, in "Validation Integration" phase, we firstly execute unit testing. Then we integrate Go-Fuzz, a state-of-the-art fuzz testing tool for Golang. We record the situation when an inserted assertion fails. Depending on testing results, *Go-Sanitizer* infers a bug report.

```
1  // file: regexp_without_anchors.go
2  func getPathFromInput() string {
3      // suppose we input "../../etc/passwd"
4      return "../../etc/passwd"
5  }
6
7  func main() {
8      pat := "/[A-Za-z0-9]+/"
9      u, _ := user.Current()
10     home := u.HomeDir // home="/home/root"
11     p := getPathFromInput()
12     if flag, _ := regexp.MatchString(pat, p);
           ↪ flag == true {
13         // attackers can bypass the check to read
               ↪ security messages
14         filePath := path.Join(home, p)
15         if b, err := ioutil.ReadFile(filePath);
               ↪ err == nil {
16             fmt.Println(string(b))
17         }
18     }
19 }
```

Listing 2: A bug of regular expression without anchors.

Let us take the example in Listing. 2 to illustrate the whole procedure. The code snippet may suffer from a security problem because of regular expression without anchors. In detail, the check of regular expression (Line.12) on user-provided strings is not strict enough. When attackers feed "../../etc/passwd" as input (Line.4), the function actually exposes security-critical data. The program bug, in this case, corresponds to "Regular Expression without Anchors" (CWE-777). *Go-Sanitizer* can find the situation and advise users to insert an assertion to check that the spliced file path (Line.14) is not outside expectations. Then we use Go-Fuzz to test the "main" function, and successfully trigger an assertion failure. As a result, the security-critical bug in this code snippet will appear in *Go-Sanitizer*'s report.

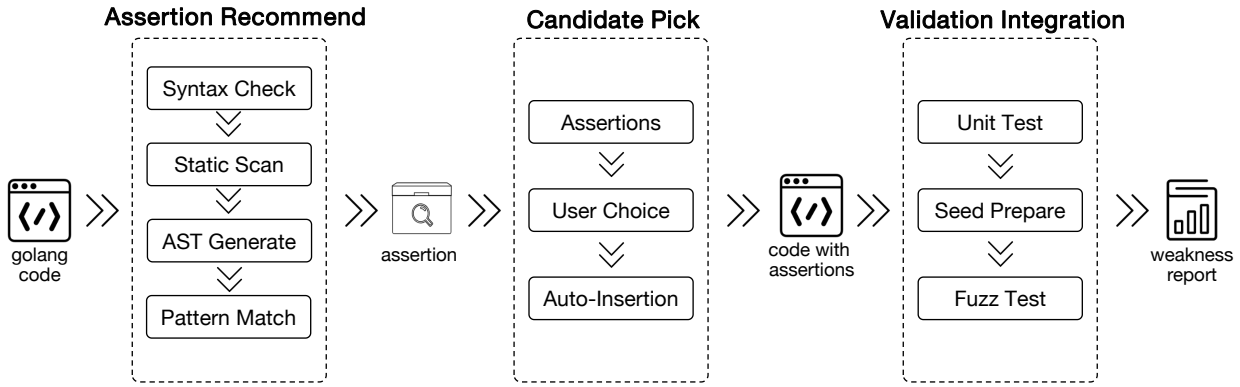

Fig. 1: Overall Framework, including assertion recommend, candidate pick and validation integration.

ID	Weakness Name	Category	Assertion Template
128	Wrap-around Error	Incorrect Calculation	AssertValEq(A, B)
190	Integer Overflow	Incorrect Calculation	AssertOverflow(A, B, A+B)
191	Integer Underflow	Incorrect Calculation	AssertUnderflow(A, B, A-B)
785	Use of Path Manipulation Function without Maximum-sized Buffer	Incorrect Access	AssertGte(len(A), len(B))
466	Return of Pointer Value outside Expected Range	Incorrect Access	AssertNNil(A)
824	Access of Uninitialized Pointer	Incorrect Access	AssertNNil(A)
478	Missing Default Case in Switch Statement	Incorrect Comparison	AssertIn(A, [B, C, ...])
1077	Floating Point Comparison Incorrect Operator	Incorrect Comparison	AssertPresion(A, B)
777	Regular Expression without Anchors	Incorrect Comparison	AssertStrNotIn(A, Blacklist)

Fig. 2: CWEs' Applicability in Golang.

B. Bug-oriented Assertion Generation

Golang has its particular restricts and rules in underlying design. Previous program bug classifications, such as Common Weakness Enumeration (CWE) [8], are not fully applicable to Golang. Thus, we analyze the program bugs and design the corresponding assertion patterns.

As shown in Fig. 2, we choose 9 bug types. To better illustrate the applicabilities of these bugs in Golang, we have built a test-suite to facilitate understanding, which can be accessed online [3].

Moreover, we design patterns to generate assertions. For example, "Regular Expression without Anchors" mentioned above, is CWE-777 bug. Notes that Column "Assertion Template" is the template to format the expressions of assertions. An assertion calls a checking function to validate the conditions. For example, the assertion of CWE-824 is to ensure that the pointer "A" is not nil (Golang uses "nil" instead of "null"). Besides, Golang does not have the grammar of assert statements, thus we also design an assertion library [4] to check all these assertions.

C. Bug-oriented Assertion Insertion

Go-Sanitizer applies a constraint-based approach to match bug types with source code and identify where to insert the assertions. It is worth mentioning that a constraint is a condition for equality matching on an abstract syntax tree. The definitions of our constraints for each bug type are:

- *scope=ast.type*: scope under sub-trees of a specific type.
- *save node.name if node.type=ast.type*: to find a node of a specific type, and record the variable name.
- *node1.name=node2.name*: validate equality of names.
- ..., etc.

Besides, we design a structure of assertions for each bug type. The structure includes file, location, expression, and reason. After preparing these, we present our insertion algorithm in Algo. 1. The algorithm takes a Golang package and the constraints of bugs as input. The algorithm checks the constraints of each bug type (Line.5-10). For those match all the constraints, we record the information (Line.11-12) and append it into the list of assertion candidates (Line.13). Finally, this algorithm works out a list of assertion candidates.

D. Valiadation Integration

After we work out bug-oriented assertions, users can read assertions in detail. To validate these assertions, we integrate traditional testing methods. Go-Fuzz [7] is a state-of-the-art fuzz testing tool for Golang. It tests Golang projects dynamically guided by coverage. We modify Go-Fuzz to record the situation when our inserted assertion fails. Meanwhile, we also execute the unit testing scripts to validate these inserted assertions as well. Depending on testing results, *Go-Sanitizer* infers a bug report.

[3]https://github.com/wangcong15/cwe-testsuite-golang
[4]https://github.com/wangcong15/goassert.

Algorithm 1 Assertion Generation

Input:
A golang package, p
Constraints and assertion templates of weakness, $list_w$
Output:
List of $assertion$, $list_{assert}$

$list_{assert} \Leftarrow$ a empty list
$list_{src} \Leftarrow$ GetSourceFiles(p)
for each $file \in list_{src}$ **do**
  $root \Leftarrow$ root node of $file$'s AST
  for each $w \in list_w$ **do**
    for each $c \in w.constraints$ **do**
      if MatchFails($root$, c) **then**
        Break, Goto STEP 5
      end if
    end for
    $loc, expr \Leftarrow$ GetInfo(w)
    $assert \Leftarrow \{file, loc, expr, w.reason\}$
    $list_{assert} \Leftarrow list_{assert} \cup assert$
  end for
end for
return $list_{assert}$

IV. Evaluation

During the experiment, we answer the following questions.

Q1: Is the approach effective in finding bugs: We should validate that our approach can discover real problems, especially those problems missed by traditional testing methods.

Q2: How is the accuracy of these assertions: A good approach is not only effective, but also accurate. Thus, we need to evaluate the false positive rate of bug detection.

Q3: How much burden do assertions bring in: Although assertions are simple code statements, they still bring burden (extra checks). Thus, we need to evaluate its burden.

A. Experiment Setup

Platform: All experiments are conducted on a Macbook Pro (Intel Core i5 2.7 GHz, 8 GB 1867 MHz DDR3), and the Golang version is 1.12.4 darwin/amd64.

ID	Package	Path	KloC	File
1	badger	.	12.019	39
2	y	y	1.008	13
3	pb	pb	2.024	1
4	skl	skl	0.865	3
5	trie	trie	0.108	2
6	table	table	1.536	5
7	options	options	0.007	1
8	main	badger	0.022	1
9	cmd	badger/cmd	1.419	9
10	main	integration/testgc	0.195	1
Total	-	-	19.203	75

TABLE I: Information of Badger's Source Code

Data Sets: We choose Badger [9] to conduct experiments. Badger is an efficient and persistent key-value store. As a fast database software, Badger is widely used in industrial applications. Table. I shows the information of Badger's source code (commit ID: c32e701). Column "KloC" means a thousand lines of code. Golang projects are organized by "packages", so Column "Package" means the name of code package. For example, the first package "badger" is located in the root path of the project, which contains 12,019 lines of code (Blank lines and comments are not included). Then, we do experiments on these code packages to discover program bugs, and solve the three industrial questions above.

Data Collection: For Q1, we utilize *Go-Sanitizer* to generate assertions and test the code with Go-Fuzz and unit testing scripts. Then we collect crash logs, which are triggered by our bug-oriented assertions. For Q2, we manually check the syntax correctness and review the code to validate whether a crash is a false positive. Notes that a false positive means a situation when crashes do not occur on a bug. For Q3, we repeat running Badger's unit testing code to compare the time consumption. Finally, we can use the excess time to evaluate the burden brought by our assertions.

B. Experiment Results

Q1: Is the approach effective in finding bugs: Firstly, we present the data of assertions. We use *Go-Sanitizer* to insert assertions into the source code of Badger. As shown in Fig. 3, we generate 48 assert statements in total. For example, we insert 24 assertions in the package with ID 1, which is package "badger" in the root path. Among these packages, P5, P7, and P8 are free of assertions. Code amounts are small in these packages, and *Go-Sanitizer* does not match any patterns to insert bug-oriented assertions.

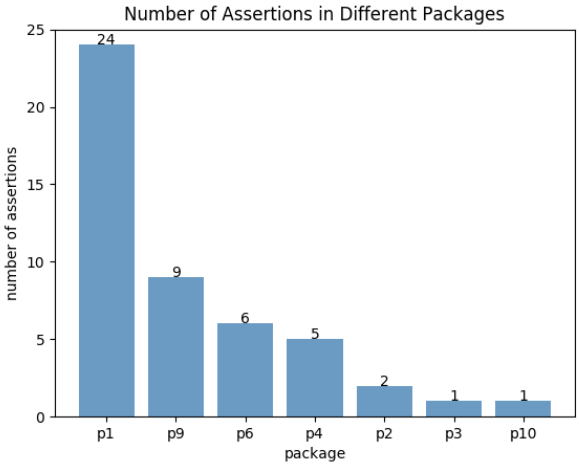

Fig. 3: Asserts: 48 assertions are inserted in total.

In addition, Fig. 4 shows the number of assertions for each bug type. C190 (Integer Overflow or Wraparound) has the maximum number of assert statements, which means, we have instrumented assertions into 11 spots to check integer overflow

ID	PID	CWE	File	Line	Assertion	#Fuzz	#Unit	#Go-Sanitizer
1	9	191	badger/cmd/bank.go	228	AssertUnderflow(highTs, lowTs, highTs-lowTs)	×	×	✓
2	9	190	badger/cmd/bank.go	243	AssertOverflow(lowTs, highTs, lowTs+highTs)	×	×	✓
3	6	478	table/iterator.go	78	AssertIntIn(whence, []int{origin, current})	×	×	✓
4	6	478	table/iterator.go	279	AssertIntIn(whence, []int{origin, current})	×	×	✓
5	6	190	table/table.go	205	AssertOverflow(off, sz, off+sz)	×	×	✓
6	6	191	skl/arena.go	67	AssertUnderflow(maxHeight, height, maxHeight -height)	×	×	✓
7	6	191	skl/arena.go	78	AssertUnderflow(n, l, n-l)	×	×	✓
8	6	191	skl/arena.go	93	AssertUnderflow(n, l, n-l)	×	×	✓
9	6	191	skl/arena.go	106	AssertUnderflow(n, l, n-l)	×	×	✓
10	3	190	pb.pb.go	828	AssertOverflow(offset, 1, offset+1)	×	×	✓
11	1	128	value.go	1125	AssertValEq(n, headerBufSize)	×	×	✓
12	1	785	iterator.go	186	AssertGte(len(key), len(badgerMove))	×	×	✓
*	4	128	skl/skl.go	124	AssertValEq(valOffset, value)	×	×	✓

TABLE II: Report: Information of Detected Weaknesses.

issues in this experiment. CWE-824 (Access of Uninitialized Pointer) bug is not detected because there is not a suspicious code snippet matching the pattern.

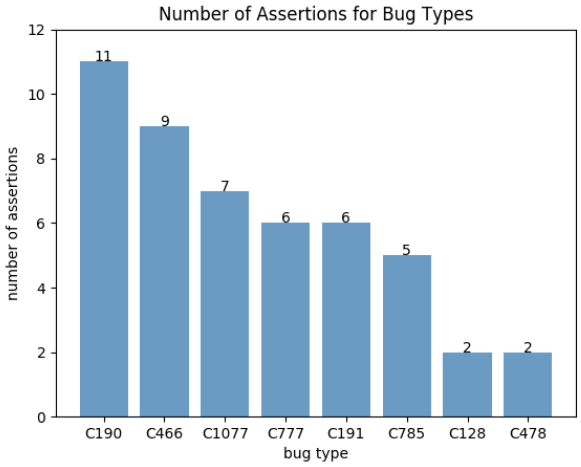

Fig. 4: Assertion Numbers of Bugs.

Table. II shows the information on detected bugs. Column "PID" corresponds to the package ID in Table. I. Column "Line" is the location of assertion in source files. Column "Assertion" presents the assertion expressions which are inserted into programs. For example, the first bug locates in Line.228 of file "badger/cmd/bank.go", which is an integer underflow issue, because the calculation "highTs-lowTs" does not detect the underflow problem. The last three columns show whether the case can be detected by fuzz testing, unit testing or *Go-Sanitizer*. All theses bug cases can hardly be found by pure testing methods, because they do not raise execution crashes.

As shown in the table, *Go-Sanitizer* has successfully discovered 12 previous unknown bugs in Badger. Notes that the last row (marked with an asterisk) is a case of false positive. We will explain it in Q2. Among these bugs, we have 5 underflow issues, 3 overflow issues, 2 missing defaults issues, a wrap-around issue, and a missing length check issue. Five types of bugs are discovered, while Badger is free of the other four types. Objectively speaking, Badger's code is very elegant. They have a lot of security checks in programs, in order to avoid some common bugs. In addition to the experiment on Badger, the effect of detecting bugs can be proven on the test-suite [16]. This test-suite is written as simple Golang functions, each of which corresponds to a bug type. *Go-Sanitizer* is capable to generate bug-oriented assertion on the test-suite. **From the experiment results above, we can conclude that *Go-Sanitizer* is effective in discovering program bugs, especially those happen without signals of crash.**

Q2: How is the accuracy of these assertions: We analyze the accuracy of bug-oriented assertion from two aspects: 1) whether to introduce syntax problems after inserting assertions, 2) whether to introduce false positives in testing.

In our experiments, code with assertions can complete compilation normally. Thus, the code is also syntax-correct with assertions. On the other hand, there is only one false positive case. Listing. 3 shows this code snippet. In Line.5, we generate an assertion to check wrap-around issues for "value". Through testing, *Go-Sanitizer* reports it as a bug case. However, it is considered as correct code by human validation, because this code meets programmers' need to finish decoding tasks. Compared with detecting program bugs, it is a more difficult goal to guess programmers' thoughts. Except for this, the other 12 cases are real bugs through manual validation. **Thus, *Go-Sanitizer* has a good accuracy rate in assertion generation and insertion.**

```
1 // file: skl/skl.go
2 func decodeValue(value uint64) (valOffset uint32,
      ↪ valSize uint16) {
3     valOffset = uint32(value)
4     valSize = uint16(value >> 32)
5     goassert.AssertValEq(valOffset, value)
6     return
7 }
```

Listing 3: Case of False Positive

Q3: How much burden do assertions bring in: Table. III shows the time consumption of Badger's integral unit testing. Through comparing the source code with assertion-instrumented code, the latter has only 0.36% more time

consumption on average. Theoretically, assertions need more calculations in code execution. **Thus, the burden is tolerable.**

Round	Source Code	Code with Asserts	Burden
1	214.779s	215.644s	+0.40%
2	206.378s	207.269s	+0.43%
3	217.341s	216.953s	-0.18%
4	215.606s	216.553s	+0.44%
5	209.387s	210.857s	+0.70%
Average	212.70	213.46	**+0.36%**

TABLE III: Time Consumption of Badger's unit testing

Case Study: To demonstrate the experiment results more detail, we pick an example for illustration. Listing. 4 shows the second bug case of Table. II. This function uses the binary search principle to find invalid data. However, variable "highTs" and "lowTs" are external parameters, leading to the potential possibility of integer overflow in Line.5. Once an overflow happens, variable "midTs" can be an unqualified value. Therefore, we insert an assertion in Line.4 to validate whether this situation could happen. Through testing and human checks, it is confirmed as a real program bug.

```
1  // file: badger/cmd/bank.go
2  func findFirstInvalidTxn(db *badger.DB, lowTs,
       ↪ highTs uint64) uint64 {
3      // ...
4      goassert.AssertOverflow(lowTs, highTs, lowTs+
           ↪ highTs)
5      midTs := (lowTs + highTs) / 2
6      err := checkAt(midTs)
7      if err == badger.ErrKeyNotFound || err == nil
           ↪  {
8          return findFirstInvalidTxn(db, midTs+1,
               ↪ highTs)
9      }
10     return findFirstInvalidTxn(db, lowTs, midTs)
11 }
```

Listing 4: An integer overflow bug

C. Lessons Learned

From the design and practical application of *Go-Sanitizer*, we have learned two lessons:

There are many bug categories ignored by existing testing methods. According to our experiment, the 9 bug types can hardly be captured by traditional testing methods, including both fuzz testing and unit testing. Some of them are corner issues, while some are security problems, such as the "regular expression without anchors". Although most of these types would not result in system crashes, it may still lead to vulnerability or function failure. Thus, it can be really helpful in practice for test engines to support these bug types.

Bug-Oriented assertions are effective in discovering program bugs. Assertions and oracles can help to find a series of program bugs, especially for those do not raise a crash. While many researchers have devoted huge efforts to develop more efficient fuzzing and verification algorithms, the efforts for assertion or oracle generation are less. More types of bug should be analyzed and supported. Furthermore, to integrate testing techniques or verification techniques with bug-oriented assertion generation is a more powerful way to ensure the correctness of the program.

V. Conclusion

We have proposed *Go-Sanitizer*, the first bug-oriented assertion generator for Golang, which can discover program bugs without signals of crash. We have applied it on Badger, and successfully discovered 12 unreported bug cases in Badger's code packages, which can not be detected by pure fuzz or unit testing methods. All the bugs are confirmed as real problems with manual validation. The experimental results show that *Go-Sanitizer* is effective and accurate in practice.

Our future work mainly includes two aspects. The first is to support more types of bug. We will define more assertion patterns and design more accurate location identification algorithms to insert those assertions. The second is to combine the bug-oriented assertions with the traditional testing methods more efficiently and seamlessly, for example, guide the Go-Fuzz to the locations with assertions.

References

[1] F. S. Shoumik, M. I. M. M. Talukder, A. I. Jami, N. W. Protik, and M. M. Hoque, "Scalable micro-service based approach to fhir server with golang and no-sql," in *2017 20th International Conference of Computer and Information Technology (ICCIT)*. IEEE, 2017, pp. 1–6.

[2] M. Andrawos and M. Helmich, *Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go*. Packt Publishing Ltd, 2017.

[3] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, 2018.

[4] N. Chalaemwongwan and W. Kurutach, "State of the art and challenges facing consensus protocols on blockchain," in *2018 International Conference on Information Networking (ICOIN)*. IEEE, 2018, pp. 957–962.

[5] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *International Conference on Software Engineering*, 2013.

[6] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey."

[7] D. Vyukov, "go-fuzz," https://github.com/dvyukov/go-fuzz, 2019.

[8] "Common weakness enumeration," https://cwe.mitre.org, 2019.

[9] dgraph io, "Badger," http://github.com/dgraph-io/badger, 2019.

[10] M. Zalewski, "American fuzzy lop," @http://lcamtuf.coredump.cx/afl/, [Accessed 2019].

[11] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 475–485.

[12] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 61–64.

[13] C. Wang, F. He, X. Song, Y. Jiang, M. Gu, and J. Sun, "Assertion recommendation for formal program verification," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2017, pp. 154–159.

[14] C. Wang, Y. Jiang, X. Zhao, X. Song, M. Gu, and J. Sun, "Weak-assert: A weakness-oriented assertion recommendation toolkit for program analysis," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 69–72.

[15] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 499–510.

[16] wangcong15, "cwe-testsuite-golang," https://github.com/wangcong15/cwe-testsuite-golang, 2019.