Local Analysis for Global Inputs

Alexander Kampmann
Saarland University
Saarbrücken, Germany
kampmann@st.cs.uni-saarland.de

***Abstract*—Fuzz testing and symbolic test generation both face their own challenges. While symbolic testing has scalability issues, fuzzing cannot uncover faults which require carefully engineered inputs. In this paper I propose a combination of both approaches, compensating weaknesses of each approach with the strength of the other approach.**

I present my plans for evaluation, which include applications of the hybrid tool to programs which neither of the approaches can handle on its own.

I. Introduction

Most software developers see testing as boring and tedious. This motivates attempts to automate test generation. There are two opposing strategies for automated bug finding:

1) The main idea in *fuzzing* is to use randomized inputs (e.g. [1], [2]). One would expect that the input validation of the program under test filters most of those, but surprisingly fuzzers can e.g. trigger the Heartbleed-Bug[3]. The reason is that fuzzing tries many different inputs really quickly, thereby rapidly exploring the program under test.
   Whether fuzzing discovers a bug or not depends on the probability of generating an input which triggers the bug. If only a few specific inputs in a large input space do, it will likely remain undiscovered. For most programs, fuzzing is already unlikely to generate a valid input. Therefore, fuzzing mostly uncovers problems in the validation logic.
2) Another approach to automated testing is *symbolic execution*. This technique selects a program path and generates a *path constraint*, a logical formula which describe conditions on the input that need to be fulfilled in order to execute this program path. A constraint solver can solve the constraint, and thereby generate an input which executes precisely the selected program path.
   However, large programs or large inputs structures lead to huge constraints, so the constraint solver needs a long time to solve them. Thus symbolic execution requires much more computational resources than fuzzing.

While fuzzing is fast and lacks precision, symbolic analysis is precise and lacks speed. In this proposal, I present the Basilisk-Framework, a *hybrid approach*, which combines the *precision* of symbolic techniques with the *rapid exploration capabilities* of fuzzers. The basic idea is to use system-level fuzzing to explore the program input space rapidly and unit-level symbolic execution to gain precision. This technique will be faster than symbolic test generation, because symbolic analysis is applied to parts of the system, rather than the whole system. At the same time, the Basilisk-Framework is more precise than fuzzing, because it leverages symbolic analysis for hard-to-reach parts of the system.

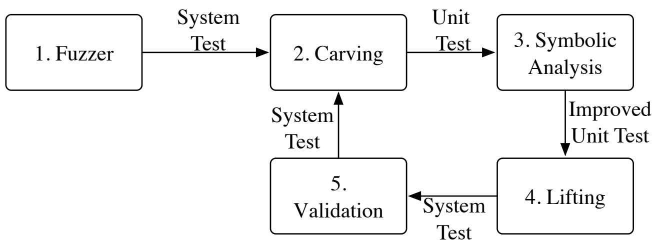

Figure 1. Overview of the Basilisk-Framework

An overview of the proposed framework can be found in Figure 1. The Basilisk-Framework generates system tests in five steps:

1) *Fuzzing*. First, large numbers of randomized system tests are generated. For example, the string `{"M7NR":0}` may be generated.
2) *Carving*. If a substring of the random input, say `M7NR`, occurs as a function argument in the program execution, I generate a unit test which invokes the observed function with the observed arguments. This technique is known as carving [4].
3) *Symbolic Analysis*. The unit tests will be analyzed with symbolic techniques, which yields more unit-level inputs. With respect to the running example, suppose the input `testitem` covers a new branch.
4) *Lifting*. The unit-level inputs will be *lifted* back up to the system level, which generates new system tests. In the running example, lifting generates the input `{"testitem":0}`.
5) *Validation*. The system tests will be executed to check whether they show the same behavior as their unit-level counterparts. If so, they become part of the generated test suite. New tests may again be subject to carving.

My hypothesis is that in this framework, fuzzing and symbolic execution cancel out each other's weaknesses. My research questions are designed to validate this hypothesis:

1) Does a hybrid approach outperform symbolic execution?
2) Does a hybrid approach outperform fuzzing?

Right now, automated unit-level test generation is useless. The reason is that the tests most of the time contain unit-level interactions which lead to misbehavior in the unit, but will never occur in the context of the whole system. This is called a *false positive*. However, unit-level test generation is faster than system-level test generation. I believe that my lifting and validation techniques can be integrated with any unit-level test generator in order to mitigate false positives. This opens the possibility to make unit-level test generation useful. Thereby research questions 3 and 4 are:

3) Can unit-level test generation be improved with lifting and validation?
4) Does unit-level fuzzing with lifting and validation outperform system-level fuzzing?

Also, existing research implies that there are programs where fuzzing or symbolic execution respectively are effective. This triggers another research question.

5) What properties of a program make fuzzing, symbolic execution or a hybrid approach effective (or ineffective)?

II. Related Work

In automated software testing, fuzzing and symbolic testing usually are perceived as two opposite approaches.

Symbolic testing analyzes the software under test and generates test cases which are carefully designed to cover specific test goals. Usually, those test goals are obtained from structural coverage criteria like branch coverage (e.g. [5]). Empirical studies (e.g. [6]) suggest that those criteria might not be sufficient.

Symbolic tools fall into two categories[1].

- Tools for *symbolic execution*, such as [5], execute the program symbolically, that is, all values are symbolic. In case of a branch, the tool decides which branch it wants to take and generates the constraint for it.
- In *dynamic symbolic execution* (e.g. [7], [8]) concrete and symbolic execution happen in parallel. In case of a branch, the concrete execution decides which branch to take, and the symbolic engine collects the constraint for this concrete path only.

While those approaches require instrumentation of the program under test, Fuzz testing[1] works with no or very limited knowledge of the program under test.

However, the distinction between black-box and white-box approaches is not that clear any longer. Feedback-driven fuzz testing tools like AFL [2] are based on search algorithms. They use program telemetry data to guide the search process. Still, they are much more randomized than the aforementioned white-box approaches. Some researchers refer to this kind of lightweight instrumentation as grey-box.

Grammar-based fuzzers like Peach [9], LangFuzz [10] or XMLMate [11] use a description of the input format, a grammar, to generate valid inputs only. This enables them to reach deeper layers of the program under test, however, a grammar needs to be written manually. In a personal conversation, one of the authors of LangFuzz reported that writing a grammar for JavaScript took more time than developing the fuzzer itself.

[1]And different researchers name the categories differently. There is even a paper([7]) which describes a symbolic technique with the term ”fuzzing”.

Höschele et al. [12] present how to use program telemetry to obtain input specifications. They utilize tainting in order to capture the data flow of programs and build context-free grammars from the data flow. This means the tedious process of manually writing a grammar can be automated.

There is some work on hybrid approaches as well. Păsăreanu et al. [13] presents a framework which combines system-level concrete executions with unit-level symbolic executions. However, they select values for the system-level executions via model-based simulations, which requires a pre-existing model.

Adding more and more program analysis to fuzzing comes with a price. Namely, the runtime of analysis is usually higher than that of program executions. Böhme et al. [14] analyze the probability of reaching a coverage goal and observe that if too much time is spent on collecting and analyzing telemetry data from test runs, it may not pay off, because randomized approaches may by chance reach the same goal faster.

In the evaluation of automated testers, known bugs in old versions of software may be used. There are several collections of programs with known bugs. Böhme et al. [15] created the CoreBench suite, a collection of 70 realistic regression errors from GNU Coreutils. Another collection is presented by Do et al. [16]. In the absence of real bugs, artificial bugs can be seeded. This technique is known as mutation testing [17].

III. The envisioned Framework

My envisioned framework works in five steps:

1) *Fuzzing* generates large numbers of randomized system tests.
2) *Carving* generates unit tests from the system tests, in a similar fashion to the technique by Elbaum et al. [4].
3) *Symbolic Analysis* analyzes the unit tests and yields more unit-level inputs.
4) *Lifting* translates the unit tests to system tests which trigger similar behavior.
5) *Validation* executes the system tests in order to eliminate false positives.

In the following, each step will be discussed in the context of an example.

A. Motivating example

I am using the program in Figure 2 as an example. It utilizes a JSON parser for input validation. JSON represents data in a recursive structure of key-value pairs.

The program first reads in a file (Line 2) and forwards it to the parser (Line 3). Afterwards, the function `process` checks whether the string `testitem` is used as a key within the input (Line 8). If so, lines 14 and 15 simulated a programming mistake.

Despite being highly artificial, this example demonstrates the limitations of both approaches. I executed a symbolic tool, KLEE [5], with a timeout of ten minutes on the example. It generated 52153 inputs, but none of them contained the string `testitem`.

In a second experiment, I wrote a simple Python script which randomly generates valid JSON documents. Those reached the

```
1  int main(int argc, char **argv) {
2    char *text = readEntireFile(argv[1]);
3    cJSON *json = cJSON_Parse(text);
4    return process(json);
5  }
6
7  int process(cJSON *json) {
8    cJSON *out = cJSON_GetObjectItem(
9    json, "testitem");
10   if(NULL == out) {
11     fprintf(stderr, "Invalid␣input!");
12     return 1;
13   } else {
14     char *test = NULL;
15     test[0] = 'c';
16   }
17   cJSON_Delete(json);
18   return 0;
19 }
```

Figure 2. Example code for JSON processing

```
{"M7NR": "UyXFx",
"6T": 0.302331771192897,
"0XQ": {"Z2qmQqSmys": []},
"8wcwBs": 0.17965215716521588}
```

Figure 3. Randomly generated JSON input.

`process` function, but none of them triggered the bug. The probability of generating `testitem` as a random string is too small.

B. Fuzzing

The first step in my proposed framework is *fuzzing*. For the example, I used randomly generated JSON documents like the one in Figure 3.

In my dissertation, I will not get involved with developing fuzzing tools, but rely on pre-existing tools like Peach or XMLMate.

C. Carving

The proposed framework executes the generated test inputs and collects information on *branch coverage*. Table I shows the coverage data for the five least covered methods. One of them is `cJSON_strcasecmp`. This method is called by `cJSON_GetObjectItem` (line 8) to compare the existing keys with the search key.

Function	branch coverage
cJSON_Delete	0
parse_hex4	0
parse_string	0.217
cJSON_strcasecmp	0.3333
main	0.3333

Table I
COVERAGE PER METHOD FOR RANDOMLY GENERATED INPUTS.

```
void snippet() {
  cJSON_strcasecmp("M7NR", "testitem");
}
```

Figure 4. The code snippet that was carved from the system test in Figure 3.

```
{"testitem": "UyXFx",
"6T": 0.302331771192897,
"0XQ": {"Z2qmQqSmys": []},
"8wcwBs": 0.17965215716521588}
```

Figure 5. Improved JSON input.

I can observe several calls to `cJSON_strcasecmp`, including an invocation with `M7NR` and `testitem` as parameters. Test carving [4] uses this invocation to generate the code snippet in Figure 4. This method is small and can easily be analyzed with symbolic execution.

For the example, I am using KLEE as a symbolic execution tool. `M7NR` was part of the original input, so the Basilisk-Framework instructs KLEE to find a replacement which covers more branches. KLEE is not allowed to change the second input, `testitem`, because it was not part of the JSON document. KLEE reports that the string `testitem` covers more branches within `cJSON_strcasecmp` than `M7NR`. Thereby, a unit test which calls `cJSON_strcasecmp` with `testitem` as first and second argument is generated.

Simple string comparisons may not always be precise enough to generate a meaningful mapping from inputs to function arguments. In this case, dynamic tainting may offer more precision, however, it also comes with greater runtime costs.

D. Lifting

In the last step, the framework *lifts* the analysis results from the unit tests back to the system level. In the example, I can replace `M7NR` with `testitem` in the system level input (Figure 3). This generates the input in Figure 5 which triggers the bug.

Grammar inference tools like AUTOGRAM automatically generate a context-free grammar of the input format. Most likely, a constituent of the grammar is handled as one piece by the program under test. If this holds true, the constraints that symbolic execution generates in the carving step deal with those input parts as a unit. In this case, the constraint can be translated into a representation which reasons about the constituents of the grammar, rather than individual input characters. Then, the grammar can be augmented with this constraint and fuzzers can generate inputs which fulfill the constraint. In this setup, the Basilisk-Framework uses symbolic execution to get a complicated condition right, and it uses the fuzzer again to explore the parts of the program which were not reachable without fulfilling this constraint.

E. Validation

The system test may fail to trigger the interactions that were observed in the unit test. That is because the program path of the original system test and the newly generated system test may diverge before the unit under analysis is reached. In the example, this would happen if the string replacement in the previous step yields an invalid JSON document. In this case, the unit test is a false positive. The system test will be discarded.

IV. Expected contributions

In my dissertation, I am planning to evaluate hybrid approaches in automated test generation. I will answer the following questions:

1) Does a hybrid approach outperform symbolic execution?
2) Does a hybrid approach outperform fuzzing?
3) Can unit-level test generation be improved with lifting and validation?
4) Does unit-level fuzzing with lifting and validation outperform system-level fuzzing?
5) What properties of a program make fuzzing, symbolic execution or a hybrid approach effective (or ineffective)?

Also, my implementation will be available as a basis for future experiments.

V. Evaluation

RQs 1, 2, 3 and 4 deal with the effectiveness of hybrid approaches. In order to evaluate those, runs of hybrid tools on different test subjects are needed. Interesting factors are the selection of test subjects as well as the criteria to collect.

Section II already discussed different criteria, including structural criteria, mutation scores and bug finding capabilities on software with known bugs. CoreBench contains only bugs from GNU Coreutils. Those can be handled by Klee. I will use them for comparison to Klee, but as my approach is supposed to be more powerful, I do need additional challenges. The collection by Do et al. contains several programs which do not read rich input data (e.g. tcas), are not written in C (e.g. ant) or are part of the GNU Coreutils as well (e.g. grep). So they may not be suitable either. I do not intend to compile my own collection of known bugs, so I will work with mutation scores and structural coverage criteria.

The second question is test subject selection. This is vital for RQ 5, which asks for properties of programs. So a diverse set of programs with different properties is needed. I think the relevant axis will be how much structure the input has and how large the program is. Those dimensions are most likely not independent.

GNU Coreutils rank as medium to low size, with only very little input structure. It has been shown that Klee can handle those [5]. Web servers like Apache HTTPD or nginx are larger, and, at the same time, HTTP requests are much more structured than the inputs of Coreutils. The ultimate challenge are compilers, e.g. gcc, or interpreters, e.g. php or lua. Here the inputs are turing-complete, so input validation rules are complex, and the input structure is very rich.

VI. Conclusion

As strategies for automated bug finding, fuzzing suffers from a lack of precision and symbolic execution lacks speed. I presented an artificial example which demonstrates both problems in just a few hundred lines of code.

My plan is to implement a *hybrid approach* which combines the strength of each of the individual approaches, mitigating each other's weaknesses. I demonstrated that this approach would be capable of detecting the bug in my example. My evaluation plans were presented in Section V.

My prototypical implementation already handles small examples. I expect full evaluation capabilities by May 2017.

Acknowledgment

This research has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7) / ERC Grant Agreement n. 290914 SPECMATE.

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] american fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[3] How heartbleed could've been found. [Online]. Available: https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[4] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving Differential Unit Test Cases from System Test Cases," *Sigsoft'06*, pp. 253–263, 2006.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, 2008.

[6] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 409–424.

[7] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[8] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[9] Peach fuzzer: Discover unknown vulnerabilities. [Online]. Available: http://www.peachfuzzer.com/

[10] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458.

[11] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller, "XMLMate: Evolutionary XML Test Generation," *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 719–722, 2014.

[12] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 720–725.

[13] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," *Proceedings of the 2008 international symposium on Software testing and analysis ISSTA 08*, pp. 15–26, 2008.

[14] M. Böhme and S. Paul, "A Probabilistic Analysis of the Efficiency of Automated Software Testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2016.

[15] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2014, pp. 105–115.

[16] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[17] R. J. Lipton, R. A. DeMillo, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE computer*, vol. 11, no. 4, pp. 34–41, 1978.