Summary Fuzzing and Concolic Testing

Sander Liebens 4207750
Patrick Brand 4330676

February 2016

1 Blackbox Fuzzing[5]

Software security issues are risky and expensive. Security testing employs a mix of techniques to find vulnerabilities in software. One of these techniques is *fuzz testing*, a process that automatically generates random data input.

1.1 JsfunFuzz

Jsfunfuzz is a black-box fuzzing tool for the JavaScript engine. *Jsfunfuzz* not only searches for crashes but can also detect certain correctness errors by differential testing. Jsfunfuzz was the first JavaScript fuzzer that was publicly available and thus inspired LangFuzz. In contrast, LangFuzz does not specifically aim at a single language. Instead our approaches aim to be solely based on grammar and general language assumptions and to combine random input generation with code mutation.

Mutation testing alone can miss a large amount of code due to missing variety in the original inputs. Still the authors believe that mutating code snippets is an important step that adds regression detection capabilities.

LangFuzz is a pure black-box approach, requiring no source code or other knowledge of the tested interpreter. While we consider coverage to be an insufficient indicator for test quality in interpreters, such an extension may also prove valuable for LangFuzz.

1.2 LangFuzz

A framework called *LangFuzz* was introduced that allows black-box fuzz testing of engines based on a context-free grammar. To adapt to *specific* targets, LangFuzz can use its grammar to learn *code fragments* from a given *code base*.

The combination of fuzz testing based on a language grammar and reusing project-specific issue-related code fragments makes LangFuzz an effective tool for security testing. At the same time, the approach can generically handle arbitrary grammars, as long as they are weakly typed.

The framework requires three basic input sources: *a language grammar* to be able to parse and generate code artifacts, *sample code* used to learn language

fragments, and a *test suite* used for code mutation. LangFuzz then generates new test cases using code mutation and code generation strategies before passing the generated test cases to a test driver executing the test case.

Typically, LangFuzz starts with a *learning phase* where the given sample code is parsed using the supplied language grammar, thereby learning code fragments.

Then the tool starts the actual working phase:

1. From the next test to be mutated, several fragments are randomly selected for replacement.

2. As a single fragment can be considered as multiple types, we randomly pick one of the possible interpretations for each of those fragments.

3. Finally, the mutated test is executed and its result is checked.

In the learning and mutation phase, we parse the given source code. For this purpose, LangFuzz contains a parser subsystem such that concrete parsers for different languages can be added. The parser is first used to learn fragments from the given code base which LangFuzz then memorizes as a token stream. We can mutate directly on the cached token stream.

The code generation step uses the stepwise expansion algorithm to generate a code fragment. However, because LangFuzz is a proof-of-concept, this subsystem only understands a subset of the ANTLR grammar syntax and certain features that are only required for parsing. LangFuzz uses further simplifications internally to make the algorithm easier.

With these simplifications done, the grammar only consists of rules for which each alternative is only a sequence of terminals and non-terminals. In case our stepwise expansion contains one or more synthesized rules, we replace those by their minimal expansion. All other remaining non-terminals are replaced by learned code fragments as described earlier.

After code generation, the fragment replacement code adjusts the new fragment to fit its new environment. For this purpose, LangFuzz searches the remaining test for available identifiers and maps the identifiers in the new fragment to exisiting ones.

In order to be able to run a mutated test, LangFuzz must be able to run the test with its proper *test harness* which contains definitions required for the test. LangFuzz implements this logic in a test suite class which can be derived and adjusted easily for different test frameworks.

LangFuzz and jsfunfuzz detect different defects (overlap of 15%) and thus should be used complementary to each other. A generic grammar-based fuzzer like LangFuzz can be 53% as effective as a language-specific fuzzer like jsfunfuzz.

2 Symbolic Execution[1]

A key goal of symbolic execution in the context of software testing is to explore as many different program paths as possible in a given amount of time, and

for each path to (1) generate a set of concrete input values exercising that path, and (2) check for the presence of various kinds of errors. The ability to generate concrete test inputs is one of the major strengths of symbolic execution. Symbolic execution is not limited to finding generic errors, but can reason about higher-level program properties.

The key idea behind symbolic execution is to use *symbolic values*, instead of concrete data values as input and to represent the values of program variables as *symbolic expressions* over the symbolic input values.

The goal of symbolic execution is to generate a set of inputs so that all the execution paths depending on the symbolic input values - or as many as possible in a given time budget - can be explored exactly once by running the program on those inputs. Symbolic execution maintains a symbolic state σ, which maps variables to symbolic expressions, and a symbolic path constraint PC, which is a quantifier-free first-order formula over symbolic expressions. At the end of a symbolic execution along an execution path of the program, PC is solved using a constraint solver to generate concrete input values. If the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way.

If a symbolic execution instance hits an exit statement or an error, the current instance of symbolic execution is terminated and a satisfying assignment to the current symbolic path constraint is generated, using an off-the-shelf constraint solver. The satisfying assignment forms the test inputs. Symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic.

A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along an execution path contains formulas that cannot be (efficiently) solved by a constraint solver.

One of the key elements of modern symbolic execution techniques is their ability to mix concrete and symbolic execution. Directed Automated Random Testing (DART), or Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete state and a symbolic state: the concrete state maps all variables to their concrete values while the symbolic state only maps variables that have non-concrete values.

The *Execution-Generated Testing (EGT)* approach works by making a distinction between the concrete and symbolic state of a program. To this end, EGT intermixes concrete and symbolic execution by dynamically checking before every operation if the values involved are all concrete. If so, the operation is executed just as in the original program. Otherwise, if at least one value is symbolic, the operation is performed symbolically, by updating the path condition for the current path.

Concolic testing and EGT are two instances of modern symbolic execution techniques whose main advantage lies in their ability to mix concrete and symbolic execution.

2.1 Advantages and Challenges

One of the key advantages in mixing concrete and symbolic execution is that imprecision caused by the interaction with external code or constraint solving timeouts can be alleviated using concrete values. Besides external code, imprecision in symbolic execution creeps in many other places and the use of concrete values allows dynamic sumbolic execution to recover from that imprecision, albeit at the cost of missing some execution paths, and thus sacrificing completeness. Dynamically symbolic execution's ability to simplify constraints using concrete values helps it generate test inputs for execution paths for which symbolic execution gets stuck, but this comes with a caveat: due to simplification, it could loose completeness, i.e. they may not be able to generate test inputs for some execution paths.

One of the key challenges of symbolic execution is the huge number of programs paths in all but the smallest programs, which is usually exponential in the number of static branches in the code. As a result, given a fixed time budget, it is critical to explore the most relevant paths first. There are two key approaches that have been used to address this problem: heuristically prioritizing the exploration of hte most promising paths, and using sound program analysis techniques to reduce the complexity of the path exploration.

2.2 Path explosion

The key mechanism used by symbolic execution tools to prioritize path exploration is the use of search heuristics. Most heuristics focus on achieving high statement and branch coverage, but they could also be employed to optimize other desired criteria. More recently symbolic execution was combined with evolutionary search, in which a fitness function is used to drive the exploration of the input space.

The other key way in which the path explosion problem has been approached was to use various ideas from program analysis and software verification to reduce the complexity of the path exploration in a sound way. Compositional techniques improve symbolic execution by caching and reusing the analysis of lower-level functions in subsequent computations. A related approach to avoid repeatedly exploring the same part of the code is to automatically prune redundant paths during exploration.

2.3 Imperfect symbolic execution

Symbolic execution of large programs is bound to be imprecise due to complex program statements and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision at a reasonable cost. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution. Whenever an actual execution path does not match the

program path predicted by symbolic execution for a given input vector, we say that a *divergence* has occurred.

2.4 Constraint Solving

Despite significant advances in constraint solving technology during the last few years, constraint solving continues to be one of the key bottlenecks in symbolic execution, where it often dominates runtime. In fact, one of the key reasons for which symbolic execution fails to scale on some programs is that their code is generation queries that are blowing up the solver.

The vast majority of queries in symbolic execution are issued in order to determine the feasibility of taking a certain branch side. Thus one effective optimization is to remove from the path condition those constraints that are irrelevant in deciding the outcome of the current branch.

One important characteristic of the constraint sets generated during symbolic execution is that they are expressed in terms of a fixed set of static branched from the program sourceode. For this reason, many paths have similar constraint sets, and thus allow for similar solutions; this fact can be exploited to improve the speed of constraint solving by reusing the results of previous similar queries.

The precision with which program statements are translated into symbolic constraints can have a significant influence on the coverage achieved by symbolic execution, as well as on the scalability of constraint solving. The trade-off between precision and scalability should be determined in light of the code being analyzed, and the exact performance difference between different constraint solving theories. In addition, note that in dynamic symbolic execution, one can tune both scalability and precision by customizing the use of concrete values in symbolic formulas.

3 Whitebox Fuzzing[3][2][4]

Hackers find security vulnerabilities in software products using two primary methods. The first is *code inspection of binaries*. The second is *blackbox fuzzing*, a form of blackbox random testing, which randomly mutates well-formed program inputs and then tests the program with those modified inputs. Although blackbox fuzzing can be remarkably effective, its limitations are well known: blackbox fuzzing usually provides low code coverage and can miss security bugs.

An alternative was developed: *whitebox fuzzing*. It builds upon recent advances in systematic dynamic test generation and extends its scope from unit testing to whole-program security testing. Whitebox fuzzing consists of *symbolically executing* the program under test *dynamically*, gathering constraints on inputs from conditional branches encountered along the execution.

In theory, systematic dynamic test generation can lead to full program path coverage, that is, *program verification*. In practice, however, the search is typically incomplete both because the number of execution, constraint generation,

and constraint solving can be imprecise due to imperfect symbolic execution that cannot all be solved perfectly in a reasonable amount of time.

Static analysis is usually more efficient but less precise than dynamic analysis and testing, and their complementarity is well known. They can also be combined. *Static test generation* consists of analyzing a program statically to attempt to compute input values to drive it along specific program paths *without ever executing the program*. In contrast, *dynamic* test generation extends static test generation with additional runtime information, and is therefore more general and powerful. Symbolic execution has also been proposed in the context of generating vulner- ability signatures, either statically or dynamically.

3.1 SAGE

Whitebox fuzzing was first implemented in the tool *SAGE (Scalable Automated Guided Execution)*. SAGE implements a novel directed-search algorithm called *generational search*, that maximizes the number of new input tests generated from each symbolic execution.

SAGE uses several optimizations that are crucial for dealing with huge execution traces. *Symbolic-expression caching* ensures that structurally equivalent symbolic terms are mapped to the same physical object; *unrelated constraint elimination* reduces the size of constraint solver queries by removing the constraints that do not share symbolic variables with the negated constraint; *local constraint caching* skips a constraint if it has already been added to the path constraint; *flip count limit* establishes the maximum number of times a constraint generated from a particular program branch can be flipped.

Building a system such as SAGE poses many other challenges: how to recover from imprecision in symbolic execution, how to check many properties together efficiently, how to leverage grammars for complex input formats, how to deal with path explosion, how to reason precisely about pointers, how to deal with floating-point instructions and input-dependent loops.

SAGE combines and extends program analysis testing, verification, model checking, and automated theorem-proving techniques that have been developed over many years.

3.2 Generational search

The generational search algorithm is designed to systematically yet partially explore the state spaces of large applications executed with large inputs and with very deep paths. It also maximizes the number of new tests generated from each symbolic execution while avoiding any redundancy in teh search. Generational search uses heuristics to maximize code coverage as quickly as possible. Besides that, it is also resilient to divergences: whenever divergences occur, the search is able to recover and continue.

4 Grammar-based Whitebox Fuzzing

Blackbox fuzzing sometimes uses grammars to generate the well-formed inputs, as well as to encode application=specific knowledge and test heuristics for guiding the generation of input variants. Whitebox fuzzing combines fuzz testing with dynamic test generation and executes the program under test with an initial, well-formed input, both concretely and symbolically.

Unfortunately, the current effectiveness of whitebox fuzzing is limited when testing applications with highly-structured inputs. *Grammar-based whitebox fuzzing* enhances whitebox fuzzing with a grammar-based specification of valid inputs. A dunamix test generation algorithm is presented, where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. The algorithm has two key components:

- Generation of higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional symbolic bytes read as input.
- A custom constraint solver that solves constraints on symbolic grammar tokens. The solver looks for solutions that satisfy the constraints and are accepted by a given (context-free) grammar.

4.1 Whitebox fuzzing algorithm

Grammar-based whitebox fuzzing prunes in one iteration the entire subtree of lexer executions corresponding to all possible non-parsable inputs. Given a sequential deterministic program P under test and an initial program input I, this dynamic test generation algorithm generates new test inputs by negating constraints generated during the symbolic execution of program P with input I. These new inputs exercise different execution paths in P. This process is repeated and the algorithm executes the program with new inputs multiple times - each newly generated input may lead to the generation of additional inputs. The algorithm terminates when a testing time budget expires or no more inputs can be generated.

Dynamic execution allows any imprecision in symbolic execution to be alleviated using concrete values and randomization: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete values of those inputs.

4.2 Grammar-based whitebox fuzzing algorithm

Grammar-based whitebox fuzzing is an extension of the algorithm described above:

- The new algorithm requires a grammar G that describes valid program inputs.

- Instead of marking the bytes in program inputs as symbolic, grammar-based whitebox fuzzing marks tokens returned from a tokenization function as symbolic; thus grammar-based whitebox associates a symbolic variable with each token, and symbolic execution tracks the influence of the tokens on the control path taken by the program P.

- The algorithm uses the grammar G to require that new input not only satisfies the alternative path constraint but is also in the language accepted by the grammar. This additional requirement gives two advatages to grammar-based whitebox fuzzing: it allows pruning of the search tree corresponding to invalid inputs, and it allows the direct completion of satisfiable token constraints into valid inputs.

The constraint solver computes language intersection: it checks whether the language $L(pc)$ of inputs satisying the path constraint pc contains an input that is in the language accepted by the grammar G. A *context-free constraint solver* takes as inputs a context-free grammar G and a regular expression R, and returns either a string $s \in L(G) \bigcap L(R)$, or $\perp$ if the intersection is empty.

4.3 Challenges and limitations

Computing language intersection - Computing the intersection of a context-free grammar with a regular expression is a well-known problem. *Approcimate grammars* - Grammar-based whitebox fuzzing can be used with approximate grammars. *Domain knowledge* - Grammar-based whitebox fuzzing requires a limited amount of domain knowledge, namely the formal grammar, identifying the tokenization function to be instrumented, and providing a de-tokenization function to generate input byte strings from input token strings generated by a context-free constraint solver. *Lexer and parser bugs* - Using a grammar to filter out invalid inputs may reduce code coverage in the lexer and parser themselves, since the grammar explicitly prevents the execution of code paths handling invalid inputs in those stages.

4.4 Results and observations

Among all the automated test generation strategies considered, grammar-based whitebox achieves the best total coverage as well as the best coverage in the deepest examined module, the code generator. It achieves results that are closest to the manual test suite, which predictably provides best coverage. The manual suite is diverse and extensive, but was developed with the cost of many man-months of work. In contrast, grammar-based whitebox requires minimal human effort, and quickly generates relatively good test inputs. Also the following was observed:

- *Grammar-based whitebox fuzzing* achieves much better coverage than regular whitebox fuzzing.

- Grammar-based whitebox fuzzing performs also significantly better than grammar-based blackbox.
- Grammar-based whitebox fuzzing achieves the highest coverage using the fewest inputs, which means that this strategy generates inputs of higher quality.
- The blackbox and whitebox strategies achieved similar results in all categories.
- Reachability results show tha talmost all tested inputs reach the lexer. The results show that grammar-based whitebox has the highest percentage of deep-reaching inputs.

The results of the experiments validate the claim that grammar-based whitebox fuzzing is effective in reaching deeper into the tested application and exercising the code more thoroughly than other automated test generation strategies. Grammar-based whitebox fuzzing tightly integrates constraint-based whitebox testing with grammar-based blackbox testing, and leverages the strengths of both. Since grammars are bound to be partial specifications of valid inputs, grammar-based blackbox approaches are fundamentally limited. Thanks to whitebox dynamic test generation, some of this incompleteness can be recovered, which explains why grammar-based whitebox fuzzing also outperforms grammar-based blackbox fuzzing in the experiments.

5 Evaluating Initial Inputs for Concolic Testing[6]

An evaluation method is presented to help concolic testing tool select initial inputs. This method assess different candidate initial inputs and ranks them according to their bug detection ability, which can help concolic testing tool select better initial inputs. The key insight behind this evaluation method is that: if the concolic execution triggered by the initial input can cover more error-prone operations with different execution contexts, it is likely to detect more unknown bugs more quickly.

During concolic testing, the selected initial input is used to generate new test cases to cover various paths of the target program.

In order to alleviate the influence caused by unknown suspicious points in uncovered paths and improve the accuracy of the evaluation method, an initial input scoring algorithm is presented, which not only considers the suspicious points encountered in execution trace of candidate initial input, but also path conditions related to these suspicious points (conditions having data-flow or strict control-flow dependencies with suspicious points.

For each candidate initial input, the evaluation method computes its ability to cover error-prone operations with different execution contexts to reflect its error detection ability. First, fine-grained dynamic taint analysis is used to identify operations which are prone to errors, and bytes of the input that can

flow into these suspicious points. Second, the suspicious conditions coverage and unique suspicious points coverage is computed.

The input evaluation method ranks the candidate initial inputs mainly according to their suspicious conditions coverage scores. The unique suspicious points coverage score is left as an auxiliary mechanism for it may be affected by unexecuted suspicious points.

5.1 Suspicious Points

Specifically, focus is given to the following two kinds of suspicious points.

- Dangerous functions: Security sensitive functions whose parameters can be affected by untrusted input data. When provided with a carefully crafted input, these dangerous functions are probably to trigger insufficient memory allocation, buffer overflow or integer overflow vulnerabilities.
- Dangerous instructions: Security sensitive instructions whose operands can be affected by untrusted input data.

Typically, there are two kinds of dependence relationships for dynamic taint analysis to consider: data-flow and control-flow dependencies. For data-flow dependencies, the dynamic taint analysis engine propagates the colors on data movement and arithmetic operations. Specifically, strict control dependencies are implemented that track the most informative dependencies. Finally, the dynamic taint analysis engine identifies the concerned security sensitive operations that can be affected by inputs as suspicious points, and logs all colors and input bytes involved in the the operands or parameters of suspicious points.

To identify suspicious conditions, the method checks whether the variables involved in one path condition have the same colors as some identified suspicious points. If true, this path condition is classified as "suspicious condition", because the condition and suspicious point can be affected by the same input bytes.

5.2 Initial Input Evaluation

To reflect the bugs detection ability of different candidate initial inputs, the method attempts to assess their ability to cover suspicious points with different contexts.

In the scoring algorithm, for each candidate initial input, the method assigns each byte of the input a weight approximating its influence degree on suspicious points.

For each candidate, the method evaluates its suspicious conditions coverage and unique suspicious points coverage in turn. In the unique suspicious points coverage evaluation procedure, each candidate input is evaluated based on simple statistics on the different suspicious points encountered in its execution trace. The current evaluation method treats different suspicious points equally.

Finally, each candidate initial input is assigned with unique suspicious points coverage score and suspicious condition coverage score. The more points the candidate cumulatively gets, the better coverage ability it has. The overall rank of the candidate input is determined by both the unique suspicious point coverage scores and suspicious conditions coverage scores

By default, for each candidate, our evaluation method assesses it by monitoring and analyzing its own execution trace.

- During input evaluation procedure, the initial input scoring algorithm not only considers the suspicious points themselves, but also suspicious conditions.
- The target candidate initial inputs is restricted to well- formed ones. In contrast, well-formed inputs usually result in similar processing procedure in the target program and their bug detection ability can be estimated based on their execution trace as analyzed in the motivating example section.

References

[1] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[2] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[3] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[4] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[5] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[6] Weiguang Wang and Qingkai Zeng. Evaluating initial inputs for concolic testing. In *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on*, pages 47–54. IEEE, 2015.