A Survey of Automatic Software Testing Techniques

Roeland Krak
University of Twente
r.s.krak@student.utwente.nl

Joris Diesvelt
University of Twente
j.j.diesvelt@student.utwente.nl

***Abstract*—Automated software testing is a field which comprises multiple techniques that can be applied to systematically test software systems. We discuss fuzz testing, symbolic testing, and concolic testing and their appliances. We illustrate these techniques with tools that have been successful at finding bugs in real software. Lastly, we discuss path explosion, which is one of the major challenges for symbolic and concolic execution.**

1. INTRODUCTION

Software testing is an important part of software development which helps discover complex bugs and vulnerabilities. There are two broad approaches to software testing. Either the code of the tested software is available, or it is not. These approaches are known as white-box and black-box testing respectively.

Traditional testing has focused on white-box unit testing [5], which means developers write code which tests the correct performance of individual software components such as functions and classes. Another approach is automatic testing, which requires less work from developers because they do not have to construct individual test cases. Automatic testing can be applied both in white-box and black-box settings.

The standard technique of automated black-box testing is known as *fuzzing*, or fuzz testing, which means many concrete inputs are provided to the software. The testing program then monitors the tested program for exceptions, such as whether the tested software crashes or assertions fail [3]. The tested inputs can be randomly constructed, or generated using specific heuristics which should be more likely to find bugs. However, fuzzing generally does not use knowledge of the inner workings of the tested software which means it can be hard to find inputs which cover rarely executed program paths.

Another automated testing technique is *symbolic* execution, which requires access to the code of the tested software and is thus a white-box testing technique. Symbolic execution interprets the code of the tested software and reasons about symbolic values instead of concrete values. Because of this, it can generate constraints on inputs which determine which execution paths are taken by the tested software [2, 6]. This allows symbolic execution to reach high program coverage as it can direct testing towards untested execution paths.

Symbolic execution has several challenges, which can generally be categorized as path explosion problems, dealing with black-box functionality, and complex constraints [2]. Path explosion problems are situations in which the number of possible execution paths is too large to cover completely [1]. Even if the code for software is available, there are often system parts involved for which the code isn't available or significantly complex, such as kernel functions and libraries. The third challenge arises when constraints become so complex that constraint solvers cannot provide a solution in reasonable time.

A recent development is the combination of concrete and symbolic execution, known as *concolic execution* [5][4, 2]. This combination allows testing of software – or parts thereof – using concrete values, while at the same time reasoning about execution paths. This allows the testing program to partially overcome the challenges of symbolic testing while at the same time being able to achieve high program coverage.

Symbolic and concolic execution find appliance in many different testing situations. They have been applied on a wide range of languages, including Java [4, 5], C [5], and ARM and x86 instructions [6]. They have been applied on firmware, software for general-purpose devices, and applications running on specialized operating systems such as those for smartphones. Fuzzing is able to detect bugs that are detectable by monitoring exceptions generated by a program, but symbolic execution is also able to reason about higher-level program properties [2] and logic flaws, such as whether authentication interfaces can be bypassed.

2. FUZZING

In this section we illustrate fuzz testing with LangFuzz, which is able to detect bugs and security issues in code interpreters. Because only valid code is accepted by interpreters, strategies are needed for fuzzing frameworks. LangFuzz is such a framework which generates code for testing, based on a context-free grammar, sample code and a test suite [3].

2.1 Background

Fuzz testing ideas have been around since 1972. Since then, several tools have been developed to test compilers and interpreters. In 2001, CSmith was presented, which generates and tests C programs for correctness. Like LangFuzz, it uses filter functions to limit the number of different productions. The most influential tool in this field is jsfunfuzz, which was designed to black-box test a JavaScript engine. Two approaches for this type of fuzzing are random test generation and modifying existing valid inputs.

2.2 LangFuzz

LangFuzz adapts these two approaches and has developed two major techniques: generative and mutative code generation. In generative code generation, random output is generated based on constraints or rules. Mutative generation, on the contrary, creates new output from randomly modifying existing code. LangFuzz's primary technique is mutation, but uses both.

Mutation consists of learning and mutation phase. During learning, code fragments are loaded into a fragment pool. During mutation, code fragments are loaded again but parts are interchanged with fragments from the pool to generate test cases. Mutation is capped by the available code fragments. Code generation instead generates random code based on a grammar. By randomly padding non-terminal symbols with other symbols, a syntax tree is build. This is known as step-wise expansion.

A problem with code mutation is that resulting code might not be semantically correct. Identifiers are often used without being declared. A solution is to interchange identifiers with ones that are used somewhere else in the program or use built-in objects/globals.

2.3 LangFuzz Implementation

LangFuzz starts with the learning phase as described above. It uses ANTLR to be able to easily parse any language. Because LangFuzz is a proof of concept, many complex expressions are simplified. Then, by using step-wise expansion, random code blocks are generated in a predefined number of steps. All remaining non-terminals are replaced with learned code fragments.

To be able to run tests on an interpreter, LangFuzz uses a "test harness". This contains definitions that are required to run one or more tests. To speed up testing, LangFuzz uses a persistent shell: with the harness, a driver is started, which keeps the interpreter alive for the remainder of the tests. The persistent shell has an additional effect: the different tests influence each other and therefore increase the probability of detected flaws. In this case, to find out which test caused an error, the delta debugging algorithm is used.

2.4 Evaluation

LangFuzz was evaluated by comparing its performance with jsfunfuzz. jsfunfuzz is specifically designed to test JavaScript engines and their newest features, where LangFuzz takes a more generic approach. Both systems are tested against Mozilla's JS engine TraceMonkey for a few reasons. The development process and artifacts are made public and jsfunfuzz was already being used by the Mozilla team. The downside of this is that jsfunfuzz would probably not find many issues and thus instead, the list of issues that are found by the Mozilla team is used for this research. The tools are ran on certain revisions of the system between periods of time called testing windows. Each testing window is tested for 24 hours by both tools.

2.4.1 *Overlap and detection rate*

The LangFuzz project attempts to answer to what extend the detected defects by LangFuzz and jsfunfuzz overlap. The overlap is defined by the number of defects found by both tools divided by the number of defects found in total. After the tests, the overlap was determined to be 15%. Some defects missed by jsfunfuzz where mostly garbage collector related.

The other performance measure is the detection rate and by extend the effectiveness of LangFuzz, compared to jsfunfuzz. Effectiveness is defined as the number of defects found by LangFuzz divided by the number of defects found by jsfunfuzz. This tries to answer if the program can be a useful contribution, even though a different tool already performs well. It was concluded that LangFuzz was 53% – about half – as effective as jsfunfuzz.

2.4.2 *Generative vs Mutative*

LangFuzz implements two different techniques, generative and mutative code generation. The evaluation of this program also attempts to figure out which technique is most important and which gives the best performance. Different configurations of LangFuzz are used to answer these questions. First only learning and replacing is used, then only code generation. The second approach is difficult because it generates completely random code which, when becoming larger, has a high chance of introducing semantic errors. Mostly because you cannot syntactically generate a good environment (the undeclared identifier problem for example). Both configurations are ran on the Mozilla JS engine and the defects found are compared. As it turns out, both configurations find most of the existing defects, but both separate configurations also found unique defects. In conclusion, both code generation and mutation should be used in a mixed setting.

LangFuzz was also tested on the newest version of TraceMonkey, Google's V8 engine and the PHP engine. For each program, between 51 and 59 defects where found in the course of 9 months. Of these defects, a small portion were defined as security issues by the respective developers and awarded with bug bounties.

2.5 Adaptation to PHP

LangFuzz was adapted to PHP as a proof of concept, to show how it can be used in different settings. Some steps were needed to make this possible: Parser/Lexer classes had to be created using ANTLR, a test suite had to be written and language-dependent information had to be added. This last step is optional but increases performance [3].

2.6 Discussion on its validity

LangFuzz has been tested with JavaScript engines and the PHP language, but it cannot be generalized that it works for all languages. Its comparison with jsfunfuzz is limited to a single version of this program, thus gives no guarantee for all cases. The quality and size of test suites are very important for LangFuzz's performance and can influence results. Both tools also use a lot of randomness, thus results may very on each iteration of tests.

3. SYMBOLIC EXECUTION

Symbolic execution is a method of automatically analyzing complex code and finding deep errors and vulnerabilities, and generating test cases. The concept of symbolic execution is over three decades old, it wasn't practical until recent advances in the field have made it practical. Recent advances have combined Symbolic execution with concrete execution, using concrete values to run parts of code which are impractical to solve symbolically. The combination of these techniques is known as concolic execution [2].

Symbolic execution is used to test many different program paths and reach deep program states, and generate sets of concrete inputs that reach those states. The technique can be used to reason about high-level program assertions and as such it can be used to test for many different kind of bugs and vulnerabilities. The main challenges for symbolic execution are path explosion, black-box functions, constraint solving, and memory modeling.

3.1 Classical Symbolic Execution

Symbolic execution interprets software systems using symbolic values instead of concrete inputs. Output values of functions and programs are then expressed as functions of the symbolic input values. Branching operations are expressed as constrains on symbolic values, and the execution tree of the program is a tree where each branch takes one of the possible execution paths at each branching operation. Execution paths are then sequences of true and false statements expressing which branch was taken at each consecutively encountered branching operation.

Symbolic execution maintains a system state in which each variable is expressed as a symbolic expression, and a path constraint determines which execution path has been taken so far. At any point a constraint solver can be used to generate a set of concrete inputs that force the software to take a certain execution path corresponding to those path constraints.

Symbolic execution can run either until a limit on time or execution depth has been hit, or until it encounters a program crash, assertion violation, or exit statement. Execution containing loops or recursion based on symbolic conditions may result in an infinite number of execution paths. The maximum depth to explore in these situations is generally restricted [2].

Classical symbolic execution is limited by the ability of the constraint solver to efficiently solve functions, and may not be able to solve nonlinear constraints or system calls of which the code is not available.

3.2 Modern Symbolic Execution

3.2.1 Concolic Testing

Performs symbolic execution while at the same time executing a program on concrete values, maintaining a concrete and a symbolic state simultaneously. Concolic execution requires concrete inputs to initialize the concrete state. The constraint solver can be used to generate alternative concrete inputs which steer execution to a certain execution path. We go into the details of concolic execution in section 4.

3.2.2 Execution-Generated Testing

The EGT approach makes a distinction between concrete and symbolic execution states. It checks before every operation if all values involved are concrete. If no values are symbolical the operation is executed concretely [2].

3.3 Challenges

There are numerous challenges for symbolic execution. In this section we describe several of these challenges and briefly touch on some possible solutions.

3.3.1 Path explosion

A major problem for symbolic execution is path explosion, the issue of having too many execution paths to test within reasonable time. Several optimizations exist to deal with this problem, we go further into this problem and the solutions in section 5.

3.3.2 Constraint Solving

Constraint solving is a significant bottleneck for symbolic execution, because some queries are too hard to solve within reasonable time.

One optimization is to eliminate irrelevant constraints which do not play a role in a branching operation. This optimization can often be applied because in practice branches only depend on a small number of program variables [2].

Another optimization is to store solved queries and use previously solved queries to solve new ones. This is often feasible because constraints are generated by a limited set of branches, which means many constraints contain similarities.

3.3.3 Memory Modeling

Memory modeling may cause challenges because there is a trade-off between the efficiency of modeling program variables as a certain type, whereas it may miss certain corner-case executions caused by the actual implementation of the program variables [2]. Corner-cases that may be missed are e.g. integer-overflows and the like.

Another memory modeling challenge is to deal with pointers, as actual memory layouts are highly dependent on compilers and may be affected by functions which make execution intention unclear.

3.3.4 Handling Concurrency

Testing is considered very difficult for large real-world applications because of concurrency. These issues are caused by the non-deterministic behavior of concurrent applications. Symbolic execution has been successfully applied to test concurrent programs effectively [2].

3.4 Firmalice

We illustrate symbolic execution using Firmalice, which is a tool that implements symbolic execution to detect authentication bypass vulnerabilities in firmware [6]. This class of vulnerabilities allows an attacker to execute commands which should be privileged, but can be accessed without proper authentication because of the vulnerability.

The reason Firmalice focuses on firmware is that embedded devices – possibly connected to the Internet – have become ubiquitous, and the firmware they run is becoming increasingly complex. This firmware may, just as any other software, contain vulnerabilities. Furthermore, many of these devices manage privacy sensitive data acquired by sensors or through interaction with other devices.

Analyzing firmware is challenging because of several reasons. Firstly, source code of firmware is often not available to third parties because the firmware is proprietary. Secondly, binary images of firmware often have nonstandard and undocumented structures which complicates analysis. Because of this, initialization functions and execution starting points require effort to discover.

Firmalice operates on security policies, which define points in the binary which should only be executed if a user has properly been authenticated. The tool requires users to specify these points for instance by certain strings printed after authentication, or by defining certain memory addresses which aren't supposed to be accessed. Firmalice then reasons that input deterministically leading to those privileged points constitutes an authentication bypass vulnerability [6].

3.4.1 *Authentication Bypass Vulnerabilities*

The authentication bypasses that can by found by Firmalice tend to be of one of several classes. These include hardcoded credentials, hidden authentication interfaces, and unintended bugs.

3.4.2 *Approach*

The approach of Firmalice is to first load the firmware. The analyst then supplies a security policy which Firmalice translates into analyzable program properties. The firmware is then analyzed statically to create authentication slices from entry points to the privileged program point. The authentication slice is then executed symbolically to find a path given certain input, from the entry point to the privileged program point. Any found paths with corresponding inputs are then analyzed to determine if they constitute authentication bypasses. Firmalice may then generate a function which gives the user access to the privileged program point by providing input that triggers the authentication bypass.

3.4.3 *Firmware Loading*

Some embedded devices run general-purpose Operating Systems such as Linux. The device's functionality is then implemented in user-space programs. Other firmware takes the form of a single binary image for which it is undocumented how to initialize the runtime environment or start program execution.

Firmalice has implemented solutions to deal with the latter form's challenges. Firstly, the binary is disassembled using existing techniques. Secondly, Firmalice uses jump tables – which contain absolute code addresses – to determine at which base address to load the firmware in memory. After that, the input entry points of the program are determined. To do so, Firmalice first scans through the binary to identify functions. It creates a directed call graph between these functions and marks root nodes as potential entry points [6].

3.4.4 *Security Policies*

Authentication bypass is a form of a logic flaw. Logic flaws are harder to detect than certain other types of vulnerabilities as what constitutes a logic flaw is highly dependent on the intention of the developer. Firmalice solves this by leveraging privileged program points supplied by an analyst [6].

Privileged program points can take several forms. They can be static output of a certain string of text. They can also be behavioral rules about certain actions a device can take. These actions are e.g. instructions to certain actuators or file accesses. Furthermore, they can be accesses to certain memory locations or can be a certain privileged function of the program.

3.4.5 *Static Program Analysis*

To deal with the problem of path explosion, Firmalice first reduces the code it has to analyze. It does so by selecting a code slice which covers the code required to go from entry points to privileged program points. This code reduction is called a slice, and Firmalice specifically creates a backward slice from the privileged program point backwards to the entry point.

First, a control flow graph (CFG) is created by statically analyzing the firmware. The CFG is a representation of all possible execution paths through the program. Firmalice can handle computed jumps by leveraging its symbolic execution engine.

Then, a control dependency graph (CDG) is created from the CFG. A CDG represents, for each statement, which other statements determine whether that statement is executed. Thus, a CDG and CFG may be used to identify which statements may be executed before any given statement is executed.

Subsequently, a data dependency graph (DDG) is created, which shows how instructions correlate with each other to produce and consume data. As the DDG is formed from the CFG, it also suffers from any imprecisions in the CFG.

The CDG and DDG together comprise the program dependency graph (PDG), which allows one to reason about the control and data flow required to arrive at a specific point in a program. Using the PDG, a backwards slice is computed for a relevant program point. The backwards slice comprises every statement on which the program point depends, for a specific entry point.

3.4.6 *Symbolic Execution Engine*

The symbolic execution engine performs symbolic execution on the backwards slice. The engine operates on symbolic program states and constraints on the values of abstract variables contained in these states [6].

Whenever the symbolic execution engine discovers a path to a privileged program point, it labels the state as privileged and passes it to the authentication bypass check module for further analysis.

Firmalice achieves efficiency through symbolic summaries, which are human supplied semantic descriptions of the effects that certain commonly-seen functions provide. The use of summaries avoids unnecessary branching during the execution of these functions. Furthermore, constraints generated by summarized functions are often simpler than they would otherwise be.

As it is impossible to straightforwardly know which functions in a binary-blob are initialization functions, programs may run into the problem of not properly initializing all memory locations before execution. To deal with this, Firmalice notices when it attempts to read from uninitialized memory. It then identifies other procedures with direct memory writes to that location. As it cannot be known which of these procedures are certainly initialization procedures the symbolic state is duplicated, and the potential initialization procedure is executed on the duplicated state before program execution continues [6].

3.4.7 Authentication Bypass Check

The Authentication Bypass Check module takes a privileged state from the symbolic execution engine and attempts to uniquely concretize the constraints on the input required to reach the privileged program point. If it is able to find this concretization, possibly based on output generated by the program, the path generated by such input is labeled as an authentication bypass. If the input is indeed dependent on program output, the module creates a function which generates valid input based on this output.

Firmalice can reason about network traffic, stdin and stdout, and firmware-specific interrupts. One of the core ideas of Firmalice is that data exposed by output also reveals information about any variables related to that data, which may leak information required for an authentication bypass [6].

3.4.8 Evaluation

Firmalice was evaluated against firmware for three different devices, and was able to find – previously known – vulnerabilities in two of those.

3.4.9 Discussion

Firmalice is unable to detect all authentication bypasses. Specifically it is unable, or deterred, to detect bypasses in the presence of obfuscated code, constraints which are hard to solve, and irreversible operations such as hash functions.

Furthermore, Firmalice expects authentication bypasses to have unique inputs, and will therefore not be able to discover bypasses which have multiple solutions.

Many optimizations of Firmalice in respect to scalability have a negative impact on its soundness. This is a general trade-off found in symbolic execution systems.

One of the large problems of Firmalice, and symbolic analysis in general, is the path explosion caused by symbolic loop analysis. Firmalice is able to partially mitigate this by use of symbolic summaries.

4. CONCOLIC EXECUTION

In this section we present concolic execution, which involves execution of programs simultaneously on concrete values as well as on symbolic values [5]. After the concrete execution has finished, all symbolic values are used to create constraints on the followed execution path, known as the *path constraint* [4].

Next, the individual constraints in the path constraint are sequentially negated to generate constraints for alternative execution paths. These new constraints are passed on to a constraint solver which uses it to generate new concrete inputs confirming to these constraints. If such an input is found, it is guaranteed to follow a unique execution path.

Concolic execution allows the execution engine to replace any constraints which are too complex for the constraint solver, with the concrete counterparts [5]. This allows concolic execution to easily recover from situations in which the constraint solver has a time out, as well as from situations in which constraints are generated that the constraint solver cannot handle.

An advantage over fuzzing is that concolic execution achieves higher path coverage, but at the cost of computational work, especially by the constraint solver. Furthermore, because concolic execution keeps track of a symbolic state, it is able to reason about the same higher-level program properties as symbolic execution.

An advantage over classical symbolic execution is that execution on concrete values alleviates imprecision from symbolic execution, because it allows to execute certain code parts and outside world interactions concretely and randomly. Code parts that are hard to solve symbolically are e.g. functions that cause constraint solver timeouts. Executing those parts concretely allows to recover from that imprecision, but comes at the cost of possible incompleteness [2].

However, concolic execution also comes with disadvantages. After every execution, the path constraints need to be negated and a solution to the negated constraints has to be generated by a constraint solver. Constraint solving is no trivial task and may require significant computational time for complex constraints. Because concolic execution requires a significant amount of constraints to be solved, this means computational time spent on constraint solving will be significant.

4.1 CUTE

CUTE is a concolic execution tool which was originally developed for C [5]. One of the functions of CUTE is to find bugs caused by race conditions in concurrent software. jCUTE is a tool based on CUTE and implements the same functionality for Java.

CUTE consists of two modules, one of which inserts code into the program under test to call the testing module during runtime. The testing module performs symbolic execution, constraint solving, and thread scheduling. The latter is important because CUTE was developed to handle concurrent software.

CUTE has been successfully applied on `SGLIB`, an open source C library for generic data structures, and on the thread-safe Collection framework, which is part of the `java.util` package of the standard Java library of Sun Microsystems. CUTE was able to find bugs in both of these systems.

4.2 jFuzz

Concolic execution, or "concolic white-box fuzzing" as they call it, was also implemented in jFuzz [4]. jFuzz is an automatic testing tool for Java programs which was implemented on the NASA Java Pathfinder (JPF) framework. We will use this tool as an illustration on concolic execution.

Several optimizations have been implemented in jFuzz that alleviate the interactions with the constraint solver. Firstly, constraints are cached name-independently. This means a cache is maintained of previously solved constraints. These constraints are normalized so matches in the cache are found independent of variable names. This allows optimization based on constraint patterns, and not on name-specific matches only. This can reduce a significant amount of redundant calls to the constraint solver [4].

4.3 Concolic Versus JPF Random Fuzzing

Because jFuzz was implemented on a testing framework, a proper comparison can be made with other testing techniques.

jFuzz was compared to random fuzzing using the JPF framework. It was found that jFuzz added approximately 15% overhead above normal JPF execution, which means it was able to generate 15% fewer inputs. However, because the inputs were generated systematically higher line coverage was achieved. In the performed tests the line coverage of jFuzz was 18%, versus 15% coverage by normal JPF execution.

It was furthermore found that the constraint optimizations are relevant for practical applications, as they reduced computational time spent on constraint solving by approximately 25%.

5. DEALING WITH PATH EXPLOSION

The number of possible execution paths is enormous in most real software systems, as it is generally exponential in the number of static branches. Dealing with this huge number of paths is one of the main challenges of symbolic execution [2]. To deal with path explosion heuristics are often applied to prune unfeasible execution paths, or paths which do not depend on symbolic input. Tests on multiple medium-sized applications showed less than 42% of executed statements depended on symbolic input, and often less than 20% of symbolic branches encountered during execution had two feasible sides.

Search heuristics are the main tool to prioritize path exploration, often focusing on high statement or branch coverage. Heuristics may for example guide exploration towards uncovered instructions, or to statements that are executed the fewest number of times. Random branching strategies have also proven successful to achieve high statement coverage.

Concrete, random, initial testing may be combined with symbolic execution, to initially reach deep program states from which to continue searching symbolically.

Multiple paths may cover the same lines of code redundantly, and these may be pruned if they have similar constraints on the same piece of code. This technique can significantly reduce the number of paths te explore.

Lazy test generation uses a multifaceted approach. It first uses concolic execution to analyze functions with unconstrained inputs. Secondly, it attempts to expand the trace to concretely realizable executions by recursively expanding the called functions to find concrete executions and adding constraints to function inputs. This way the functions can be stitched together with the original trace to form a complete program execution. This allows the analysis to be performed on a higher level of abstraction, not reasoning about interprocedural paths but intraprocedural paths.

Static path merging is a technique which merges constraints on paths and thereby hands the complexity from the symbolic analysis engine to the constraint solver.

Another method is the use of code slices of the program and constructed program dependency graphs (PDG) to analyze the control and data flow of the program. This method is used by Firmalice and is discussed in section 3.4.5.

5.1 The RWset analysis approach

A strong approach to attack path explosion in constraint-based test generation is the read-write set (RWset) analysis method, developed for the EXE symbolic execution tool [1]. It attempts to reduce path explosion by pruning the paths that have the same effects on the program execution as previously explored paths. There are two main ideas when to remove paths. First, if an execution reaches a point in the same state as a previous execution it can be pruned and second, if two states only differ in program values that will not be read then they can be treated as the same and can thus also pruned.

5.1.1 Overview

The constraint-based execution discussed is in context of the testing tool EXE. This system allows users to mark memory locations which hold symbolic data. It then tracks these symbolic values on execution. There, each statement is added as a constraint to the current path and execution is branched. These constraint are solved using the constraint solver. Naturally, with the number of conditional statements, the amount of explorable paths increases exponentially.

The main idea of RWset is to stop an execution path once it is known and proven that this path will only produce effects that were already seen before. In such a case EXE generates a test case and stops execution, which is called a cache hit. To increase the odds of such an event happening, some refinements are used. If a value would not be read after some execution point, it is independent from the remaining execution and is dropped from the constraint cache, which contains all added constraints from a certain execution path. To determine if values are still "alive", a list of live variables is kept.

In loops, EXE attempts to create a limited number of possible paths by setting a concrete value for the loop constraint. Each path then has its own constraint. It is possible that all but one paths are pruned, which leaves only one execution path after the loop.

5.1.2 Key implementation details

Symbolic execution can be made more efficient by doing as many things concretely as possible. For efficiency symbolic and concrete cases are separated. A program state includes both path constraints (symbolic state) and values for concrete memory locations (concrete state).

Starting from the initial state, the program tracks all values that are set along the path. These are grouped in the *writeset*. For example, if value x is assigned to a memory location v, the entry (v, x) is added to the writeset. Entries are removed from this set when the value is either deallocated or is changed into a symbolic value, in which case the symbolic value is added to the path constraint set.

To check whether two states in program execution are equal, the tool also needs to check the context from which functions are called. For example, if a certain function is executed, the programs state is not automatically the same as when the same function is called from a different position. The RWset implementation uses a MD4 hash of the callstack to verify this equality, but many other approaches would also work.

The set of locations where a value is being read after a program point is collected in the readset. This means that any value not in this set can be discarded. To build the readset for a certain value, a complete depth-first traversal of all paths after the program point is performed. Any values read in these paths is added to the readset. When propagating backwards, all locations that are deallocated or otherwise changed are removed from the set.

Given a program state, the readset and writeset, irrelevant parts of the program can be removed. In a concrete state, only locations are kept in the intersection of the readset and writeset. For the symbolic counterpart, only the constraints on this symbolic state are kept that include the values from the readset.

5.1.3 Performance of RWset

The RWset analysis was performed on five medium-sized open-source applications. In 30 minutes the performance of EXE with and without this algorithm was recorded by capturing the maximum branch coverage and how many test cases were necessary to achieve that coverage. In general, about half the number of test cases were needed with the RWset version to reach the same branch coverage. The performance on several device driver were also tested, and results were positive in this case as well. The RWset version found several unique bugs more than the base version because it reached a higher code coverage.

6. CONCLUSION

Automatic software testing techniques are an effective way of testing both complete software systems and parts thereof. Fuzzing is a technique specifically aimed at black-box tests, whereas symbolic execution is a technique which can only be applied to white-box tests.

Concolic testing is a more recent technique for white-box testing, and combines both concrete and symbolic execution.

These testing techniques have been widely applied in a diversity of tools of which we used some to illustrate the techniques. These tools have all proven successful in finding bugs and vulnerabilities in real software systems.

There are still challenges in the field of symbolic and concolic execution, of which one of the most important ones is dealing with path explosion. However, multiple techniques have been developed to alleviate path explosion problems.

7. REFERENCES

[1] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

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

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

[4] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NASA Formal Methods*, pages 121–125, 2009.

[5] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[6] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.