DOI:10.1145/3363824

Reviewing software testing techniques for finding security vulnerabilities.

BY PATRICE GODEFROID

Fuzzing: Hack, Art, and Science

FUZZING, OR FUZZ TESTING, is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed, inputs.[35] Since the early 2000s, fuzzing has become a mainstream practice in assessing software security. Thousands of security vulnerabilities have been found while fuzzing all kinds of software applications for processing documents, images, sounds, videos, network packets, Web pages, among others. These applications must deal with untrusted inputs encoded in complex data formats. For example, the Microsoft Windows operating system supports over 360 file formats and includes millions of lines of code just to handle all of these.

Most of the code to process such files and packets evolved over the last 20+ years. It is large, complex, and written in C/C++ for performance reasons. If an attacker could trigger a buffer-overflow bug in one of these applications, s/he could corrupt the memory of the application and possibly hijack its execution to run malicious code (elevation-of-privilege attack), or steal internal information (information-disclosure attack), or simply crash the application (denial-of-service attack).[9] Such attacks might be launched by tricking the victim into opening a single malicious document, image, or Web page. If you are reading this article on an electronic device, you are using a PDF and JPEG parser in order to see Figure 1.

Buffer-overflows are examples of security vulnerabilities: they are programming errors, or bugs, and typically triggered only in specific hard-to-find corner cases. In contrast, an exploit is a piece of code which triggers a security vulnerability and then takes advantage of it for malicious purposes. When exploitable, a security vulnerability is like an unintended backdoor in a software application that lets an attacker enter the victim's device.

There are approximately three main ways to detect security vulnerabilities in software.

Static program analyzers are tools that automatically inspect code and

» key insights

- **Fuzzing means automatic test generation and execution with the goal of finding security vulnerabilities.**
- **Over the last two decades, fuzzing has become a mainstay in software security. Thousands of security vulnerabilities in all kinds of software have been found using fuzzing.**
- **If you develop software that may process untrusted inputs and have never used fuzzing, you probably should.**

IMAGE BY IRINA VINNIKOVA

flag unexpected code patterns. These tools are a good first line of defense against security vulnerabilities: they are fast and can flag many shallow bugs. Unfortunately, they are also prone to report false alarms and they do not catch every bug. Indeed, static analysis tools are typically unsound and incomplete in practice in order to be fast and automatic.

Manual code inspection consists in peer-reviewing code before releasing it. It is part of most software-development processes and can detect serious bugs. Penetration testing, or pen testing for short, is a form of manual code inspection where security experts review code (as well as design and architecture) with a specific focus on security. Pen testing is flexible, applicable to any software, easy to start (not much tooling required), and can reveal design flaws and coding errors that are beyond the reach of automated tools. But pen testing is labor-intensive, expensive, and does not scale well since (good) pen testers are specialized and in high demand.

Fuzzing is the third main approach for hunting software security vulnerabilities. Fuzzing repeatedly executes an application with all kinds of input variants with the goal of finding security bugs, like buffer-overflows or crashes. Fuzzing requires test automation, that is, the ability to execute tests automatically. It also requires each test to run fast (typically in a few seconds at most) and the application state to be reset after each iteration. Fuzzing is therefore more difficult to set up when testing complex distributed applications, like cloud or server applications running on multiple machines. In practice, fuzzing is usually most effective when applied to standalone applications with large complex data parsers. For each bug found, fuzzing provides one or several concrete inputs that can be used to reproduce and examine the bug. Compared to static analysis, fuzzing does not generate false alarms, but it is more computationally expensive (running for days or weeks) and it can also miss bugs.

Over the last two decades, fuzzing has been shown to be remarkably ef-

Figure 1. How secure is your JPEG parser?

fective in finding security vulnerabilities, often missed by static program analysis and manual code inspection. In fact, fuzzing is so effective that it is now becoming standard in commercial software development processes. For instance, the Microsoft Security Development Lifecycle[21] requires fuzzing at every untrusted interface of every product. To satisfy this requirement, much expertise and tools have been developed since around 2000.

This article presents an overview of these techniques, starting with simple techniques used in the early fuzzing days, and then progressively moving on to more sophisticated techniques. I also discuss the strengths and limitations of each technique.

Note this article is not an overview of the broader areas of automatic test generation, search-based software testing, program verification, or other applications of fuzzing techniques beyond security testing. When it was first introduced,[8] the term fuzz testing simply meant feeding random inputs to applications, without a specific focus on security. However, today, fuzzing is commonly used as a shorthand for security testing because the vast majority of its applications is for finding security vulnerabilities. Indeed, fuzzed inputs are often improbable or rather harmless unless they can be triggered and controlled by an attacker who can exploit them to deliberately break into a system and cause significant damage.

Blackbox Fuzzing

The first and simplest form of fuzzing is *blackbox random fuzzing*, which randomly mutates well-formed application inputs, and then tests the application with these modified inputs.[8]

Figure 2 shows a simple program for blackbox random fuzzing. The program takes as input a well-formed `input seed` (line 1). It then chooses a random number of bytes that will be fuzzed in that input (line 2). That number `numWrites` varies from 1 to the length of the `seed` input divided by 1,000. This arbitrary 1,000 value is optional, but it prevents fuzzing too many bytes in the original `seed`. Next, the loop of lines 4–8 repeatedly selects a random location `loc` in the input (line 5) and a new random byte `value` (line 6) that is then written at that location (line 7), until the selected number `numWrites` of bytes have been fuzzed. The program then executes the application under test with that `newInput` (line 9), and reports an error if a bug is detected (line 10). In practice, the application is being run under the monitoring of a runtime checking tool (like Purify, Valgrind, AppVerifier or AddressSanitizer) in order to increase the chances of finding non-crashing security vulnerabilities (like buffer overflows).

The program of Figure 2 can be repeatedly executed to generate as many new fuzzed inputs as the user wants. Despite its simplicity, this fuzzing strategy can already be effective in finding security vulnerabilities in applications, especially if they have never been fuzzed before and if they process binary input formats. Indeed, binary formats, like the JPEG image format used for Figure 1, typically use raw byte values to encode key input properties, like image sizes, dimensions, and input-file data pointers; fuzzing these key byte values (whose locations vary from image to image) in otherwise well-formed inputs may already reveal buffer-overflow bugs due to incomplete input validation.

In practice, the effectiveness of blackbox random fuzzing crucially depends on a diverse set of well-formed seed inputs to start the fuzzing process. Indeed, well-formed seed inputs will exercise more code more quickly in the application to be fuzzed, covering various options and encodings supported by the input format. In contrast, fuzzing without well-formed seed inputs will very likely generate pure garbage, which the application under test will quickly detect and discard. This is why the program of Figure 2 defines the constant 1,000 in line 2 as its fuzzing density: if every byte in a seed input was fuzzed, the new input generated would be completely garbled and random; but if at most one byte every 1,000 bytes is fuzzed on average, fuzzing adds only limited noise to the original seed input, and testing with this slightly corrupted new input is more likely to exercise more error-handling code in more diverse parts of the application under test, hence increasing the chances of finding bugs.

Grammar-Based Fuzzing

Blackbox random fuzzing provides a simple fuzzing baseline, but its effectiveness is limited: the probability of generating new interesting inputs is low.[35] This is especially true when fuzzing applications with structured input formats, like XML or JSON dialects: randomly fuzzing inputs in these formats is likely to break key structural properties of the input, which the application will quickly detect in a first lexical analysis and then discard, hence exercising little of the application code.

Grammar-based fuzzing is a powerful alternative for fuzzing complex formats. With this approach, the user

provides an input grammar specifying the input format of the application under test. Often, the user also specifies what input parts are to be fuzzed and how. From such an input grammar, a grammar-based fuzzer then generates many new inputs, each satisfying the constraints encoded by the grammar. Examples of grammar-based fuzzers are Peach,[29] SPIKE,[32] and Sulley,[34] among others.[35]

Figure 3 shows a code fragment representing input constraints which a grammar-based fuzzer like SPIKE uses to generate new inputs. The input grammar is represented here directly using code, which can be interpreted, for example, in Python. In line 2, the user specifies with a call to s_string that a constant string with the fixed value POST /api/blog/ HTTP/1.2 should be generated first. Then another constant string Content-Length: should be appended (line 3). The call to s_blocksize_string in line 4 adds a string of length 2 (second argument) with the size of the block named blockA (first argument) as value. A call to s_block_start defines the start of a block (line 5), while s_block_end denotes the end of a block (line 9), whose name is specified in the argument. In line 7, the user specifies with a call to s_string_variable that a fuzzed string is to be appended at this location; this string can be the constant XXX specified in the call or any other string value taken out of a user-defined dictionary of other values (not shown here). Tools like SPIKE support several ways of defining such fuzzing dictionaries as well as custom fuzzing rules (for example,for numeric values). By executing the code shown in Figure 3, SPIKE might generate this string sequence (shown here on 2 lines):

```
POST /api/blog/ HTTP/1.2
Content-Length:10{body:XXX}
```

This approach is very general: the user can specify how to generate input strings using nearly arbitrary code, including function recursion to generate hierarchies of well-balanced delimiters, like { and } in the example here, and strings of various sizes.

Grammar-based fuzzing is very powerful: the user expertise is used to focus and guide fuzzing toward specific input corner cases of interest, which would never be covered with blackbox random fuzzing in practice. Sophisticated grammar-based fuzzers exist for finding security vulnerabilities in Web browsers,[19] which must take as untrusted inputs Web pages including complex HTML documents and JavaScript code, as well as for finding complex bugs in compilers.[38] Grammar-based fuzzing also works well for network-protocol fuzzing where sequences of structured messages need to be fed to the application under test in order to get good code coverage and find bugs.[31]

Work on grammar-based test input generation can be traced back to the 1970s.[17] Test generation from a grammar is usually either done using random traversals of the production rules of a grammar,[26] or is exhaustive and covers all its production rules.[24] Imperative generation[6] is a related approach in which a custom-made program generates the inputs (in effect, the program encodes the grammar), as shown in Figure 3.

Grammar-based fuzzing is also related to model-based testing.[36] Given an abstract representation of a program—called a model—model-based testing consists in generating tests by analyzing the model in order to check the conformance of the program with respect to the model. Test generation algorithms used in model-based testing often try to generate a minimum number of tests covering, say, every state and transition of a finite-state machine model in order to generate test suites that are as small as possible. Similar algorithms can be used to cover all production rules of a grammar without exhaustively enumerating all possible combinations.

How to automatically learn input grammars from input samples for fuzzing purposes is another recent line of research. For instance, context-free grammars can be learned from input examples using custom generalization steps,[1] or using a dynamic taint analysis of the program under test in order to determine how the program processes its inputs.[20] Statistical machine-learning techniques based on neural networks can also be used to learn probabilistic input grammars.[16] While promising, the use of machine learning for grammar-based fuzzing is still preliminary and not widely used today.

In summary, grammar-based fuzzing is a powerful approach to fuzzing that leverages the user's expertise and creativity. Unfortunately, grammar-based fuzzing is only as good as the input grammar being used, and writing input grammars by hand is laborious, time consuming, and error-prone. Because the process of writing gram-

Figure 2. Sample blackbox fuzzing code.

```
1 RandomFuzzing(input seed) {
2   int numWrites = random(len(seed)/1000)+1;
3   input newInput = seed;
4   for (int i=1; i<=numWrites; i++) {
5        int loc = random(len(seed));
6        byte value = (byte)random(255);
7        newInput[loc] = value;
8   }
9   result = ExecuteAppWith(newInput);
10  if (result == crash) print("bug found!");
11  }
```

Figure 3. Sample SPIKE fuzzing code.

```
1  ...
2  s_string("POST /api/blog/ HTTP/1.2 ");
3  s_string("Content-Length: ");
4  s_blocksize_string("blockA", 2);
5  s_block_start("blockA");
6  s_string("{body:");
7  s_string_variable("XXX");
8  s_string("}");
9  s_block_end("blockA");
10 ...
```

mars is so open-ended and there are so many possibilities for fuzzing rules (what and how to fuzz), when to stop editing a grammar further is another practical issue.

Whitebox Fuzzing

Blackbox random fuzzing is practically limited, and grammar-based fuzzing is labor intensive. Moreover, when can one safely stop fuzzing? Ideally, further testing is not required when the program is formally verified, that is, when it is mathematically proved not to contain any more bugs.

Cost-effective program verification has remained elusive for most software despite 40+ years of computer-science research.[18,22] However, significant advances in the theory and engineering of program analysis, testing, verification, model checking, and automated theorem proving have been made over the last two decades. These advances were possible in part thanks to the increasing computational power available on modern computers, where sophisticated analyses have now become more affordable. Whitebox fuzzing is one of these advances.

Starting with a well-formed input, *whitebox fuzzing*[14] consists of symbolically executing the program under test *dynamically*, gathering constraints on inputs from conditional branches encountered along the execution. The collected constraints are then systematically negated one-by-one and solved with a constraint solver, whose solutions are mapped to new inputs that exercise different program execution paths. This process is repeated using systematic search techniques that attempt to sweep through all (in practice, many) feasible execution paths of the program while checking simultaneously many properties (like buffer overflows) using a runtime checker.

For example, consider this simple program:

```
int foo (int x) { // x is an input
 int y = x + 3;
 if (y == 13) abort (); // error
 return 0;
}
```

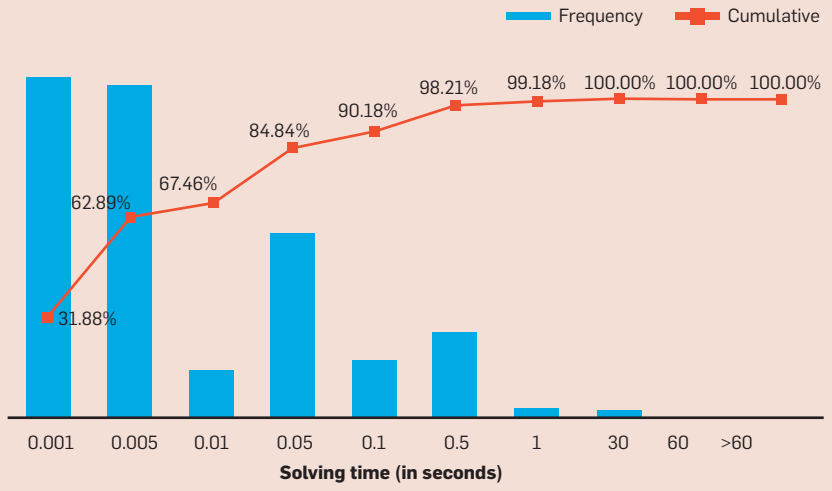
Dynamic symbolic execution of this program with an initial concrete value 0 for the input variable x takes the else branch of the conditional statement, and generates the path constraint $x + 3 \neq 13$. After negating this input constraint and solving it with a constraint solver,[7] the solver produces a solution $x = 10$. Running the program with this new input causes the program to follow the then branch of the conditional statement and finds the error. Note that blackbox random fuzzing has only 1 in 2^{32} chances of exercising the then branch if the input variable x has a randomly-chosen 32-bit value, and it will never find the error in practice. This intuitively explains why whitebox fuzzing usually provides higher code coverage.

Whitebox fuzzing can generate inputs that exercise more code than other approaches because it is more precise.[11] It can therefore find bugs missed by other fuzzing techniques, even without specific knowledge of the input format. Furthermore, this approach automatically discovers and tests code corner cases where programmers may fail to properly allocate memory or manipulate buffers, leading to security vulnerabilities. Note that full program statement coverage is a necessary but not sufficient condition to find all the bugs in a program.

In theory, exhaustive whitebox fuzzing provides full program path coverage, that is, program verification (for any input up to a given size). The simple program foo has two feasible execution paths, which can be exhaustively enumerated and explored in order to prove that that this program does not contain any buffer overflow. In practice, however, the search is typically incomplete because the number of execution paths in the program under test is huge, and because symbolic execution, constraint generation, and constraint solving may be imprecise due to complex program statements (pointer manipulations, floating-point operations, among others), calls to external operating system and library functions, and large numbers of constraints which cannot all be solved perfectly in a reasonable amount of time. Because of these limitations, whitebox fuzzing, like blackbox fuzzing, still relies on a diverse set of seed inputs to be effective in practice.

Whitebox fuzzing was first implemented in the tool SAGE,[14] and extends the scope of prior work on dynamic test generation,[4,13] also called execution-generated tests or concolic testing, from unit testing to security testing of large programs. SAGE performs dynamic symbolic execution at the x86 binary level, and implements several optimizations that are crucial for dealing with huge execution traces with hundreds of millions of machine instructions, in order to scale to large file parsers embedded in applications with millions of lines of code, like Microsoft Excel or PowerPoint. SAGE also uses search heuristics based on code coverage when exploring large state spaces: instruction coverage is measured for every test executed, and tests that discover many new instructions

Figure 4. Most constraints are easy to solve.



are symbolically executed with higher priority so that their unexplored neighborhoods are explored next. Tests and symbolic executions can be run in parallel, on multiple cores or machines. Like blackbox fuzzing, whitebox fuzzing starts with a small diverse set of seed inputs, whenever possible, in order to give its search a head start. Such seed inputs can also be generated using grammar-based fuzzing when an input grammar is available.[25]

Since 2008, SAGE has been running in production for over 1,000 machine-years, automatically fuzzing hundreds of applications.[2] This is the "largest computational usage ever for any Satisfiability-Modulo-Theories (SMT) solver" according to the authors of the Z3 SMT solver,[7] with around 10 billion constraints processed to date. On a sample set of 130 million constraints generated by SAGE while fuzzing 300 Windows applications, Figure 4 shows that about 99% of all constraints are solved by Z3 in one second or less.[2]

During all this fuzzing, SAGE found many new security vulnerabilities (buffer overflows) in hundreds of Windows parsers and Office applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7,[15] saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide. Because SAGE was typically run last, these bugs were missed by everything else, including static program analysis and blackbox fuzzing.

Today, whitebox fuzzing has been adopted in many other tools. For instance, the top finalists of the DARPA Cyber Grand Challenge,[37] a competition for automatic security vulnerability detection, exploitation and repair, all included some form of whitebox fuzzing with symbolic execution and constraint solving in their solution. Other influential tools in this space include the open-source tools KLEE,[3] S2E,[5] and Symbolic PathFinder.[28]

Fuzzing is commonly used as a shorthand for security testing because the vast majority of its applications is for finding security vulnerabilities.

Other Approaches

Blackbox random fuzzing, grammar-based fuzzing and whitebox fuzzing are the three main approaches to fuzzing in use today. These approaches can also be combined in various ways.

Greybox fuzzing extends blackbox fuzzing with whitebox fuzzing techniques. It approximates whitebox fuzzing by eliminating some of its components with the goal of reducing engineering cost and complexity while retaining some of its intelligence. AFL[40] is a popular open source fuzzer which extends random fuzzing with code-coverage-based search heuristics as used in SAGE, but without any symbolic execution, constraint generation or solving. Despite (or because) of its simplicity, AFL was shown to find many bugs missed by pure blackbox random fuzzing. AFL is related to work on search-based software testing[27] where various search techniques and heuristics (such as genetic algorithms or simulated annealing) are implemented and evaluated for various testing scenarios. Another form of greybox fuzzing is *taint-based fuzzing*,[10] where an input-taint analysis is performed to identify which input bytes influence the application's control flow, and these bytes are then randomly fuzzed, hence approximating symbolic execution with taint analysis, and approximating constraint generation and solving with random testing.

Hybrid fuzzing[33,39] combines blackbox (or greybox) fuzzing techniques with whitebox fuzzing. The goal is to explore trade-offs to determine when and where simpler techniques are sufficient to obtain good code coverage, and use more complex techniques, like symbolic execution and constraint solving, only when the simpler techniques are stuck. Obviously, many trade-offs and heuristics are possible, but reproducible statistically-significant results are hard to get.[23] Grammar-based fuzzing can also be combined with whitebox fuzzing.[12,25]

Portfolio approaches run multiple fuzzers in parallel and collect their results, hence combining their complementary strengths. Project Springfield[30] is the first commercial cloud fuzzing service (renamed Microsoft Security Risk Detection in 2017), and uses a portfolio approach. Customers who subscribe to this service can submit fuzzing jobs targeting their own software. Fuzzing jobs are processed by

creating many virtual machines in the cloud and by running different fuzzing tools and configurations on each of these machines. Fuzzing results (bugs) are continually collected by the service and post-processed for analysis, triage and prioritization, with final results available directly to customers on a secured website.

Conclusion

Is fuzzing a hack, an art, or a science? It is a bit of all three. Blackbox fuzzing is a simple *hack* but can be remarkably effective in finding bugs in applications that have never been fuzzed. Grammar-based fuzzing extends it to an *art*[a] form by allowing the user's creativity and expertise to guide fuzzing. Whitebox fuzzing leverages advances in computer *science* research on program verification, and explores how and when fuzzing can be mathematically "sound and complete" in a proof-theoretic sense.

The effectiveness of these three main fuzzing techniques depends on the type of application being fuzzed. For binary input formats (like JPEG or PNG), fully-automatic blackbox and whitebox fuzzing techniques work well, provided a diverse set of seed inputs is available. For complex structured non-binary formats (like JavaScript or C), the effectiveness of blackbox and whitebox fuzzing is unfortunately limited, and grammar-based fuzzing with manually-written grammars are usually the most effective approach. For specific classes of structured input formats like XML or JSON dialects, domain-specific fuzzers for XML or JSON can also be used: these fuzzers parse the high-level tree structure of an input and include custom fuzzing rules (like reordering child nodes, increasing their number, inversing parent-child relationships, and so on) that will challenge the application logic while still generating syntactically correct XML or JSON data. Of course, it is worth emphasizing that no fuzzing technique is guaranteed to find all bugs in practice.

What applications should be fuzzed also depends on a number of parameters. In principle, any application that may process untrusted data should be fuzzed. If the application runs in a critical environment, it should definitely be fuzzed. If the application is written in low-level code like C or C++, the danger is even higher, since security vulnerabilities are then typically easier to exploit. If the application is written in higher-level managed code like Java or C#, fuzzing might reveal unhandled exceptions which may or may not be security critical depending on the context (service-side code is usually more critical).

Despite significant progress in the art and science of fuzzing over the last two decades, important challenges remain open. How to engineer exhaustive symbolic testing (that is, a form of verification) in a cost-effective manner is still an open problem for large applications. How to automate the generation of input grammars for complex formats, perhaps using machine learning, is another challenge. Finally, how to effectively fuzz large distributed applications like entire cloud services is yet another open challenge. **C**

a Art is "the expression or application of human creative skill and imagination."

References

1. Bastani, O., Sharma, R., Aiken, A. and Liang, P. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2017, 95–110.
2. Bounimova, E., Godefroid, P., and Molnar, D. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of 35th Intern. Conf. Software Engineering*, (San Francisco, May 2013), 122–131.
3. Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI'08* (Dec 2008).
4. Cadar, C. and Engler, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of 12th Intern. SPIN Workshop on Model Checking of Software 3639* (San Francisco, CA, Aug. 2005) Lecture Notes in Computer Science, Springer-Verlag.
5. Chipounov, V., Kuznetsov, V. and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of ASPLOS'2011*.
6. Claessen, K. and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP'2000*.
7. de Moura, L. and Bjorner, N. Z3: An Efficient SMT Solver. In *Proceedings of 14th Intern. Conf. Tools and Algorithms for the Construction and Analysis of Systems 4963* (Budapest, April 2008), 337–340. Lecture Notes in Computer Science, Springer-Verlag.
8. Forrester, J.E. and Miller, B.P. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symp.*, Seattle, (Aug. 2000).
9. Gallagher, T., Jeffries, B., and Landauer, L. *Hunting Security Bugs.* Microsoft Press, 2006.
10. Ganesh, V., Leek, T., and Rinard. M. Taint-based directed whitebox fuzzing. In *Proceedings of ICSE '2009.*
11. Godefroid, P. Higher-order test generation. In *Proceedings of ACM SIGPLAN 2011 Conf. Programming Language, Design and Implementation* (San Jose, June 2011). 258–269.
12. Godefroid, P., Kiezun, A., and Levin, M.Y. Grammar-based whitebox fuzzing. In *Proceedings of ACM SIGPLAN 2008 Conf. Programming Language Design and Implementation*, (Tucson, AZ, USA, June 2008), 206–215.
13. Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *Proceedings of ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation* (Chicago, IL, June 2005). 213–223.
14. Godefroid, P., Levin, M.Y., and Molnar, D. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security* (San Diego, Feb. 2008), 151–166.
15. Godefroid, P., Levin, M.Y., and Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM 55*, 3 (Mar. 2012), 40–44.
16. Godefroid, P., Peleg, H., and Singh, R. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM Intern. Conf. Automated Software Engineering*, Nov. 2017.
17. Hanford, K.V. Automatic generation of test cases. *IBM Systems J. 9*, 4 (1970).
18. Hoare, C.A.R. An axiomatic approach to computer programming. *Commun. ACM 12*, 10 (1969), 576–580.
19. Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In Proceedings of the 21st USENIX Security Symp., 2012.
20. Höschele, M. and Zeller, A. Mining input grammars with AUTOGRAM. In *Proceedings of ICSE-C'2017*, 31–34.
21. Howard, M. and Lipner, S. *The Security Development Lifecycle.* Microsoft Press, 2006.
22. King, J.C. Symbolic execution and program Ttesting. *J. ACM 19*, 7 (1976), 385–394.
23. Klees, G.T., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the ACM Conf. Computer and Communications Security*, 2018.
24. Lämmel, R. and Schulte, W. Controllable combinatorial coverage in grammar-based testing. In *Proceedings of TestCom*, 2006.
25. Majumdar, R. and Xu, R. Directed test generation using symbolic grammars. In *Proceedings of ASE*, 2007.
26. Maurer, P.M. Generating test data with enhanced context-free grammars. *IEEE Software 7*, 4 (1990).
27. McMinn, P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability 14*, 2 (2004).
28. Pasareanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., and Rungta, N. Symbolic pathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 2013, 20:391–425.
29. Peach Fuzzer; http://www.peachfuzzer.com/.
30. Project Springfield; https://www.microsoft.com/springfield/, 2015.
31. Protos; http://www.ee.oulu.fi/research/ouspg/protos/.
32. SPIKE Fuzzer; http://resources.infosecinstitute.com/fuzzer-automation-with-spike/.
33. Stephens, N. et al. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of Network and Distributed Systems Security*, 2016.
34. Sulley; https://github.com/OpenRCE/sulley.
35. Sutton, M., Greene, A., and Amini, P. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.
36. Utting, M., Pretschner, A., and Legeard, B. A Taxonomy of model-based testing approaches. *Intl. J. Software Testing, Verification and Reliability 22*, 5 (2012).
37. Walker, M. et al. DARPA Cyber Grand Challenge, 2016; http://archive.darpa.mil/cybergrandchallenge/.
38. Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of PLDI'2011*.
39. Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symp.*, 2018.
40. Zalewski, M. AFL (American Fuzzy Lop), 2015; http://lcamtuf.coredump.cx/afl/

Patrice Godefroid (pg@microsoft.com) is a partner researcher at Microsoft Research, Redmond, WA, USA.

Watch the author discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/fuzzing