Assuring Software Quality Through the Use of Fuzzy Testing

David Jefts
SE625, Software Quality Assurance

Embry-Riddle Aeronautical University

***Abstract*—THIS IS NOT MY ABSTRACT Software engineering researchers solve problems of several different kinds. To do so, they produce several different kinds of results, and they should develop appropriate evidence to validate these results. They often report their research in conference papers. I analyzed the abstracts of research papers submitted to ICSE 2002 in order to identify the types of research reported in the submitted and accepted papers, and I observed the program committee discussions about which papers to accept. This report presents the research paradigms of the papers, common concerns of the program committee, and statistics on success rates. This information should help researchers design better research projects and write papers that present their results to best advantage.**

This paper makes the case for TaaS—automated software testing as a cloud-based service. We present three kinds of TaaS: a "programmer's sidekick" enabling developers to thoroughly and promptly test their code with minimal upfront resource investment; a "home edition" on-demand testing service for consumers to verify the software they are about to install on their PC or mobile device; and a public "certification service," akin to Underwriters Labs, that independently assesses the reliability, safety, and security of software.

***Index Terms*—Fuzzy Testing, Fuzz Testing, Fuzzing, Smart Fuzzing, Fuzzers, Software Quality Assurance, Quality Assurance, Software Vulnerabilities, Software Reliability, Testing, Automated Testing, Error Detection, Software Errors, Debugging, Computer Bugs, Computer Security, Testing, Product Testing, System Testing**

NOMENCLATURE

AI	Artificial Intelligence
CPU	Computer Processing Unit
CVE	Common Vulnerabilities and Exposures
FC	Fuzzy Clustering
FDA	United States Food and Drug Administration
GA	Genetic Algorithm
IM	Instant Messaging
OSS	Open Source Software
RISC	Reduced Instruction Set Computer
SAGE	Scalable, Automated, Guided Execution
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VPN	Virtual Private Network

I. INTRODUCTION

ENSURING that software works properly before public (or private) release is important because software bugs, or in-code errors, can cost the developing company a lot of money. Software bugs cause program faults, faults open up security vulnerabilities or cause program failures, and the maintenance and repair of software is 65-85% of system development and deployment cost [1]. Naturally, the goal during software development is to limit these defects by testing the software before release. However, "software testing is a very labor intensive and costly task, [even though] many software testing techniques to automate the process of software testing have been [designed and] reported in the literature" [2]. One such technique is Fuzzy Testing, also called Fuzzing.

Fuzzy testing is an important topic to study because it offers an automated way to test software that is mostly reliable. What is most interesting about it is that it incorporates many different fields from the software and computer fields: Software Development/Engineering is the main application of Fuzzy Testing, Software Quality Assurance is the entire goal of Fuzzy Testing, Artificial Intelligence and Machine Learning are being used to help optimize various fuzzy testing techniques, and Software Penetration and Security Testing also uses Fuzzy Testing to "verify security functionality" [3].

This paper seeks to summarize much of the leading-edge research on fuzzy testing. It makes the case for a more proliferated use of fuzzy testing in the software industry and promotes additional research for better ways to optimize fuzzy testing. "Automated software testing, available to anyone and everyone at low cost, can transform the current development paradigm into one that involves more-thorough yet less-time-consuming testing" [4]. By promoting fuzzy testing as a whole, along with further development of fuzzy testing, automated software testing can be made more reliable and easier to access.

The rest of this paper is organized as follows. The general types of software testing and their breakdowns are discussed in Section II. In Section III, Fuzzy Testing and its real-world applications are presented. Section IV describes many of the current uses of fuzzy testing in real-world applications, and Section V enumerates many public security vulnerabilities and threats that were found with fuzzy testing. This paper presents the argument for the adoption of Fuzzy Testing in Section VI. All conclusions and future projects are discussed in Section VII.

II. SOFTWARE TESTING

There are two main 'sectors', or approaches, to software testing: white-box testing and black-box testing [5]. White Box Testing, also called Structural Testing [6], is the testing

of a system *with* knowledge of the internal systems software. Testers in this paradigm "have access to the source code and are aware of the system architecture" [5]. Generally, White Box testers analyze the source code of the software and develop test cases and identify specific code paths to achieve the most raw code coverage. They develop test cases with respect to executed statements, branches, paths, and so forth. "After initial testing, programmers frequently face the problem of finding additional test data to evaluate program elements not yet covered" [7]. Determining the test data required to evaluate 100% of the software is often very labor-intensive and expensive. Test generators in the White Box testing paradigm work to find program input(s) on which a selected portion of the software is executed. According to [7], there are currently three types of automatic generators for this: *random* test generators, *path-oriented* generators, and *goal-oriented* generators. White Box testing generally finds more bugs and defects as compared to Black Box Testing. However it requires more time and money than Black Box Testing, since it often requires a direct analysis of the source code and tests must cover 100% of the code.

Black Box Testing, also called Functional Testing [6], is the testing of a system with no knowledge or direct access of the internal systems in the software. Testers in this paradigm "do not have access to the source code and are oblivious of the system architecture" [5], and select test data "based on the functional specification of the program (e.g., equivalence class partitioning, boundary-value analysis)" [7]. Black Box testers just test the software through the provided user interface and analyzing inputs and outputs for correctness. Test generators working off of the Black Box paradigm create test cases and test data by "using a set of rules and procedures; the most popular methods include equivalence class partitioning, boundary value analysis, [and] cause-effect graphing" [7]. "By their nature Black Box testing methods might not lead to the execution of all parts of the code. Therefore, this method may not uncover all faults in the program" [2].

Grey Box Testing is a combination of White Box and Black Box Testing, which aims to take advantage of each of the aforementioned testing paradigms while minimizing their drawbacks. It generally uses only a minimal knowledge of the program structure and behavior. It does not require access to the source code and analyzes the software from an external perspective. Generally the tester will know how the internal system components interact, such as the internal data structures and algorithms [8], but will not have a full and detailed knowledge about the internal program functions and/or operation. It offers a clear distinction between "the developer and tester, thereby minimizing the risk of personnel conflicts" [8].

Assertion-Based testing is reportedly a largely effective software testing technique [7]. It involves finding program inputs in which various assertions are violated. When these assertions are violated then there is a fault in the program [7]. Each assertion specifies "a constraint that applies to some state of a computation" [7], for example making sure that a variable holds valid data. Some languages support this by default for example Java and Perl, while others have libraries that . According to [2] it is fairly easy to generate assertion tests; "creating boundary checks, division by zero, null pointers, and variable overflow/underflow" [2]. The problem with this method of software testing is that the implementation of large amounts of assertions can be costly and is often impractical for industrial-sized software. Additionally, finding program inputs that cover all assertions in the software can be difficult and time consuming.

Fuzzy Testing can fall under White Box, Black Box, or a combination of both depending on how aware the tester is of the program structure. Fuzzy Testing is the process of submitting "malformed, malicious, and random data to a system's entry points [or user interface] in an attempt to uncover faults" [3]. "The tool then reports the faults to the tester for further analysis" [3]. Fuzzing may also be described as "a security testing approach based on injecting invalid or random inputs into a program in order to obtain an unexpected behavior and identify errors and potential vulnerabilities" [9]. This method is generally much cheaper and easier to utilize than the others since it is designed to be automated, but it suffers from many of the same drawbacks as Black Box testing- namely that it does not guarantee full code coverage and therefore cannot guarantee the software is bug free.

III. Fuzzy Testing

A. Techniques

Fuzzy Testing is a negative-testing technique, used to find software bugs, faults, and vulnerabilities [10]. While Fuzzy Testing is generally performed entirely automatically, it is also possible to perform semi-automated or even completely manual fuzzing [11]. Manual testing involves finding data fields and modifying them to see if they break. Semi-automatic uses small scripts that are run individually to test the software. This paper is focused on fully automatic fuzzy testers, that use a script or program to iterate over an extensive list of possible inputs until the program crashes or has a fault [12]. These inputs can be generated completely from scratch (random), in a brute-force manner (sequential generation of all possible input combinations), or with data mutation (capturing valid input data and mutating it, either randomly or in a brute-force manner) [13]. These random inputs can be generated in a variety of ways but some more efficient fuzzers use genetic algorithms to 'intelligently' generate test data [14]. All fuzzers share a similar set of features [13]:

- data generation (creating data to be passed to the target);
- data transmission (getting the data to the target);
- target monitoring and logging (observing and recording the reaction of the target), and;
- automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime).

Fuzzy testing is similar to stress testing, which involves sending an excessive amount of bogus data to a software or digital service with the goal of making it crash, to determine its stability and ensure that it is stable under normal operating conditions or to determine the limits of its use.

B. Applications

Currently, the most popular application of fuzzy testing is for exposing security vulnerabilities in software products, either by hackers or software developers [15]. This is because fuzzers are often built to generate large amounts of potentially corrupt data in a relatively short time, allowing one to test a plethora of different edge cases that the original programmer may not have accounted for. Fuzzers are generally fairly effective at finding regular software bugs in applications that have a user input. The mass-production of random data is useful in finding edge-cases, corner cases, and boundary cases. Edge cases are software states that are generally only observed at extreme operation parameters (e.g. a stereo speaker operating at or beyond its maximum volume). Corner cases occur outside of normal operating parameters, when multiple variables and/or conditions are at extreme, albeit valid, levels. Boundary cases are software instances where an input, variable, or condition is at or beyond its maximum limits.

"Anyone who has access to an application can fuzz it. Access to the source code is not required" [13]. It requires very little expertise to identify basic bugs, as compared to White Box testing and Assertion-Based testing. Additionally, implementation is fast due to a wide variety of fuzzing applications and can take only a few minutes in some cases. Developers can apply fuzzy testing for vulnerability discovery and resolution during the entirety of the software development life-cycle. "Software vendors such as Cisco, Microsoft, Juniper, AT&T, and Symantec all employ fuzzing as a matter of course" [13]. End-users, small- to medium-sized enterprises, and corporations can all take advantage of fuzzers to use as a software quality assurance tool. Virtual attackers and hackers may also use fuzzing to identify and inject malicious data or code into Internet software to gain unintended access and exploit on-line systems.

IV. Large-Scale and High-Profile Utilizations of Fuzzing

A. The Monkey

First developed around 1983, The Monkey was one of the first tools to utilize fuzzy testing [10]. "The Monkey was a small desk accessory that used the journaling hooks to feed random events to the current application, so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon" [16].

B. U.S. Food and Drug Administration Medical Cybersecurity Laboratory

One popular use of fuzzy testing was by the FDA, it "adopted a generational fuzzer as the foundation of its newly created cybersecurity laboratory, in which medical devices will be tested with the aim of improving safety and reliability" [17]. In July of 2014 the FDA reported that the first tool its lab would use was Defensics, a fuzz testing platform created by Codenomicon [18].

C. ClusterFuzz

On April 26th, 2012, Google announced their cloud-based fuzzing infrastructure for finding security-critical components for Google's Chromium and Chrome web browsers [19]. According to [19], it is highly scalable, has accurate deduplication of crashes, minimizes test cases, supports blackbox fuzzing, and offers statistics and regression finding for analysis of fuzzer performance.

D. Project Springfield

Microsoft announced its own cloud-based bug detector based off of SAGE, called Project Springfield, in 2016 [20]. It is "one of the most sophisticated tools [Microsoft] has for rooting out potential security vulnerabilities in software" [20]. Its main goal is to save software developers from having to take the "costly effort" of releasing a patch reactively, after it has already been deployed to the general public.

E. OSS-Fuzz - Continuous Fuzzing for Open Source Software

Released in December 2016, OSS-Fuzz is a fuzz testing tool developed by Google and was mainly used by them to test their Google Chrome web browser [21]. At the time of writing, it supported the C and C++ programming languages and reportedly found "hundreds of security vulnerabilities and stability bugs" [21].

V. Security Threats Discovered with Fuzzy Testing

A. Heartbleed

Officially referred to as 'CVE-2014-0160', the Heartbleed Bug is a security vulnerability in the OpenSSL cryptographic software library. It allows the stealing of information normally protected by the SSL/TLS encryption used to secure the internet [22]. "SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM), and some virtual private networks (VPNs)" [22]. It reveals the memory of systems normally protected by SSL/TLS to anyone who wishes to see it, allowing malicious people to gather names and passwords, eavesdrop on communications, steal data directly from services and users, and impersonate services and users. In April 2015, Hanno Böck revealed that this bug could have been easily discovered with a "reasonably simple fuzzing setup ...[that] doesn't require any prior knowledge about specifics of the Heartbleed bug or the TLS Heartbeat extension" [23].

B. GOD MODE UNLOCKED - Hardware Backdoors

In July of 2018, Christopher Domas revealed a hardware backdoor, colloquially called the *rosenbridge*, in x86 CPUs developed by VIA Technologies, Inc. [24]. Domas' backdoor "offers ring 3 (userland) to ring 0 (kernel) privilege escalation, providing a well-hidden, devastating circumvention to the long-standing x86 ring privilege model, wherein untrusted code is effectively separated from the heart of the system" [24]. Domas surmised that the VIA Technologies Inc., C3 family of

x86 processors had a second *deeply embedded core* that was a RISC processor tightly connected to the main processor core. In [24], Domas used a processor fuzzing tool called *Sandsifter* to help identify the *god mode bit* to the RISC core. He also used *Sandsifter* to find the *bridge instruction* used to send and execute commands on the main x86 processor.

VI. A Fuzzy Testing Comparison

A. Test Data Generation

The three methods of test data generation listed in Section III-A each have their own advantages and disadvantages.

1) Random Data: Often referred to as 'blind-fuzzing', this approach is desirable because it requires very minimal effort to initiate testing, and assumptions do not restrict the scope of testing [13]. However, random test generation does not guarantee the generation of the combination(s) required to cause a failure.

2) Brute-Force Data: This method involves sequentially building every possible permutation of the input space. Like the random data generation, this method requires no knowledge of the target system, with the exception that the size of the input space should be known in order to limit the amount of data generated [13]. Brute-Force test data generation falls apart when there is a large input space, as it is very inefficient and will require more storage and/or time in order to exhaust every possible input. "In order to brute force fuzz all values of a 32-bit integer, a total of 4,294,967,295 test instances would be required. Disregarding the time and space required to generate and store this test data, it would take 500 days to process each of the required test instances assuming it would take one hundredth of a second to process each one" [13].

3) Data Mutation: This method involves capturing valid software input data while the software is in use, then modifying and mutating the data (either random or brute-force). The data capture step is generally basic and fast and due to the similarity to valid input, the generated test data will have a much higher efficiency [13]. This allows the mutation method to have the advantages of random and brute-force test data: minimal required effort and knowledge of the data, while avoiding all of the disadvantages: poor code coverage and efficiency [13].

Data Generation Method	Finite test data	Requires analysis of application	Likely to produce valid static numbers	Likely to maintain data structure	Likely to produce valid CRCs
Random generation	N	N	N	N	N
Brute force generation	Y	N	N	N	N
Data mutation (random)	N	N	Y	Y	N
Data mutation (brute force)	Y	N	Y	Y	N
Analysis-based data generation	Y	Y	Y	Y	Y

Figure 1. Comparison of test data generation approaches

B. Other Testing Techniques

bleh

VII. Conclusion

REFERENCES

[1] M. Towhidnejad, "Lecture 4 - defects," February 2019, in-class presentation.

[2] A. M. Alakeel, "Using fuzzy logic techniques for assertion-based software testing metrics," *The Scientific World Journal*, vol. 2015, 2015, copyright - Copyright © 2015 Ali M. Alakeel. Ali M. Alakeel et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited; Last updated - 2018-10-09. [Online]. Available: http://search.proquest.com.ezproxy.libproxy.db.erau.edu/docview/1679858177?accountid=27203

[3] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security Privacy*, vol. 3, no. 1, pp. 84–87, Jan 2005.

[4] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 155–160. [Online]. Available: http://doi.acm.org.ezproxy.libproxy.db.erau.edu/10.1145/1807128.1807153

[5] K. K. Mohan, A. K. Verma, and A. Srividya, "Software reliability estimation through black box and white box testing at prototype level," in *2010 2nd International Conference on Reliability, Safety and Hazard - Risk-Based Technologies and Physics-of-Failure Methods (ICRESH)*, Dec 2010, pp. 517–522.

[6] M. Towhidnejad, "Lecture 6 - software testing," February 2019, in-class presentation.

[7] B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation," in *Proceedings of IEEE 18th International Conference on Software Engineering*, March 1996, pp. 71–80.

[8] (4 November 2011) Gray box testing. [Online]. Available: www.softwaretestingfundamentals.com/gray-box-testing

[9] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 427–430.

[10] A. Takanen, *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Norwood, UNITED STATES: Artech House, 2017. [Online]. Available: http://ebookcentral.proquest.com/lib/erau/detail.action?docID=5430720

[11] I. van Sprundel, "Fuzzing: Breaking software in an automated fashion," 8 December 2005, paper for CCC Conference.

[12] ——, "Fuzzing," December 2005, presentation Slides. [Online]. Available: https://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf

[13] T. Clarke, "Fuzzing for software vulnerability discovery," Royal Holloway, University of London, Tech. Rep. RHUL-MA-2009-04, 17 February 2009.

[14] S. Dalal and S. Hooda, "A novel technique for testing an aspect oriented software system using genetic and fuzzy clustering algorithm," in *2017 International Conference on Computer and Applications (ICCA)*, Sep. 2017, pp. 90–96.

[15] J. Neystadt, "Automated penetration testing with white-box fuzzing," *Microsoft Docs*, 13 October 2009. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)

[16] A. Hertzfeld, "Monkey lives," *Folklore*, 1983. [Online]. Available: www.folklore.org/StoryView.py?story=Monkey_Lives.txt

[17] J. Knudsen, "Practical considerations of fuzzing: Generating insight into areas of risk," *Biomedical Instrumentation & Technology*, vol. 48, pp. 48–53, Spring 2014, copyright - Copyright Allen Press Publishing Services Spring 2014; Document feature; Last updated - 2014-08-15. [Online]. Available: http://search.proquest.com.ezproxy.libproxy.db.erau.edu/docview/1530406192?accountid=27203

[18] B. N. Saddler, "Codenomicon defensics- fuzz testing software," *Department of Health and Human Services - Food and Drug Administration*, 21 July 2013. [Online]. Available: https://www.fbo.gov/spg/HHS/FDA/DCASC/FDA-13-1120705/listing.html

[19] Google. (26 April 2012) Clusterfuzz. Online. [Online]. Available: https://github.com/google/clusterfuzz

[20] Microsoft. (2016) Microsoft previews project springfield, a cloud-based bug detector. Online. [Online]. Available: https://blogs.microsoft.com/ai/microsoft-previews-project-springfield-cloud-based-bug-detector/

[21] Google. (2016) Oss-fuzz - continuous fuzzing for open source software. Online. [Online]. Available: https://github.com/google/oss-fuzz

[22] Synopsys, "The heartbleed bug," 4 April 2014. [Online]. Available: heartbleed.com

[23] H. Böck, "How heartbleed could've been found," *Hanno's blog*, 7 April 2015. [Online]. Available: https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[24] C. Domas, "Hardware backdoors in x86 cpus," *Project : Rosenbridge*, 27 July 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf