Configuration Fuzzing for Software Vulnerability Detection

Huning Dai, Christian Murphy, Gail Kaiser
Department of Computer Science
Columbia University
New York, NY 10027 USA
{*dai, cmurphy, kaiser*}*@cs.columbia.edu*

***Abstract*—Many software security vulnerabilities only reveal themselves under certain conditions, *i.e.*, particular configurations of the software together with its particular runtime environment. One approach to detecting these vulnerabilities is fuzz testing, which feeds a range of randomly modified inputs to a software application while monitoring it for failures. However, typical fuzz testing makes no guarantees regarding the syntactic and semantic validity of the input, or of how much of the input space will be explored. To address these problems, in this paper we present a new testing methodology called *configuration fuzzing*. Configuration fuzzing is a technique whereby the configuration of the running application is randomly modified at certain execution points, in order to check for vulnerabilities that only arise in certain conditions. As the application runs in the deployment environment, this testing technique continuously fuzzes the configuration and checks "security invariants" that, if violated, indicate a vulnerability; however, the fuzzing is performed in a duplicated copy of the original process, so that it does not affect the state of the running application. In addition to discussing the approach and describing a prototype framework for implementation, we also present the results of a case study to demonstrate the approach's efficiency.**

***Keywords*-Vulnerability; Configuration fuzzing; Fuzz testing; In Vivo testing; Security invariants**

I. Introduction

As the Internet has grown in popularity, security testing is undoubtedly becoming a crucial part of the development process for commercial software, especially for server applications. However, it is impossible in terms of time and cost to test all configurations or to simulate all system environments before releasing the software into the field, not to mention that software distributors may later add more configuration options. Fuzz testing as a form of black-box testing was introduced to address this problem [1]. Empirical studies [2] have proven its effectiveness in revealing vulnerabilities of software systems. Yet, typical fuzz testing has been inefficient in two aspects. First, it is poor at exposing certain errors, as most generated inputs fail to satisfy syntactic or semantic constraints and therefore cannot exercise deeper code. Second, given the immensity of the input space, there are no guarantees as to how much of it will be explored [3].

To address these limitations, this paper presents a new testing methodology called *configuration fuzzing*. Instead of generating random inputs that may be semantically invalid, configuration fuzzing mutates the application configuration in a way that helps valid inputs exercise the deeper components of the program-under-test and check for violations of "security invariants" [4]. These invariants represent rules that, if broken, indicate the existence of a vulnerability. Examples of security invariants may include: avoiding memory leakage that may lead to denial of service; a user should never gain access to files that do not belong to him; critical data should never be transmitted over the Internet; only certain sequences of function calls should be allowed, *etc*.

The configuration fuzzing approach is based on the observation that most vulnerabilities occur under specific conditions [5], *i.e.*, an application running with one configuration may prevent the user from doing something bad, while another might not. To facilitate this method, configuration fuzzing occurs within software as it runs in the deployment environment. This allows it to conduct tests in application states and environments that may not have been conceived in the lab. Therefore, this increases the effectiveness of configuration fuzzing by continuing to check for security invariants in the mutated configurations even after the software is released. However, the fuzzing of the configuration occurs in an isolated "sandbox" that is created as a clone of the original process, so that it does not affect the end user of the program.

In this paper, we motivate and describe the configuration fuzzing approach to checking for software vulnerabilities, and discuss an implementation framework. We also present the results of empirical studies that demonstrate that the performance overhead of configuration fuzzing is low enough so that the approach may be carried out on software applications as they execute in the deployment environment with minimal impact on the user.

II. Background

The foundation of the configuration fuzzing methodology is the fact that many applications, especially network-related applications, come with numerous options in the configuration. Take Apache HTTP server as an example: it has more than 50 options that generate over 2^{50} possible settings. Though 2^{50} is relatively small compared to the input space, it is still impractical for testers to test all

potential combinations manually, while vulnerabilities are often revealed in the corner cases that are overlooked. The configuration fuzzing methodology can automate the process of testing multiple configurations and checking for security invariant violations.

Configuration fuzzing is designed as an extension to the In Vivo Testing approach [6], which was originally introduced to detect behavior bugs that reside in software products. In Vivo Testing was principally inspired by the notion of "perpetual testing" [7], which suggests that latent defects still reside in many (if not all) software products and these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. Therefore, testing of software should continue throughout the entire lifetime of the application.

In Vivo Testing conducts tests and checks properties of the software in a duplicated process of the original; this ensures that, although the tests themselves may alter the state of the application, these changes happen in the duplicated process, so that any changes to the state are not seen by the user. This duplicated process can simply be created using a "fork" system call, though this only creates a copy of the in-process memory. If the test needs to modify any local files, In Vivo Testing uses a "process domain" [8][9] to create a more robust "sandbox" that includes a copy-on-write view of the file system. This layered file system allows different processes to have their own view of file system, sharing any readonly base but writing into their own private copies of files and directories.

In previous research into In Vivo Testing, the approach of continuing to test these applications even after deployment was proven to be both effective and efficient in finding remaining misbehavior flaws related to functional correctness [6][10], but not necessarily security defects. In this work, we modify the In Vivo Testing approach to specifically look for security vulnerabilities. Extending the In Vivo Testing approach to configuration fuzzing is motivated by two reasons.

First, many security-related bugs only reveal themselves under certain conditions, which is the configuration of the software together with its running environment. For instance, the FTP server wu-ftpd 2.4.2 assigns a particular user ID to the FTP client in certain configurations such that authentication can succeed even though no password entry is available for a user, thus allowing remote attackers to gain privileges[1]. As another example, certain versions of the FTP server vsftpd, when under heavy load, may allow attackers to cause a denial of service (crash) via a SIGCHLD signal during a malloc or free call[2], depending on the software's configuration. Because In Vivo tests execute within the current environment of the program, rather than by creating a clean slate, it follows that configuration fuzzing increases the possibility of detecting such vulnerabilities that only appear under certain conditions.

Second, the "perpetual testing" foundation of In Vivo Testing ensures that testing can be carried out after the software is released. Continued testing improves the amount of the configuration space that can be explored through fuzzing; therefore it is more likely that an instance will find vulnerabilities under their error-prone configurations.

To address the problem of exploring a potentially large configuration space, configuration fuzzing tests can be assigned to multiple machines using the distributed In Vivo Testing approach [10], in which the testing assignments are split amongst applications running in a homogenous "application community" [11]. If there are many users in the application community, it follows that many more tests will be run, thus increasing the number of possible configurations that are explored as a result of fuzzing, and ideally increasing the likelihood of revealing a vulnerability.

III. Approach

In this section, we describe the steps that software testers would take when using the configuration fuzzing approach. We currently assume access to the source code, though such assumptions could be lifted with the use of a system for binary instrumentation such as Kheiron [12]. The general workflow of the methodology is as follows:

A. Identifying the configuration/setting variables

Most software applications use external configuration, such as .config or .ini files, and/or internal configuration, namely global variables. Given an application to be tested, the tester first locates these configuration parameters that can be mutated. We assume that the tester can annotate the configuration files in such a way that each field is followed by the corresponding variable from the source code and the range of possible values of that variable. A sample annotated configuration file is shown in Listing 1, with the corresponding variables and their values in braces. Every example listed is taken from our empirical study in Section IV using psftp[3], an sftp client.

```
# Passive Telnet
Passive yes #[cfg.passive_telnet]@{0,1}
# X11 forward
X11 no #[cfg.x11_forward]@{0,1}
# Agent forward
Agentforward yes #[cfg.agentfwd]@{0,1}
# Don't allow authenticated users.
NoUserAuth no #[cfg.ssh_no_usrauth]@{0,1}
```

Listing 1. Part of the annotated configuration file for psftp

[1]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1668

[2]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-2259

[3]http://www.chiark.greenend.org.uk/ sgtatham/putty/

Our method mainly fuzzes those configuration variables that are in charge of changing modes or enabling options. These variables often have a binary value of 1/0 or y/n, or sometimes a sequence of numbers representing different modes. Not all configuration variables are modifiable in the sense of revealing vulnerabilities, *e.g.*, fuzzing the host IP address of an ftp server will only lead to unable-to-connect errors. Also, configuration variables that rely on external limitations, such as hardware compatibility, should not be fuzzed. For instance, changing the variable representing the number of CPUs to four when the actual host only has two might cause vulnerabilities instead of detecting them. On the other hand, a considerable number of vulnerabilities are triggered under certain mode/option combinations of network-related applications. For example, WinFTP FTP Server 2.3.0, in passive mode, allows remote authenticated users to cause a denial of service via a sequence of FTP sessions[4]. Also, some early versions of Apache Tomcat allow remote authenticated users to read arbitrary files via a WebDAV write request under certain configurations[5]. By only fuzzing the configuration variables representing modes and options, the size of the configuration space that our approach is fuzzing decreases considerably; however, even with such a decrease, the configuration space may still be too large to test prior to deployment, and thus an In Vivo Testing approach is still useful.

```
void fuzz_config()
{
  int r=random();   /*random number
                      generator*/
  if(r==0) {
    cfg.x11_forward=0;/* Assign values to
    cfg.agentfwd=0;      configuration
    ...                     variables*/
  } else if(r==1){
    cfg.x11_forward=0;/* Assign values to
    cfg.agentfwd=1;      configuration
    ...                     variables*/
  }
  ...
}
```

Listing 2. An example fuzzer for psftp

B. Generating fuzzing code

Given the variables to fuzz and their corresponding possible values (as specified in the configuration file), a pre-processor produces a function that is used to fuzz the configuration, as shown in in Listing 2. The function random() generates a value randomly from zero to the number of possible configurations, assigning different sets of values to the chosen configuration variables. An advanced random() function may use a covering array algorithm [13] to ensure a certain degree of coverage when exhaustive exploration of the configuration space is impossible in the lifetime of the software.

C. Identifying functions to test

The tester then chooses the functions that are to be the instrumentation points for configuration fuzzing. These can conceivably be all of the functions in the program, but would generally be the points at which vulnerabilities would most likely be revealed, or the functions that are related to the configuration variables being fuzzed. The functions are annotated with a special tag in the source code.

D. Generating test code

Given an original function named foo(), a pre-processor first renames it to _foo(), then generates a skeleton for a test function named test_foo(), which is an instance of a configuration fuzzing test. In the test function, the configuration fuzzer (as described above) is first called, and then the original function _foo() is invoked.

Then, the program's security invariants are checked. Based on the properties of the program being tested, different security invariants are predefined by the tester in order to check for violations. The tester writes a *surveillance function* called check_invariants() according to these security invariants. For example, the function could use the substring function strstr(current_directory, legal_directory) to check that the user's current directory has a specified legal directory as its root; if this function indicates otherwise, it may imply that the user has performed an illegal directory traversal. As another example, the check_invariants() function may simply wait to see if the original function _foo() returns at all; if it does not, the process may have been killed or be hanging as a result of a potential vulnerability. These surveillance functions run throughout the testing process, and log every security invariant violation with the fault-revealing configuration into a log file that can be sent to a server for later analysis. Listing 3 shows the test function for function psftp_connect().

```
int test_psftp_connect(...)
{
   fuzz_config(); /*Fuzz configuration*/
   _psftp_connect(...);        /*Call the
                        original function*/
   check_invariants(); /*Check security
                          invariants*/
}
```

Listing 3. Test function for psftp_connect()

[4]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-5666
[5]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-5461

E. Executing tests

In the last step, a wrapper function with the name foo() is created. As in the In Vivo Testing approach, when the function foo() is called, it first forks to create a new process that is a replica of the original. The child process (or the "test process") calls the test_foo() function, which performs the configuration fuzzing and then exits. Because the configuration fuzzing occurs in a separate process from the original, the user will not see its output. Meanwhile, the original function _foo() is invoked in the original process (as seen by the user) and continues as normal. The wrapper function for function psftp_connect() is shown in Listing 4.

Figure 1 shows the general workflow of configuration fuzzing testing.

```
int psftp_connect(...)
{
  int pid=fork(); /*Create new process*/
  if(pid==0) {    /*Test function*/
    test_psftp_connect(...);
    exit();     /*Test exits when done*/
  }             /*Original function*/
  return _psftp_connect(...);
}
```

Listing 4. Wrapper function for psftp_connect()

IV. Performance Evaluation

In this section, we describe the results of experiments that measure the performance cost incurred by the configuration fuzzing approach.

A. Setup

We evaluated our approach's performance by applying it to the psftp client program, which is a part of Putty 0.60 [14], chosen because it is open-source and has multiple configuration options. All experiments were conducted on an Intel Core2Quad Q6600 server with 2.40GHz and 2GB of RAM running Ubuntu 8.04.3.

The function we chose to instrument is psftp_connect(), which authenticates users' logging in. We picked this function because it has many related configuration variables. In the sense of testing the robustness of the authentication process under different modes, we (in the role of testers) picked five related configuration variables: cfg.passive_telnet, cfg.x11_forward, cfg.agentfwd, cfg.tcp_nodelay and cfg.ssh_no_userauth. All of these variables can only vary from 0 to 1 making the size of the configuration space 2^5, which was easily covered by our tests. Then the framework modified the function for configuration fuzzing.

As for security invariants, we only checked whether the forked process (the test process) runs to completion, in order to detect possible denial of service vulnerabilities. Although this alone is not sufficient to find *all* potential vulnerabilities, of course, it serves the purposes of the performance testing since the overhead created by forking a new process is expected to be significantly higher than that of checking the invariants.

For both the original code (without instrumentation) and the instrumented code, we simulated user inputs (both valid and invalid combinations of username and password) for the psftp_connect() function and recorded the function's execution time. The SFTP service was provided on the test machine, and the psftp_connect() function sent requests to IP address 127.0.0.1 rather than to other servers to eliminate any overhead from network traffic. We ran tests in which the function was called 10, 100, 1000, 10000 and 100000 times in order to estimate the overhead caused by our approach.

B. Evaluation

# Tests	Total Time (Original)	Total Time (Instrumented)	Overhead %	Avg Additional Time
10	6.6411	6.6635	0.337	0.002
100	66.592	66.809	0.326	0.003
1000	663.14	666.07	0.442	0.003
10000	6635.6	6659.4	0.359	0.002
100000	66384	66601	0.327	0.002

Table I
TIME COST OF PSFTP_CONNECT() (IN SECONDS) WITH VARYING NUMBER OF TESTS

Table I shows the results we collected from the experiments. The first column shows the number of tests that had been carried out, *i.e.*, the number of times the psftp_connect() function was called. The second and third columns are the total time in seconds for the original function and the instrumented function, respectively. The overhead is calculated in the fourth column and the average additional time (in seconds) per instrumented test is listed in the last column.

From the results we can see that the overhead introduced by our approach is rather small and is unlikely to be noticed by users. In addition, the average additional cost per test stayed around 3ms and did not increase when the number of tests grew. It is worth mentioning that most of the performance overhead comes from the cost of forking a new process, as the test processes are assigned to another core by the In Vivo Testing framework, and do not interfere with the original process. Thus, fuzzing more configuration variables or checking more security invariants would be unlikely to have much effect on the overhead, particularly when running on a multicore machine where the test processes can be assigned to another core.

V. Related Work

One approach to detecting security vulnerabilities is environment permutation with fault injection [15], which perturbs the application environment during the test and checks for symptoms of security violations. Most implementations

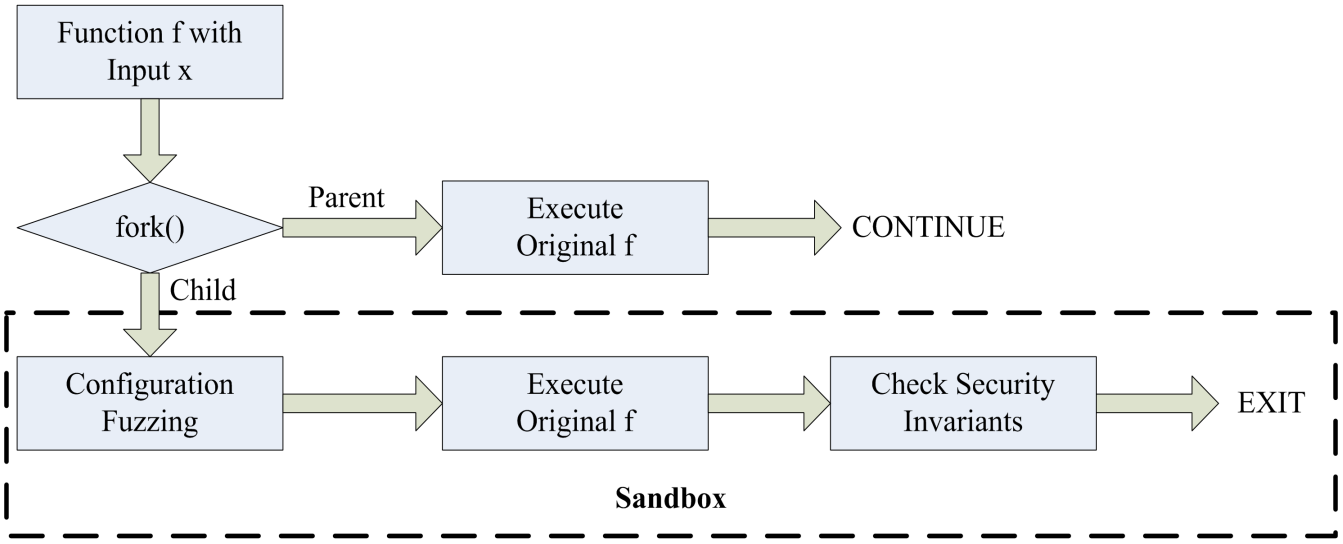

Figure 1. Workflow of Configuration Fuzzing testing

of this approach, such as [16] and [17], view the security testing problem as the problem of testing for the fault-tolerance properties of a software system. They consider each environment perturbation as a fault and the resulting security compromise a failure in the toleration of such faults. However, this hampers the effectiveness of this approach, as the number of defects it may detect is highly dependent on the number of flaws being injected and where they are injected.

Our approach uses the original configuration space of the software-under-test and expects to decrease the occurrence of false positives. Moreover, without injecting external faults but checking for violations of security invariants, we eliminate the dependency on external resources. The two approaches, however, could certainly be used in conjunction with each other; we leave this as future work.

Another popular approach is fuzz testing [1]. Typical fuzz testing is scalable, automatable and does not require access to the source code. It simply feeds malformed inputs to a software application and monitors its failures. The notion behind this technique is that the randomly generated inputs often exercise overlooked corner cases in the parsing component and error checking code. This technique has been shown to be effective in uncovering errors [2], and is used heavily by security researchers [3]. Yet it also suffers from several problems: a single unsigned int value can vary from 0 to 65535, indicating the immensity of the input space, which can hardly be covered with limited time and cost. Furthermore, by only changing the input, a fuzzer may not put the application into a state in which the vulnerability will appear. White-box fuzzing [18] is introduced to help generate well formed inputs instead of random ones and therefore increases their probability of exercising code deep within the semantic core of the computation. It analyzes the source code for semantic constraints and then produces inputs based on them or modifies valid inputs. White-box fuzzing improves the efficiency of fuzz testing; however, it overlooks the enormous size of the input space and also suffers from severe overhead [19].

Our approach deals with this problem by mutating the configuration rather than randomly generating inputs of the program-under-test. The space of the former is considerably smaller than the latter and is more relevant in triggering potential illegal states. In addition, extending the testing phase into deployed environments ensures representative real-world user inputs to test with.

VI. Future Work

Limitations reside, respectively, in configuration fuzzing and In Vivo Testing. We intend to address many of these in future work.

In the current implementation of configuration fuzzing, testers' intervention is required to locate appropriate configuration variables in the current implementation. An automated system could be built to achieve this by parsing source code or external configuration files with annotations. Moreover, since there could be constraints on configuration and the present fuzzer is designed to randomly pick a configuration, the system may end up with states which will never occur in real-world use cases. White-box fuzzing might provide a solution to this problem.

For In Vivo Testing, the most critical limitation of the current implementation is that anything external to the application process itself, *e.g.* database tables, file I/Os, *etc.*, is not replicated by forking the process and the test run in the forked process is less likely to detect vulnerabilities related to these external resources. As mention previously, we are currently looking into integrating with ZAP [8] and

DejaView [9], which can provide a sandbox that addresses local file system issues by creating a copy-on-write view of the file system for each process that is running configuration fuzzing test.

Future work may also include improving the efficiency of our implementation. Our system currently randomly fuzzes the value of all chosen configuration variables. However, there could be a way to only fuzz the values that have not previously been tested by planning out and tracking the different configurations, as in [20], either for a single installation or across multiple application instances (*i.e.*, an application community).

VII. Conclusion

We have presented a new testing methodology called *configuration fuzzing*, which mutates the configuration of a program and checks for violations of security invariants to detect vulnerabilities. By integrating with the In Vivo Testing approach, configuration fuzzing tests continue to run after software is released without affecting the users' experience. We have also provided a prototype implementation of our approach and a case study for performance analysis.

Acknowledgments

The authors are members of the Programming Systems Lab, funded in part by NSF CNS-0905246, CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 1st ed. Addison-Wesley Professional, 2007.

[2] L. Jurani, "Using fuzzing to detect security vulnerabilities," INFIGO, Tech. Rep. INFIGO-TD-01-04-2006, 2006.

[3] T. Clarke, "Fuzzing for software vulnerability discovery," Department of Mathematic, Royal Holloway, University of London, Tech. Rep. RHUL-MA-2009-4, 2009.

[4] J. Biskup, *Security in computing systems challenges, approaches, and solutions*. Springer-Verlag Berlin Heidelberg, 2009.

[5] C. Ramakrishnan and R. Sekar, "Model-based analysis of configuration vulnerabilities," *Journal of Computer Security*, vol. 10, pp. 189–209, 2002.

[6] C. Murphy, G. Kaiser, I. Vo, and M. Chu, "Quality assurance of software applications using the in vivo testing approach," in *Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009, pp. 111–120.

[7] D. Rubenstein, L. Osterweil, and S. Zilberstein, "An anytime approach to analyzing software systems," in *Proc. of 10th FLAIRS*, 1997, pp. 386–391.

[8] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments," in *Proc of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002, pp. 361–376.

[9] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh, "Dejaview: a personal virtual computer recorder," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 279–292.

[10] M. Chu, C. Murphy, and G. Kaiser, "Distributed in vivo testing of software applications," in *Proc. of the First International Conference on Software Testing, Verification and Validation*, April 2008, pp. 509–512.

[11] M. E. Locasto, S. Sidiroglou, and A.D. Keromytis, "Software self-healing using collaborative application communities," in *Proc. of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, February 2006, pp. 95–106.

[12] R. Griffith and G. Kaiser, "A runtime adaptation framework for native C and bytecode applications," in *3rd IEEE International Conference on Autonomic Computing*, June 2006, pp. 93–103.

[13] A. Hartman, *Graph Theory, Combinatorics and Algorithms*. Springer US, 2005, vol. 34, pp. 237–266.

[14] "Putty," `http://www.chiark.greenend.org.uk/sgtatham/putty`.

[15] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[16] W. Du and A. P. Mathur, "Testing for software vulnerability using environment perturbation," in *Proc of International Conference on Dependable Systems and Networks*, 2000, p. 603.

[17] H. H. Thompson, J. A. Whittaker, and F. E. Mottay, "Software security vulnerability testing in hostile environments," in *Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2002, pp. 260–264.

[18] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 474–484.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[20] A. Memon and A. Porter et al., "Skoll: distributed continuous quality assurance," in *Proc. of the 26th International Conference on Software Engineering (ICSE)*, May 2004, pp. 459–468.