CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection

Huning Dai, Christian Murphy, Gail Kaiser
Department of Computer Science
Columbia University
New York, NY 10027 USA

ABSTRACT

Many software security vulnerabilities only reveal themselves under certain conditions, i.e., particular configurations and inputs together with a certain runtime environment. One approach to detecting these vulnerabilities is fuzz testing. 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, we present a new testing methodology called *Configuration Fuzzing*. Configuration Fuzzing is a technique whereby the configuration of the running application is mutated 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. We discuss the approach and introduce a prototype framework called *ConFu* (CONfiguration FUzzing testing framework) for implementation. We also present the results of case studies that demonstrate the approach's feasibility and evaluate its performance.

Keywords: Vulnerability; Configuration Fuzzing; Fuzz testing; In Vivo testing; Security invariants

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 the fact that software distributors may later add more configuration options. The configuration of a software system is a set of options that are responsible for a user's preferences and the choice of hardware, functionality, etc. Sophisticated software systems always have a large number of possible configurations, e.g., a recent version of Firefox has more than 2^{30} possible configurations, and testing all of them is infeasible before the release. Fuzz testing as a form of black-box testing was introduced to address this problem (Sutton et al., 2007), and empirical studies (Jurani, 2006) have proven its effectiveness in revealing vulnerabilities in 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 (Clarke, 2009).

To address these limitations, this paper presents a new testing methodology called *Configuration Fuzzing*, and a prototype framework called *ConFu* (CONfiguration FUzzing framework). Instead of generating random inputs that may be semantically invalid, ConFu mutates the application configuration in a way that helps valid inputs exercise the deeper components of the software-under-test and check for violations of program-specific "security invariants" (Biskup, 2009). 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. ConFu mutates the configuration using the incremental covering array approach (Fouche et al., 2009), therefore guaranteeing considerable coverage of the configuration space in the lifetime of a certain release of the software.

Configuration Fuzzing works as follows: Given an application to test, the testers annotate the variables to be fuzzed in the configuration file and choose the functions to test. If needed, they can write additional surveillance functions for specific security invariants other than the built-in ones provided by our default implementation. The framework then generates the actual code for a fuzzer that mutates the values of the chosen configuration variables, as well as the test functions for each chosen function. Next, the framework creates instrumentation such that whenever a chosen function is called, the corresponding test function is executed in a sandbox with the mutated configuration and the security invariants are checked. Violations of these security invariants are logged and sent back to the developer.

Configuration Fuzzing is based on the observation that most vulnerabilities occur under specific configurations with certain inputs (Ramakrishnan and Sekar, 2002), i.e., an application running with one configuration may prevent the user from doing something bad, while another might not. 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. In addition, the effectiveness of ConFu is increased by using real-world user inputs rather than randomly generated ones. 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. When a vulnerability is detected, detailed information is collected and sent back to a server for later analysis.

The rest of this paper is organized as follows. Section 2 formalizes the problem statement, and identifies requirements that a solution must meet. Section 3 discusses the background, propose the Configuration Fuzzing approach, and provide the architecture of the framework called ConFu. Section 4 looks at the results of our case studies and performance evaluation. Related work is then discussed in Section 5. The paper ends with limitations in Section 6, and finally the conclusion in Section 7.

PROBLEM AND REQUIREMENTS

Problem statement

We have observed that configuration together with user input are the major factors of vulnerability exploitation. However, it is generally infeasible to test all functionality with all possible configurations in terms of time and cost before releasing the software into the field. For example, the Apache HTTP server has more than 50 options that generate over 2^{50} possible settings, and certain vulnerabilities[i, ii] will only reveal themselves under specific configurations with specific user inputs. The effectiveness and efficiency of detecting such vulnerabilities in the testing process are greatly hampered due to the immensity of both the configuration and input space. Another issue of security testing resides in the difficulty of detecting vulnerabilities when the characteristics of the vulnerabilities are not deterministic and vary greatly among different vulnerabilities. A "test oracle" (Weyuker, 1982) alone is only sufficient in evaluating the correctness of a software but not its security. Furthermore, most configuration testing approaches such as Skoll (Memon and Porter et al., 2004) or MSET (Gross et al., 2006) provide little feedback beyond "pass/fail" and thus not addressing security issues.

Requirements

A solution to this problem would need to address not only the issue of the immensity of the configuration and the input space, but also consider the effectiveness and efficiency of characterizing vulnerabilities and providing detailed information regarding detected vulnerabilities. Such a solution should meet the following requirements:

Guarantee a considerable degree of coverage of the configuration space. In a limited amount of time, the solution has to guarantee sufficient coverage of the configuration space. Because some options have to match the external environment, it might be impossible to provide full-coverage testing. However, the solution must at least have covered the most common configurations.

Support representative user inputs with which to test. The solution has to optimize the number of possible user inputs for a given configuration. The validity of the user inputs is crucial and these inputs should satisfy syntactic or semantic constraints and therefore exercise the deeper components of the software. “Maximum” coverage of the input space is desired, but with “minimum” actual test cases.

Be able to detect the most common vulnerabilities and provide an easy-to-use API to add rules for detecting other vulnerabilities. The solution should provide an effective mechanism to detect common kinds of security issues such as directory traversal, denial of service, insufficient access control, etc. Also, it must be easy to add an additional rule if a new vulnerability or exploit arises, without having to change the existing framework.

Be capable of reporting vulnerabilities back to the software developers. If a test fails and a vulnerability is discovered, the framework must allow for feedback to be sent to the software developers so that the failure can be analyzed and, ideally, fixed. In addition to sending a notification of a discovered vulnerability, the framework should also send back useful information about the system state so that the vulnerability can be reproduced.

Have low performance impact. The user of a system that is conducting tests on itself during execution should not observe any noticeable performance degradation. The tests must be unobtrusive to the end user, both in terms of functionality and any configuration or setup, in addition to performance.

APPROACH

Background

Configuration Fuzzing is designed as an extension to the In Vivo Testing approach (Murphy et al., 2009), which was originally introduced to detect behavioral bugs that reside in software products. In Vivo Testing was principally inspired by the notion of "perpetual testing"(Osterweil, 1996; Rubenstein et al., 1997), 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 approaches this problem by executing tests at specified points in the context of the running program after the software is released.

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 tests can use a "process domain" (Osman et al., 2002) 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 the file system, sharing any read only files 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 (Chu et al., 2008; Murphy et al., 2009), 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 three reasons.

First, many security-related bugs only reveal themselves under certain conditions, which is the configuration of the software together with its runtime 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[iii]. 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[iv], 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.

Third, In Vivo Testing uses real-world user inputs, which may be more likely to trigger vulnerabilities. Due to the impossibility of full coverage of the input space (Zhu et al., 1997), using real-world user inputs has a higher probability of detecting vulnerabilities over contrived lab inputs.

Model

When an instrumented function is called, Configuration Fuzzing **mutates the application configuration under predefined configuration constraints of the software-under-test to look for potential vulnerabilities**. By extending the In Vivo Testing approach, **Configuration Fuzzing tests are executed in the field**, after deployment, and will provide representative real-world user inputs to test with and reveal vulnerabilities that are dependent on the application state. Furthermore, **surveillance functions using security invariants are executed throughout the test** in order to detect violations of security rules, which indicate the occurrence of a vulnerability if broken.

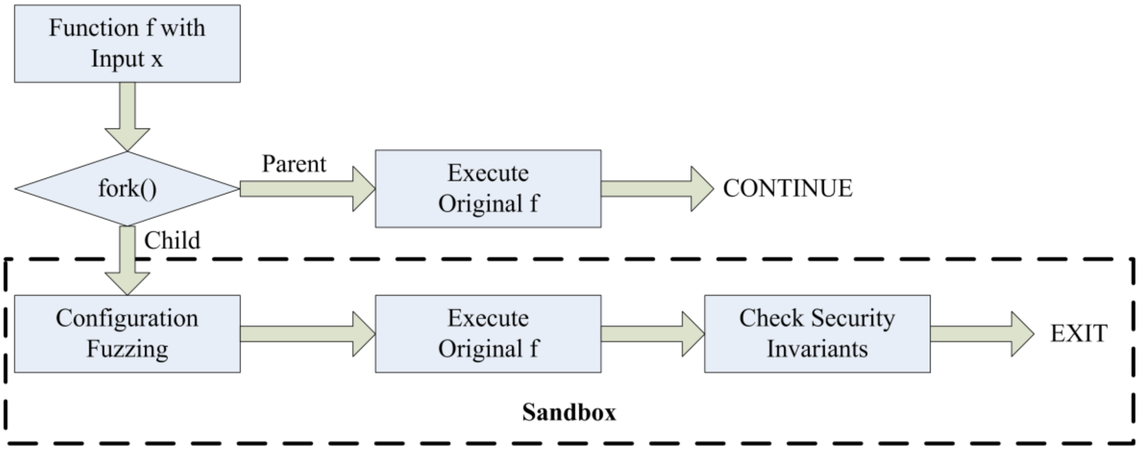

Figure 1. The model of Configuration Fuzzing Testing

The model of Configuration Fuzzing Testing is shown in Figure 1. Given a function named *f* with input *x* to test, we create a sandbox (illustrated by fork() in Figure 1) before *f* actually gets called. The sandbox is a replica of the original process with the same system state. The original function *f* will be executed in the original process as normal while Configuration Fuzzing tests take place in the sandbox. Configuration Fuzzing tests are composed of three parts. First, configuration variables are mutated. However, since Configuration Fuzzing tests take place in a replica of the original process, the configuration variables stay unchanged in the original process. Second, the original function *f* is executed with the mutated configuration trying to exploit potential vulnerabilities. Last, a surveillance function that checks for violation of security invariants is called so as to detect any vulnerability exploitation and send reports to the system administrator or developers if found. After this, the test process is terminated while the original process continues.

Architecture

Here we introduce the architecture of a framework called ConFu (Configuration Fuzzing framework for vulnerability detection). ConFu mutates the configuration of an application with a covering array algorithm (Hartman, 2005) and checks for vulnerabilities using surveillance functions that monitor violations of security invariants. This framework allows the application to be tested as it runs in the field, using real input data. As described above, multiple invocations of the instrumented functions are run; however, the additional invocations must not affect the user and must run in a separate sandbox. The steps that software testers would take when using ConFu are as follows:

Step 1: 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 Figure 2, with the corresponding variables and their values in braces. The examples listed are taken from our empirical study in Section 4 using OpenSSH[v], a secure shell server.

```
X11Forwarding yes     #[options.x11_forwarding]@{0,1}
TCPKeepAlive yes      #[options.tcp_keep_alive]@{0,1}
UseLogin        no    #[options.use_login]@{0,1}
Protocol        1     #[options.protocol]@{1,2,3}
```

Figure 2. Part of the annotated configuration file for OpenSSH

Our method mainly fuzzes those configuration variables that are responsible for 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 in 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[vi]. Also, some early versions of Apache Tomcat allow remote authenticated users to read arbitrary files via a WebDAV write request under certain configurations[vii]. 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 such as Configuration Fuzzing is still useful.

Step 2: 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 Figure 3. The fuzz_config() function uses a covering array algorithm (Hartman, 2005) to ensure a certain degree of coverage when exhaustive exploration of the configuration space is impossible in the lifetime of the software.

```
typedef struct {
     int x11_forward;
     int tcp_keep_alive;
    …
} result;

void fuzz_config()
{
  /* generate a set of options */
      result  r=covering_array();
      options.x11_forward = r.x11_forward;
      options.tcp_keep_alive = r.tcp_keep_alive;
      ...
}
```

Figure 3. An example fuzzer for OpenSSH

Consider *k* as the number of variables a configuration needs to specify and *v* as the number of possible values each of the *k* variables can be. We define the level of coverage in terms of a parameter *t*, which if equal to *k* will produce full coverage, and produce no coverage when equal to zero. The set of configurations generated is called a *t*-way covering array. Take a simple program that uses three (*k*=3) binary (*v*=2) variables as an example. A 3-way covering array will include all 2^3 configurations, therefore guaranteeing full coverage. A 2-way covering array will look like Figure 4; we notice that whichever two columns out of the three columns are chosen, all possible pairs of values appear. Specifically, the pairs 00, 01, 10 and 11 all appear in the rows when we look at the columns of AB only, AC only and BC only. This property is called "2-coverage", and corresponds to t=2. A notion called CAN(t, k ,v) represents the number of configurations in the smallest (optimal) set that holds the "t-coverage" property for a configuration space of size k^v. When using the covering array algorithm, our approach will start mutating the configuration variables with a CAN(2, k, v) covering array and increase t afterwards if time permits. Ideally it would be possible to take t=k, but that might lead to too many configurations to test. Empirical studies (Hayhurst et al., 2001) show that for most software t need not be more than 6 to find all errors. The covering_array() function is implemented using Jenny[viii], an open sourced covering array generator.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4. A 2-way covering array

Step3: 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. Future work could investigate a general approach to determine which functions to test. The chosen functions are annotated with a special tag in the source code.

Step4: Generating test code. As an example, given an original function named do_child() in the program OpenSSH, a pre-processor first renames it to ConFu_do_child(), then generates a skeleton for a test function named ConFu_test_do_child(), 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 ConFu_do_child() is invoked.

```
void ConFu_test_do_child(…)
{
    fuzz_config(); /*Fuzz configuration*/
    ConFu_do_child(…);  /*Call the
                          original function*/
    check_invariants();
}
```

Figure 5. Test function for do_child()

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 indicate 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 ConFu_do_child() 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 could be sent to a server for later analysis. Figure 5 shows the test function for function do_child(). By default, ConFu has three built-in security invariants that check for denial of service, unauthorized directory traversal and insufficient privilege control.

Step 5: Executing tests. In the last step, a wrapper function with the name do_child() is created. As in the In Vivo Testing approach, when the function do_child() 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 ConFu_test_do_child() 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 ConFu_do_child() is invoked in the original process (as seen by the user) and continues as normal. The wrapper function for function do_child() is shown in Figure 6.

```
void do_child(…)
{
    int pid = fork();  /*Create new process*/
    if(pid == 0) {     /*Test function*/
        ConFu_test_do_child(…);
        exit(0);
    }
    return ConFu_do_child(); /*Originalfunction*/
}
```

Figure 6. Wrapper function for do_child()

EVALUATION

Setup

In order to demonstrate the feasibility of using Configuration Fuzzing to detect vulnerabilities, we reproduced certain known vulnerabilities and used ConFu to find them. Due to space limitations, we present only one example vulnerability. The vulnerability we chose is that early versions of OpenSSH do not properly drop privileges when the UseLogin option is enabled, which allows local users to execute arbitrary commands by providing the command to the ssh daemon[ix]. The *CVSS Severity*[x] of this vulnerability was 10 (the highest) and we believe it was mainly caused by insufficient testing of the configurations of OpenSSH. We chose this vulnerability not only because of its high severity but also because insufficient privilege control is one of the most common vulnerabilities besides denial of service and unauthorized directory traversal. The following lists the details of each step in using ConFu to detect the vulnerability.

- Identifying the configuration variables

The sshd server OpenSSH 2.1.0 was used as the program-under-test in our feasibility study. From the configuration file (sshd_config) we found that there are a total of 15 modifiable (fuzzable) configuration variables: permit_root_login, ignore_ rhosts, ignore_user_known_hosts, strict_modes, x11_forwarding, print_motd, keepalives, rhosts_authentication, password_authentication, permit_empty_passwd, kerberos_authen tication, kerberos_or_local_passwd, kerberos_ticket_cleanup, use_login, and check_mail. We annotated these variables with the possible values in the configuration file.

- Generating fuzzing code

After recognizing these configuration variables, ConFu generated the fuzz_config() function which mutates the configuration.

- Identifying functions to test

We picked the do_child() function as the function to test. The reason we picked this function in the role of software testers is that do_child() is responsible for creating a session and authenticating the user's identity when a ssh client tries to access the sshd server, and it is one of the functions most vulnerable to insufficient privilege control. Note that a wider range of functions can also be tested, including the main() function.

- Generating test code

The original function do_child() was renamed by a preprocessor to ConFu_do_child() and the test function ConFu_test_do_child() was generated with the fuzz_config() function and the check_invariants() function. Several security invariants were checked in our experiment, including the security invariant for privilege control, which is that a user should never be able to use other users' identities. An initial surveillance function that checks this security invariant is shown in Figure 7, which checks whether the user identification has changed.

```
void check_invariants(Session *s)
{
    if(geteuid() != s.uid || getuid() != s.uid)
        /* Log the detection */
        log(“Insufficient privilege control”);
     /* Check for other security invariants */
        …
}
```

Figure 7. Surveillance function for ConFu_test_do_child()

- Executing tests

Last, ConFu generated a wrapper function with the name do_child() which forks a child process to execute the ConFu_test_do_child() function and runs the original function ConFu_do_child() in the parent process, as shown in Figure 6.

Results

To facilitate the exploitation, we simulated both valid and invalid combinations of username and password as user inputs in the real-world for the instrumented sshd server. If the vulnerability is exploited, the server program records the exploitation with its corresponding configuration in a log file. We ran the program 10,000 times which took roughly ten minutes. A fragment of the log file is shown in Figure 8. By analyzing the log file, we were able to find the mapping between the UseLogin option and insufficient privilege control. It is also worth pointing out that the number of detections in the log file was identical to the number of tests where UseLogin was enabled (with valid input). We have not yet managed to detect new vulnerabilities, however, this study demonstrates the functional feasibility of the Configuration Fuzzing approach. Performance feasibility is discussed below.

```
permit_root_login option is:0
…
…
UseLogin option is:1
Check_mail option is: 0
Insufficient privilege control

permit_root_login option is:1
…
…
UseLogin option is:1
Check_mail option is: 0
Insufficient privilege control
```

Figure 8. Log file of the OpenSSH server

Performance

The performance impact of ConFu is crucial because Configuration Fuzzing tests are executed while the program-under-test is running. We evaluated our approach's performance by applying it to the OpenSSH server with the steps stated above. do_child() is chosen as the function to test and 15 configuration variables are fuzzed. All experiments were conducted on an Intel Core2Quad Q6600 server with a 2.40GHz CPU and 2GB of RAM running Ubuntu 8.04.3.

# of tests	Overhead introduced by fuzz_config()	Overhead introduced by ConFu_do_child()	Overhead introduced by check_invariants()	Total Avg. additional time per test
100	3.446	0.271	0.001	0.037
1000	42.23	2.434	0.014	0.045
10000	378.2	29.92	0.157	0.041
100000	3694	236.8	1.628	0.039

Figure 9. Overhead of instrumented do_child()(in seconds) with varying number of tests

For both the original code (without instrumentation) and the instrumented code, we simulated user inputs (both valid and invalid) for the do_child() function and recorded the function's execution time. The OpenSSH service was provided on the test machine, and the do_child() 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 100, 1000, 10000 and 100000 times in order to estimate the overhead caused by our approach.

Figure 9 shows the results we collected from the experiments. The first column shows the number of tests that were carried out, i.e., the number of times the do_child() function was called. The second and third columns are the overhead in seconds for the fuzz_config() function and the ConFu_do_child() function, respectively. The total average additional time (in seconds) per instrumented test is listed in the last column. From the results we can see that the average additional cost per test stayed around 45ms and did not increase when the number of tests grew. Thus, a single client of a server running OpenSSH with ConFu is unlikely to notice any performance slowdown. It is worth mentioning that most of the performance overhead comes from the cost of generating the covering arrays in fuzz_config(), and this cost is only affected by the number of variables being fuzzed.

To obtain better performance, software testers could, in principle, pre-calculate the covering arrays for the chosen configuration variables before the software is shipped. However, since many configuration options are enabled or disabled at build time and later hotfixes might add more configuration options, it is more secure to calculate the covering array during the runtime of the software.

RELATED WORK

Security testing

One approach to detecting security vulnerabilities is environment permutation with fault injection (Hsueh et al., 1997), which perturbs the application external environment during the test and checks for symptoms of security violations. Most implementations of this approach view the security testing problem as the problem of testing the fault-tolerance properties of a software system (Du & Mathur, 2000; Thompson et al., 2002). They consider each environment perturbation as a fault and the resulting security compromise a failure in the toleration of such faults. However, as the errors being injected are independent of the software, most of these errors might not occur in real-world usage. Therefore, fault injection testing may raise false positives.

Instead of injecting faults, ConFu mutates the configuration under predefined configuration constraints of the software-under-test to produce potential vulnerabilities, which relies on the internal properties of the software. Hence it would decrease the occurrence of false positives considerably. The two approaches, however, could certainly be used in conjunction with each other; we leave this as future work.

Another security testing approach used to detect vulnerabilities is anomaly detection (Hangal & Lam, 2002; Krügel et al., 2002). Anomaly detection first establishes a model of normal behavior then detects data sets that cause the program to not conform to the model. Anomaly detection is potentially capable of detecting zero day attacks; however, it is always difficult to define normal behavior. Thus, these approaches depend on the validity of the normal behavior model being used. Anomaly detection may suffer severely from false positives.

ConFu treats the violations of security invariants as vulnerability exploits, and these security invariants are defined based on the consequences of known abnormal behaviors. In a sense, ConFu uses a model of abnormal behaviors that is much easier to obtain than a model of normal behaviors. Hence ConFu would be expected to have fewer false positives and false negatives; however, further empirical studies are needed to compare the effectiveness and efficiency between these two approaches.

Another popular approach is fuzz testing (Sutton et al., 2007). 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 (Jurani, 2006), and is used heavily by security researchers (Clarke, 2009). Yet it also suffers from several problems: a single unsigned int value can vary from 0 to 65535; adding another int value to the input domain causes the input space to grow exponentially, 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 (Ganesh et al., 2009) was 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 effectiveness of fuzz testing; however, it overlooks the enormous size of the input space and also suffers from severe overhead (Godefroid et al., 2008).

ConFu 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 with which to test.

Configuration testing

Configuration testing plays an irreplaceable role in security testing. The importance of configuration testing has increased as more and more vulnerabilities are discovered to be caused by inappropriate configuration. However, most of these popular configuration testing approaches were not designed to reveal security defects. One approach named Rachet (Yoon et al., 2008), is designed to test the compatibility of a software with mutated configuration in the development process. Rachet models the entire configuration space for software systems and uses the model to generate test plans to sample a portion of the space, and later uses these test plans to test the compatibility during the compile time of the software.

Another approach called Skoll (Memon and Porter et al., 2004) is composed of software quality assurance (QA) processes that leverage the extensive computing resources of volunteer communities to improve software quality. Skoll takes a QA process' configuration space to build a formal model which captures all valid configurations for QA subtasks, and uses this model to generate test cases for each machine in the community. Skoll collects the pass/fail results of all the tests to provide feedback to the developers.

Both of these approaches are able to detect functionality errors when the program-under-test fails. However, vulnerabilities such as security defects are more difficult to find since most of them will not lead the program to failure. ConFu deals with this problem by checking the violations of security invariants with surveillance functions. Whereas vulnerabilities such as directory traversal and denial of service can easily hide themselves from being detected by Rachet or Skoll, ConFu can still catch these vulnerabilities when they alter the values checked by the security invariants.

LIMITATIONS

The most critical limitation of the current implementation is that testers’ intervention is required to identify the functions to test. In principal, one can always choose the main() function, but it might be less efficient and increase the overhead. A better way to determine the function-under-test might be integrating Configuration Fuzzing with a run-time injection approach such as the one proposed by Antoni et al. (2003), or picking the most frequently called functions according to real world usage. We leave this as future work.

ConFu relies on surveillance functions that check for violations of security invariants to detect vulnerabilities. Therefore, software testers need a priori knowledge of the potential exploitation behavior in order to design specific surveillance functions. ConFu’s built-in surveillance functions check for common security invariants and are capable of detecting most well-known exploits, e.g. denial of service, directory traversal, etc. However, they might not be as effective in detecting zero day vulnerabilities.

Because each configuration is only tested with relatively few inputs, the chance of detecting a vulnerability that would only be revealed under one specific configuration with one particular input is relatively low using the current ConFu implementation. A distributed version of ConFu would increase the efficiency of detecting such vulnerabilities, in which the testing assignments are split amongst applications running in a homogenous “application community” (Locasto et al., 2006). We have developed a distributed In Vivo Testing framework(Chu et al., 2008) and we leave it as future work to develop a global coordinator that is in charge of allocating test cases for each deployed configuration across the user community, and collecting and analyzing the results for ConFu.

CONCLUSION

In this paper, we explored an approach for software vulnerability detection in the domain of security testing and developed a framework called ConFu based on our approach. Vulnerability detection is difficult to achieve not only because the characteristics of vulnerabilities are hard to define but also because of the immensity of the input and configuration space. Our proposed approach, Configuration Fuzzing, deals with these problems by extending the testing phase into the deployment environment while ensuring a considerable degree of coverage configuration space and representative samples of the input space. Surveillance functions that check for violations of security invariants are executed during Configuration Fuzzing in order to detect vulnerabilities. Configuration Fuzzing tests happen in a duplicated copy of the original process, so that they do not affect the state of the running application. As future work, we are planning to develop a distributed version of ConFu with a full-fledged sandbox, which will significantly further its potential in detecting software vulnerabilities. We believe that ConFu can help developers build more secure software and improve the security of existing software systems.

ACKNOWLEDGEMENTS

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

Antoni, L., Leveugle, R., and Feher, B. (2003). Using run-time reconfiguration for fault injection applications. *IEEE Transactions on Instrumentation and Measurement*, 52(5).

Biskup, J. (2009). *Security in computing systems challenges, approaches, and solutions*. Springer-Verlag Berlin Heidelberg.

Chu, M., Murphy, C., and Kaiser, G. (2008). Distributed in vivo testing of software applications. In *Proc. of the First International Conference on Software Testing, Verification and Validation*, pages 509–512.

Clarke, T. (2009). Fuzzing for software vulnerability discovery. Technical Report RHUL-MA-2009-4, Department of Mathematics, Royal Holloway, University of London.

Du, W. and Mathur, A. P. (2000). Testing for software vulnerability using environment perturbation. In *Proc. of the International Conference on Dependable Systems and Networks*, page 603.

Fouch é, S., Cohen, M. B., and Porter, A. (2009). Incremental covering array failure characterization in large configuration spaces. In *ISSTA '09: Proc. of the Eighteenth International Symposium on Software Testing and Analysis*, pages 177–188, New York, NY, USA. ACM.

Ganesh, V., Leek, T., and Rinard, M. (2009). Taint-based directed whitebox fuzzing. In *ICSE '09: Proc. of the 2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, Washington, DC, USA. IEEE Computer Society.

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

Gross, K. C., Urmanov, A., Votta, L. G., McMaster, S., and Porter, A. (2006). Towards dependability in everyday software using software telemetry. In *EASE '06: Proc. of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, pages 9–18, Washington, DC, USA. IEEE Computer Society.

Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In *Proc. of the 2002 International Conference on Software Engineering*, pages 291-301.

Hartman, A. (2005). *Graph Theory, Combinatorics and Algorithms*, volume 34, pages 237–266. Springer US.

Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J.,and Rierson, L.K.(2001). A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *Computer*, 30(4):75–82.

Jurani, L. (2006). Using fuzzing to detect security vulnerabilities. Technical Report INFIGO-TD-01-04-2006, INFIGO.

Krügel, C., Toth, T., and Kirda, E. (2002). Service specific anomaly detection for network intrusion detection. In *SAC '02: Proc. of the 2002 ACM Symposium on Applied Computing*, pages 201–208, New York, NY, USA. ACM.

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

Memon, A. and Porter et al., A. (2004). Skoll: distributed continuous quality assurance. In *Proc. of the 26th International Conference on Software Engineering (ICSE)*, pages 459–468.

Murphy, C., Kaiser, G., Vo, I., and Chu, M. (2009). 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)*, pages 111–120.

Osman, S., Subhraveti, D., Su, G., and Nieh, J. (2002). 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)*, pages 361–376.

Osterweil, L. (1996). Perpetually testing software. In *The Ninth International Software Quality Week*.

Ramakrishnan, C. and Sekar, R. (2002). Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10:189–209.

Rubenstein, D., Osterweil, L., and Zilberstein, S. (1997). An anytime approach to analyzing software systems. In *Proc. of 10th FLAIRS*, pages 386–391.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, first edition.

Thompson, H. H., Whittaker, J. A., and Mottay, F. E. (2002). Software security vulnerability testing in hostile environments. In *Proc. of the 2002 ACM Symposium on Applied Computing*, pages 260–264, New York, NY, USA. ACM.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4):465–470.

Yoon, I.-C., Sussman, A., Memon, A., and Porter, A. (2008). Effective and scalable software compatibility testing. In *ISSTA '08: Proc. of the 2008 International Symposium on Software Testing and Analysis*, pages 63–74, New York, NY, USA. ACM.

Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427.

[i] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-5461
[ii] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-1742
[iii] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1668
[iv] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-2259
[v] http://www.openssh.com/
[vi] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-5666
[vii] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-5461
[viii] http://burtleburtle.net/bob/math/jenny.html
[ix] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-CVE-2000-0525
[x] http://www.oracle.com/technology/deploy/security/cpu/cvssscoringsystem.htm