ConFu: Configuration Fuzzing Framework for Software Vulnerability Detection

Thesis proposal

Huning Dai
Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
dai@cs.columbia.edu

Advisor: Gail E. Kaiser

December 8, 2009

Abstract

Many software security vulnerabilities only reveal themselves under certain conditions, *i.e.*, particular configurations of the software and certain inputs 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 proposal 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; 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. Configuration Fuzzing uses a covering array algorithm when fuzzing the configuration which guarantees a certain degree of coverage of the configuration space in the lifetime of the program-under-test. In addition, Configuration Fuzzing tests that are run after the software is released ensure representative real-world user inputs to test with. In addition to discussing the approach and describing a prototype framework for implementation, we also present the results of case studies to prove the approach's feasibility and evaluate its performance.

In this thesis, we will continue developing the framework called ConFu (CONfiguration FUzzi ng framework) that supports the generation of test functions, parallel sandboxed execution and vulnerability detection. Given the initial ConFu, we will optimize the way that configurations get mutated, define more security invariants and conduct additional empirical studies of ConFu's effectiveness in detecting vulnerabilities.

At the conclusion of this work, we want to prove that ConFu is efficient and effective in detecting common vulnerabilities and tests executed by ConFu can ensure reasonable degree of coverage of both the configuration and user input space in the lifetime of the software.

Contents

1 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. The configuration of a software system is a set of options that responsible for a user's preferences and the choice of hardware, functionality, *etc*. Sophisticated software systems always have a large amount of possible configurations, *i.e.*, recent version of Firefox has more than 2^{30} configurations and test all of them is infeasible before the release. Fuzz testing as a form of black-box testing was introduced to address this problem [20]. Empirical studies [13] 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 [5].

To address these limitations, this proposal 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 program-under-test and check for violations of "security invariants" [3]. 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 [7], therefore guaranteeing a certain degree of coverage of the configuration space in the lifetime of a certain release of a software.

Configuration Fuzzing is based on the observation that most vulnerabilities occur under specific configurations with certain inputs [18], *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 proposal is organized as follows. Section 2 formalizes the problem statement, and identifies requirements that a solution must meet. In Section 3, we propose the approach Configuration Fuzzing to the solution, and specify our hypotheses. Sections 4 looks at different parts of the solution, including the model that we will use, a more detailed architecture, and the results of our initial feasibility studies and performance evaluation. Related work is then discussed in Section 5. The proposal ends with a detailed research plan in Section 6, and finally the conclusion in Section 7.

2 Problem, Definitions and Requirements

2.1 Definitions

This section formalizes some terms that are used throughout the proposal:

- *Configuration*: An arrangement of settings that represents a user's preferences and the choice of hardware, functionality, *etc*.
- *Vulnerability*: A term in computer security represents weaknesses which allow a malicious user to break a system's security.
- *Configuration Fuzzing*: A methodology of testing that mutates the configuration of a software to check for vulnerabilities after the software is released.
- *ConFu*: Configuration Fuzzing framework for software vulnerability detection
- *Security Invariant* [3]: Representing rules that, if broken, indicate the existence of vulnerabilities.
- *Surveillance Function*: Functions that checks for violations of security invariants throughout the testing process.
- *In Vivo Testing* [16]: A perpetual testing [19] approach that continues the testing process of a software to the deployment stage after it is released.
- *Covering Array* [11]: A mathematical object that can be used for software testing purposes, which ensures that the tests can cover a degree of percentation of all possible user cases.

2.2 Problem Statement

We have observed that configuration together with user input are the major factors of vulnerability exploitation. However, it is generally unfeasible to test all use cases with all possible configuration in terms of time and cost before releasing the software into the field. For example, Apache HTTP server has more than 50 options that generate over 2^{50} possible settings, and certain vulnerabilities[1],[2] 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" [2] alone is only sufficient in evaluating the correctness of a software but not the security. Furthermore, most testing approaches such as Skoll [15] or MSET [10] provide little information in the feedback with only a "pass/fail". This impedes the progress of reproducing and fixing the vulnerability for software developers.

[1] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-5461
[2] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-1742

2.3 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.

1. **Guarantee considerable degree of coverage of the configuration space.** In a limited amount of time, this solution has to guarantee a certain degree of 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 useful configurations.
2. **Support representative user inputs to test with.** The solution has to maximize the number of possible user inputs for a given configurations. 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. Full coverage of the input space might not be feasible in a limited amount of time, however "maximum" coverage of the input space is desired.
3. **Be able to detect the most popular vulnerabilities and provide an easy-to-use API to add rules for detecting other vulnerabilities.** The solution should provide an effective mechanism to detect popular kinds of vulnerabilities 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 rule arises, without having to change the existing framework.
4. **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.
5. **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.

3 Hypotheses and Proposed Approach

This section describes our proposed approach to solving the above problem. It also states the hypotheses that the thesis will investigate, and clarifies the scope of the work.

3.1 Proposed Approach

3.1.1 Background

Configuration Fuzzing is designed as an extension to the In Vivo Testing approach [16], 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" [19], 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 arbitrary 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, we are looking into using a "process domain" [17] 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 read only 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 [16][4], but not necessarily security defects. In this work, we modify the In Vivo Testing approach to specifically look for security vulnerabilities.

3.1.2 Configuration Fuzzing approach

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 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[3]. 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[4], 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 are more likely to trigger vulnerabilities. Due to the impossibility of full coverage of the input space [23], using real-world user inputs has a higher probability of detecting vulnerabilities over the test cases in the lab. It also ensures that Configuration Fuzzing tests can acknowledge the first occurrence of a vulnerability exploitation.

Our approach first **mutates the configuration under predefined configuration constraints of the**

[3] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1668
[4] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-2259

software-under-test to produce potential vulnerabilities. Next, by combining with 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 of security invariants are executed throughout the test** in order to detect violations of security rules, which indicate the occurrence of a vulnerability if broken.

3.2 Hypotheses

The main hypotheses investigated are as follows:

1. **For programs that have many configuration settings, conducting Configuration Fuzzing testing within the context of the application running in the field is feasible to detect vulnerabilities.** That is, the approach can reveal the most popular vulnerabilities in programs that might not be easily found in the development environment.
2. **In the lifetime of the software, Configuration Fuzzing tests ensure a considerable degree of coverage of both the configuration space and input space.** In the lifetime of a software release, Configuration Fuzzing tests should be executed effectively for "maximum" coverage.
3. **This can be done without affecting the application state from the users perspective, and with acceptable performance overhead.** Users of the software would ideally not even know that any testing was being performed.

We will apply this approach to software systems in the domain of server applications, focusing specifically on vulnerabilities that are a result of untested deployment environments and configurations that may not have been conceived in the lab, as opposed to computational defects that come about due to programming errors and misinterpretation of specifications. As server applications become more and more prevalent in various aspects of everyday life, it is clear that their quality and reliability take on increasing importance.

4 Configuration Fuzzing

4.1 Model

The model of Configuration Fuzzing testing is shown in Figure 1. Given a function named f with input x to test, we create a child process using fork() before f actually gets called. The child process represents a sandbox which is a replica of the original process with the same system state. The original function f will be executed in the parent process as normal when Configuration Fuzzing tests take place in the child process. Configuration Fuzzing tests are composed of several parts. First, configuration variables are fuzzed. Second, the original function f is executed with the mutated configuration. At last, a surveillance function that checks for violation of security invariants is called so as to detect any vulnerability exploitation and send reports if found.

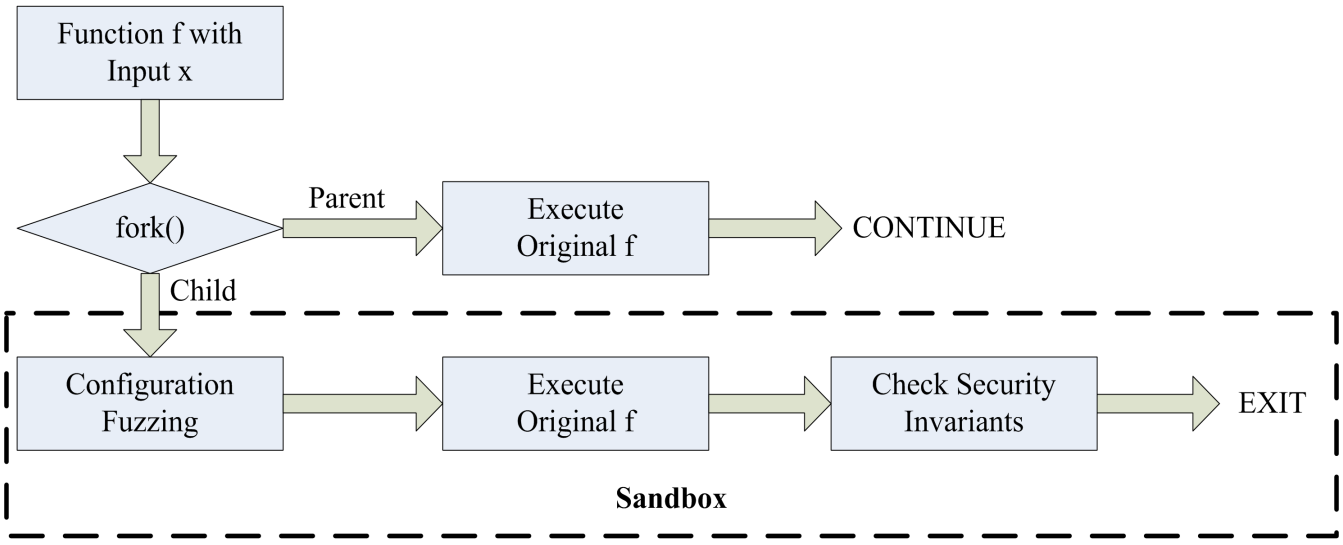

Figure 1: Model of Configuration Fuzzing testing

4.2 Architecture

This section describes the architecture of a framework, called ConFu (Configuration Fuzzing framework for vulnerability detection), and the steps that software testers would take when using ConFu. ConFu mutates the configuration of an application 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 application are run; however, the additional invocations must not affect the user and must run in a separate sandbox.

4.2.1 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 Listing 1, with the corresponding variables and their values in braces. The examples listed are taken from our empirical study in Section 4.3 using psftp[5], an sftp client.

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

```
# 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

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[6]. Also, some early versions of Apache Tomcat allow remote authenticated users to read arbitrary files via a WebDAV write request under certain configurations[7]. 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.

4.2.2 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 in Listing 3. The fuzz_config() function uses a covering array algorithm [11] to ensure a certain degree of coverage when exhaustive exploration of the configuration space is impossible in the lifetime of the software.

```
000
011
101
110
```

Listing 2: A 2-way covering array

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 equals to k will produce full coverage and produce no coverage when equals to zero. The set

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

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 guarantees full coverage. A 2-way covering array will look like Listing 2, we notice that whichever two columns out of the three columns are chosen, all possible pairs of values appeared. Specifically, the pairs 00, 01, 10 and 11 all appear in the rows when we look at the first and second columns only, the second and third columns only, *etc*. This property is called "2-coverage", and corresponds to when 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 permitted. Ideally it would be possible to take t=k, but that might lead to too many configurations to test. Empirical studies [14] show that for most software t need not be more than 6 to find all errors.

```
void fuzz_config()
{
  int r=covering_array();   /*Pick a set of
                              options*/
  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 3: An example fuzzer for psftp

4.2.3 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. Note that a general approach to determine which functions to test is outside the scope of this work. The chosen functions are annotated with a special tag in the source code.

4.2.4 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 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 _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 could be sent to a server for later analysis. Listing 4 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 4: Test function for psftp_connect()

4.2.5 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 5.

```
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 5: Wrapper function for psftp_connect()

4.3 Feasibility

4.3.1 Setup

In order to prove our approach's feasibility of detecting vulnerability, we reproduced a known vulnerability and used ConFu to detect it. The vulnerability we chose is: early versions of OpenSSH do not properly drop privileges when the UseLogin option is enabled, which allow local users to execute arbitrary commands by providing the command to the ssh daemon[8]. The *CVSS Severity*[9] of this vulnerability was 10 (the highest) and it was mainly caused by insufficient testing of the configurations of OpenSSH. We chose this vulnerability not only because its high severity but also because insufficient privilege control is one of the most popular vulnerabilities besides denial of service and directory traversal. The following shows the details of each step in using ConFu to detect the vulnerability.

- Identifying the configuration variables
  The sshd server of OpenSSH 2.1.0 was used as the program-under-test in our feasibility study. And from the configuration file (sshd_config) we found that there are totally 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_authentication, kerberos_or_local_passwd, kerberos_ticket_cleanup, use_login, check_mail.

- 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. do_child() is responsible for creating a session and authenticating the user's identity when a ssh client tries to access the sshd server; it is the most vulnerable function to insufficient privilege control. Note that a wider range of functions can also be tested including the main() function.

```
int check_invariants(input)
{
  if(geteuid()!=input.uid||getuid()!=input.uid)
        /*Log the detection*/
    log(Insufficient privilege control);
  ...   /*Check for other
  ...   security invariants*/
  ...
}
```

Listing 6: Surveillance function for test_do_child()

- Generating test code
  The original function do_child() was renamed to _do_child() and the test function test_do_child() was generated with the fuzz_config() function and the check_invariants() function. The security

[8] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-CVE-2000-0525
[9] http://www.oracle.com/technology/deploy/security/cpu/cvssscoringsystem.htm

invariant for privilege control 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 Listing 6.

- Executing tests
  At last, ConFu generated a wrapper function with the name do_child() which forks a child process to execute the test_do_child() function and runs the original function _do_child() in the parent process.

4.3.2 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 for 10000 times and a fragment of the log file is shown in Listing 7. 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 manage to detect new vulnerabilities, however, this work demonstrates the feasibility of the Configuration Fuzzing approach.

```
UseLogin option is:1
x11_forwarding option is:0
rsa_authentication option is:0
...
...
permit_empty_passwd option is:1
!!Illegal privilege detected!!
//////////////////////////////
UseLogin option is:1
x11_forwarding option is:1
rsa_authentication option is:1
...
...
permit_empty_passwd option is:0
!!Illegal privilege detected!!
```

Listing 7: Log file of the sshd server

4.4 Performance Evaluation

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

4.4.1 Setup

We evaluated our approach's performance by applying it to the psftp client program, which is a part of Putty 0.60 [1], chosen because it is open-source and has multiple configuration options. All experi-

ments 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.

4.4.2 Evaluation

Table 1 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.

# 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 1: Time Cost of psftp_connect() (in seconds) with varying number of tests

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 multi core machine where the test processes can be assigned to another core.

4.5 Limitations

There are two major limitations of ConFu:

1. **Unable to detect zero day vulnerabilities.** ConFu uses surveillance functions to detect vulnerabilities and these surveillance functions are written for known vulnerabilities based on the consequences/characteristics of exploitations. Given a zero day vulnerability, ConFu will not be able to detect it since there is no corresponding surveillance function.
2. **Unable to detect vulnerability related to external resources.** Anything external to the application process itself, *e.g.* database tables, file I/Os, *etc.*, is not replicated by forking the process and the Configuration Fuzzing test run in the forked process is less likely to detect vulnerabilities related to these external resources.

5 Related Work

5.1 Security Testing

One approach to detecting security vulnerabilities is environment permutation with fault injection [12], which perturbs the application environment during the test and checks for symptoms of security violations. Most implementations of this approach, such as [6] and [21], 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, as the errors are independent on the software being injected, 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 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 popular approach is fuzz testing [20]. 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 [13], and is used heavily by security researchers [5]. 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 [8] 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 [9].

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 to test with.

5.2 Configuration Testing

Configuration testing plays an irreplaceable role in security testing. The importance of configuration testing has boosted as more and more vulnerabilities are discovered to be caused by inappropriate configuration. However, most of the popular configuration testing approaches were not designed to reveal security defects. One approach named Rachet [22], which 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 use 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 [15] which is composed with 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 feedbacks to the developer.

Both of the two approaches are able to detect functionality errors when the program-under-test fails, however, vulnerabilities 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. When vulnerabilities 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 of the security invariants.

6 Research Plan and Schedule

6.1 Methodology

To support Configuration Fuzzing, we have developed a prototype of ConFu that supports the generation of test functions, parallel sandboxed execution, and security invariant violation detection. The major tasks of this work include:

- Optimize the way that configurations get mutated. We plan to use a covering array [11] algorithm (guarantee certain coverage) instead of mutating the configuration randomly (cannot guarantee any coverage).
- Define more security invariants of known vulnerabilities, and build a library of surveillance functions with an easy-to-use interface to adapt new security invariants.

- Additional empirical study of the effectiveness of our approach in detecting vulnerabilities will be conducted. We plan to either systematically insert vulnerabilities or use one of the testing benchmarks that have many vulnerabilities and find how many our approach is able to detect.

6.2 Schedule

Table 2 shows my plan for completion of the research.

Completion Date	Work Description	Progress
Aug. 2009	Finish initial ConFu prototype	Completed
Sept. 2009	Performance evaluation for ConFu	Completed
Sept. 2009	Submit paper to SecSE 2010	Completed
Oct. 2009	Reproduce known vulnerabilities and detect them	Completed
Nov. 2009	Write Thesis Proposal	Completed
Dec. 2009	Research on covering array algorithm	
Dec. 2009	Thesis Proposal presentation	
Jan. 2010	Research on security invariants	
Feb. 2010	Finish ConFu framework	
Feb. 2010	Submit paper to the International Journal of Secure Software Engineering	
Mar. 2010	Start writing Thesis	
Apr. 2010	Defend Thesis	

Table 2: Plan for completion of research

7 Future Work and Conclusion

There are a number of future work possibilities, both in the short term and further into the future.

7.1 Immediate Future Work Possibilities

- **Automate the process of locating configuration variables.** 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.
- **Automate the process of identifying functions to test.** In current implementation, testers have to pick functions to test based on their knowledge. However, a better way to choose these functions might be letting ConFu pick functions that are related to the configuration variables being fuzzed.
- **Distributed Configuration Fuzzing testing.** Distributed Configuration Fuzzing testing would increase the efficiency of detecting vulnerabilities, which requires implementing a global coordinator which is in charge of allocating test cases for each users, and collecting and analyzing the results.

7.2 Possibilities for Long-Term Future Directions

- **Find the best variable predictors of vulnerability exploitation.** Monitor the run-time state of an application by collecting general telemetry stream [10] as to find the best variable predictors for vulnerability exploitation instead of defining specific security invariants.
- **Create parallel sandboxes.** Find a different implementation other than forking, which can include file systems and network I/O when creating parallel sandboxes for Configuration Fuzzing tests.

7.3 Conclusion

In this thesis, we will explore approaches for software vulnerability detection in the domain of security testing and develop a framework 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 deployed environment while ensuring a considerable degree of coverage of both the input and configuration 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. We believe that ConFu can help developers build more secure software and improve the security of existing software systems.

7.4 Acknowledgments

I would like to acknowledge the guidance and advice of my thesis advisor Prof.Gail Kaiser. Special thanks to Christian Murphy, who has been helping me from the beginning of this research. I would also like to thank Xuyang Shi, who discussed the initial idea of Configuration Fuzzing with me.

References

[1] Putty: A free telnet/ssh client. `http://www.chiark.greenend.org.uk/sgtatham/putty`.

[2] L. Baresi and M. Young. Test oracles. Technical Report Technical Report CIS-TR01 -02, Dept. of Computer and Information Science, Univ. of Oregon, 2001.

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

[4] 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*, pages 509–512, April 2008.

[5] Toby Clarke. Fuzzing for software vulnerability discovery. Technical Report RHUL-MA-2009-4, Department of Mathematic, Royal Holloway, University of London, 2009.

[6] Wenliang Du and Aditya P. Mathur. Testing for software vulnerability using environment perturbation. In *Proc of International Conference on Dependable Systems and Networks*, page 603, 2000.

[7] Sandro Fouché, Myra B. Cohen, and Adam Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188, New York, NY, USA, 2009. ACM.

[8] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[10] Kenny C. Gross, Aleksey Urmanov, Lawrence G. Votta, Scott McMaster, and Adam Porter. Towards dependability in everyday software using software telemetry. In *EASE '06: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, pages 9–18, Washington, DC, USA, 2006. IEEE Computer Society.

[11] A. Hartman. *Graph Theory, Combinatorics and Algorithms*, volume 34, pages 237–266. Springer US, 2005.

[12] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[13] Leon Jurani. Using fuzzing to detect security vulnerabilities. Technical Report INFIGO-TD-01-04-2006, INFIGO, 2006.

[14] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[15] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 459–468, Washington, DC, USA, 2004. IEEE Computer Society.

[16] 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)*, pages 111–120, 2009.

[17] 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)*, pages 361–376, 2002.

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

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

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

[21] Herbert H. Thompson, James A. Whittaker, and Florence E. Mottay. Software security vulnerability testing in hostile environments. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 260–264, New York, NY, USA, 2002. ACM.

[22] Il-Chul Yoon, Alan Sussman, Atif Memon, and Adam Porter. Effective and scalable software compatibility testing. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 63–74, New York, NY, USA, 2008. ACM.

[23] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.