

# On the value of hybrid security testing

Saad Aloteibi and Frank Stajano

Computer Laboratory  
University of Cambridge  
{firstname.lastname}@cl.cam.ac.uk

**Abstract.** We propose a framework for designing a security tool that can take advantages from current approaches while increasing precision, scalability and debuggability. This could enable software developers to conduct comprehensive security testing automatically. The approaches we utilise are static, dynamic and taint analysis along with fuzzing. The rationale behind this is that the complexity of today’s applications makes the discovery of their vulnerabilities difficult using a single approach. Therefore, a combination of them is what is needed to move towards efficient security checking.

## 1 Introduction

It is well known that ensuring the security of an application usually escapes software producers’ task list during development [1]. Increasing complexity, agile approaches, marketing pressure [2] and other factors could all explain why security is not initially considered. After all, discovering a security vulnerability in an already shipped product can also be economically expensive for vendors either in terms of the cost of patching a single vulnerability, as in Microsoft cases [3], or it could negatively affect their market value when such vulnerabilities become public [2]. Hence, discovering application vulnerabilities is surely of interest to both security researchers and developers but the difference is on their priorities. Whilst, security researchers would like to put the code under their microscope and examine it precisely after having modelled the threats associated with their application, developers would not support this for obvious business reasons and appear willing to sacrifice soundness. Unfortunately, current security testing tools do not accommodate such conflicting priorities in an efficient manner. In this paper, we propose a framework for an automated testing tool that takes into consideration the business needs of software houses while preserving a conservative view. This is done by blending the strengths of current testing approaches and debugging techniques as well as carefully organising this combination to maximise the benefits.

## 2 Current approaches

### 2.1 Static analysis

Static analysis tools emulate the compiler principles to scan the source code for possible anomalies. Techniques include simple search functionality [4], syn-

tactical examination [5, 6] and abstract interpretation [7]. However, all of these suffer from the trade-off between relaxing the tool so that complex software can be analysed while accepting a huge number of false alarms or conducting deeper analysis, which may come at the price of scalability. For example, Flawfinder<sup>1</sup> [5] incorporates about 160 rule-sets of potential vulnerabilities that include buffer overflows, format string problems, race conditions and others. Code is then matched against these rules, and hits are reported to the user in a ranked order. Applying Flawfinder to complex software as Open Office resulted in 13,090 warnings. Scanning through these would require a considerable amount of man hours, not to mention the high volume of false positives. Nevertheless, the static analysis approach has the advantage of covering the whole code without the need for executing the program, which gives it an advantage over other methods [8].

## 2.2 Dynamic analysis

The idea here is to monitor the program behaviour during executions so that precise judgments about the existence of a problem can be made. An example of this is program profiling, where the control flow paths of different test cases are analysed in order to identify codes that may need to be optimised to increase performance or may have latent problems [9]. Although this provides precise results compared with static analysis, it is thought to be inefficient in providing a high level of assurance for two reasons. Firstly, it depends on the test case that the program will run. So, generalisation about the analysis results cannot be made since the monitored behaviour would only apply for this specific test case and not for every possible run as in static analysis. Secondly, examining the behaviour of the program when executing typical input files is not enough on its own since vulnerabilities are usually triggered by coordinating changes that are difficult to produce using this approach.

## 2.3 Fuzzing

Conceptually, fuzzing can be regarded as a form of dynamic analysis. The main principle here is to provide unexpected inputs to the program and monitor its behaviour for the sake of catching bugs [10]. These inputs could be generated randomly<sup>2</sup> or based on grammatical rules that govern the inputs [11]. Interestingly, fuzzing has proved to be effective in revealing vulnerabilities and is being deployed as a component of the development process, as in Microsoft Security Development Life-cycle [12], and also by hacker communities [13]. However, it also inherits the problems of dynamic analysis since code coverage is not guaranteed and because it is highly dependent on the test case provided. Furthermore, fuzzing would be more effective if it were actually directed towards possible attack points which may have unhandled exceptions. Fuzzing alone cannot achieve this and would need to be made smarter by other means.

---

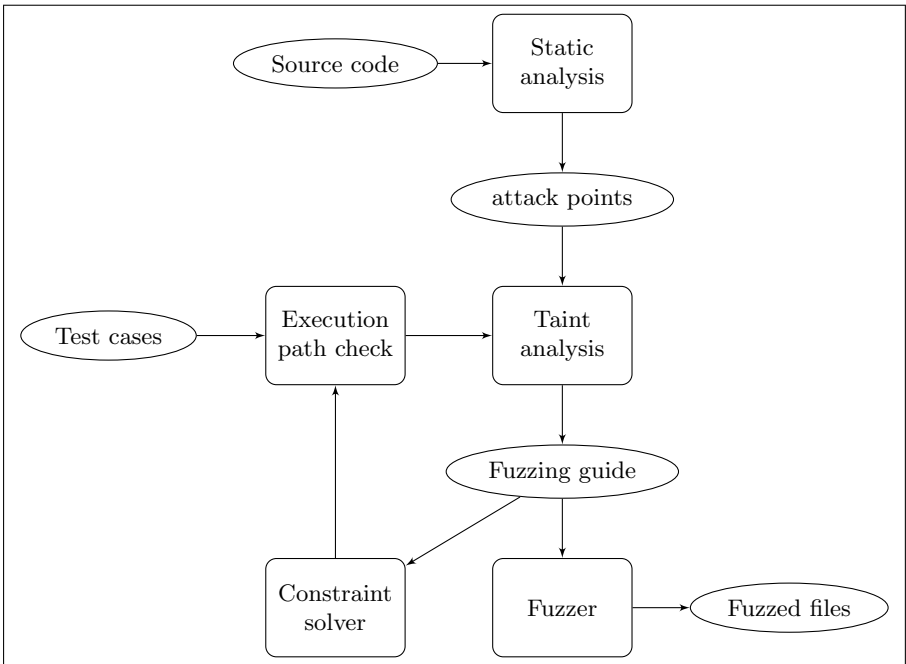
<sup>1</sup> We used version 1.27.

<sup>2</sup> This could be done by taking a valid input file and randomly change bytes on it.

### 3 Hybrid security testing architecture

If we examine software vulnerability reports, we will find that the majority of them are triggered by maliciously crafted inputs. This root cause raises two questions:

- What is the input that triggers the problem?
- What is the instruction that executes the attack?



**Fig. 1.** Hybrid security testing architecture. The fuzzing guide includes flow information, vulnerable instructions and the tainted file.

Neither one of the aforementioned approaches can answer these questions if performed alone. Static analysis might be a good candidate for the second question but would be poor at the first. Dynamic analysis and fuzzing may provide an error-revealing input but it would be the developer’s task to identify the cause of the flaw, which may not be trivial. In our framework, we aim to provide an answer to both questions. Initially, we concentrate on identifying codes that may cause a security breach, even with negligible probability. This is done statically by reasoning over the whole source code. We do not aim for precision here since each warning would automatically be checked at later stages to assess its severity, so we would accept a high rate of false positives and aim to keep the volume of false negatives low. Achieving the former goal would require

three elements. Firstly, a test case that exercises the suspected code<sup>3</sup> would need to be provided. Secondly, it is important to pinpoint which bytes from the test case are used at this particular attack point. Thirdly, these bytes need to be changed to different values that may uncover the vulnerability. For the first element, a corpus of test cases could be provided by the user during testing or it could be crawled from the Internet. For the second, we utilise the concept of taint analysis<sup>4</sup> in order to map between input bytes and the attack point that uses them so they can be mutated. For the third, it is important to collect information about the data type of those bytes that are used at each attack point so that the fuzzer can change them to their maximum, minimum and, also, random values. This would not guarantee that latent vulnerabilities would be uncovered by these values since it is still probabilistic. However, having such information is still beneficial. Figure 1 illustrates the proposed structure.

### 3.1 Scenario description

Firstly, a program would be analysed using the static analysis component. This would result in a list of attack points that could be ranked according to their seriousness and would be fed to the taint analysis stage. Then the user, or the web spider, would be asked to provide a test case for the taint analyser so that bytes that are used at each attack point and at flow control statements could be tagged for later use. Before this, a check for the feasibility of the test case would be conducted. That is, if test case B follows the exact execution path as did test case A, which was used previously, then the tool should not continue with B as it is unlikely it would help in discovering new vulnerabilities or expanding the coverage. The taint analyser would then produce two types of information. Firstly, a list of the attack points that are used at this specific test case and the associated bytes with their data type. Second, a list of tainted bytes that are used at flow control statements. The first list would be an input to the fuzzer, which would take the analysed test case as well. The fuzzer's job is to change the tagged bytes that are used at each single attack point to at least three values (maximum, minimum and random) and produce fuzzed files for each attack point. The reasons behind this are twofold. Firstly, we would like to identify the exact cause of the bug if it exists and, therefore, these points are considered individually. This is similar in concept to delta debugging. Secondly, changing only specific bits would ensure that the generated file would be syntactically valid and, hence, pass the initial parser check. The resulted fuzzed files would then be tested by debugging tools for bugs detection. The second list produced by the taint analyser would be used by a constraint solver along with the tainted file, so that we could automatically expand the coverage of a single test case without the need for using different ones. The expanded test cases would then be checked to decide whether or not such an execution path has been explored previously.

<sup>3</sup> Some might refer to this as a sensitive sink or an attack point.

<sup>4</sup> Taint analysis aims to identify user-driven data that affect values used at security critical instructions or flow control statements.

### 3.2 Essential feature

It is quite difficult with the current fuzzing techniques to decide when enough testing has been performed, either because of the code coverage problem or due to the lack of attack points identification. In our approach, the logging of the following information is required:

- What should be fuzzed? which would be obtained from the static analysis component.
- What has been fuzzed? which is a result of the fuzzer component.

If all the attack points identified initially have already been fuzzed, then the user would be notified that testing has been completed and no further vulnerabilities can be detected. If there are some points that the provided test cases cannot reach, then users would at least know that the testing process did not cover the whole code and further procedures are needed to assure the security of the application.

### 3.3 Motivated example

In the following, we present a security vulnerability that has been detected in the Open Office suite using this approach in manual settings. It is worth mentioning that it is remotely exploitable and has been proved by the developer.

```

1 switch(nOpcode) {
2 // has to pass 131 cases and go to default to reach the code
3 default :
4 // needs not to satisfy 11 if statements and go to the else
   branch
5 // nDataSize type is unsigned long
6 sal_uInt32 nTemp;
7 *pPict >> nTemp ;
8 nDataSize = nTemp;
9 nDataSize+=4;
10 }
```

This example illustrates a clear case of a programmer not following basic secure coding principles. That is, since `nDataSize` is used for allocating memory space and its value is calculated from untrusted input, then it is obvious that it must have been carefully bounded. To exploit this flaw, an attacker should modify the variable `nOpcode` to a particular value that would make the offending code reachable then change the `nTemp` to its maximum value (`0xFFFFFFFF`). This value will be assigned to `nDataSize` and then would result in integer overflow at `nDataSize+=4` statement. The attacker would be able to direct the program to read from a specific location of his choice. Our proposed approach should be able to detect this problem. Firstly, the static analysis tool should produce a warning about the statement at line 9 for possible integer overflow since it is used in memory allocation. The taint analyser would show that `nTemp` is used

at this attack point and pinpoint which bytes correspond to it in the input file. The fuzzer would change only these bytes to the maximum value and, hence, produce the conditions necessary to reveal this bug. The constraint solver would also receive the location of `nOpcode` in the input file and, therefore, the required constraints to reach this code could be negated.

## 4 Related work

A relevant study done by Lanzi et al. proposed a similar idea [14]. However, their use of static analysis is only to acquire basic knowledge about the target application since they focused on executables. This was done via the construction of inter-procedural control flow graphs and loop identifications. Another study was conducted by Ganesh et al., who assumed that users have in mind a set of suspicious areas that they want to test [15]. By default, their method is configured to deal with system and function calls as such. This choice is logical since these libraries are mostly developed by different people and programmers may not understand some of the necessary preconditions that the library developers assume programmers should do. Nevertheless, this method completely depends on the provided test cases. That is, checks for attack points would be conducted for only the provided file but not for the whole program, which may in turn give a false sense of security. Furthermore, the identification of attack points is actually the heart of the problem and leaving it on the user is not desirable. Others in academia [8] and in the commercial field have also briefly and theoretically considered the benefits of merging only static and dynamic analysis.

## 5 Conclusion

If we want to build a robust security testing tool, we must understand that neither one of the current approaches can do this alone. Static analysis would provide code coverage but would also present results that would need to be filtered, requiring considerable effort. Dynamic solutions, including fuzzing, would bring specific results that would not be generalisable. Designers of security testing tools should also take the business needs of software developers into consideration without sacrificing completeness. We show that combining static, dynamic and taint analysis along with fuzzing can be more effective in covering the whole program conservatively while reducing the needs for post-processing. Equally important, we highlight the need for identifying the exact source of the bug so debugging can be efficient. Furthermore, we show that it is necessary to inform the user about whether or not sufficient fuzzing has been done.

## Bibliography

- [1] Pistoia, M., Erlingsson, U.: Programming languages and program analysis for security: a three-year retrospective. *SIGPLAN Not.* **43**(12) (February 2009) 32–39
- [2] Telang, R., Wattal, S.: Impact of software vulnerability announcements on the market value of software vendors - an empirical investigation. In: *Workshop on the Economics of Information Security*, Harvard University, Cambridge, MA (2005) 677427
- [3] Howard, M., Leblanc, D.: *Writing Secure Code*. Microsoft Press, Redmond, WA, USA (2001)
- [4] Chess, B., McGraw, G.: Static analysis for security. *Security Privacy, IEEE* **2**(6) (2004) 76–79
- [5] Wheeler, D.A.: *Flawfinder*
- [6] Viega, J., Bloch, J.T., Kohno, Y., McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code. In: *Proceedings of the 16th Annual Computer Security Applications Conference. ACSAC '00*, Washington, DC, USA, IEEE Computer Society (2000) 257
- [7] Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: *Network and Distributed System Security Symposium*. (2000) 3–17
- [8] Ernst, M.D.: Static and Dynamic Analysis: Synergy and Duality. In: *Workshop on Dynamic Analysis*, Portland, OR, USA (May 2003) 24–27
- [9] Reps, T., Ball, T., Das, M., Larus, J.: The use of program profiling for software maintenance with applications to the year 2000 problem. In: *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering. ESEC '97/FSE-5*, New York, NY, USA, Springer-Verlag New York, Inc. (1997) 432–449
- [10] Sutton, M., Greene, A., Amini, P.: *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional (2007)
- [11] Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation. PLDI '08*, New York, NY, USA, ACM (2008) 206–215
- [12] Microsoft Corporation: *The Microsoft Security Development Lifecycle (SDL): Process Guidance*.
- [13] Microsoft Corporation: *Automated penetration testing with white-box fuzzing*.
- [14] Lanzi, A., Martignoni, L., Monga, M., Paleari, R.: A smart fuzzer for x86 executables. In: *Proceedings of the Third International Workshop on Software Engineering for Secure Systems. SESS '07*, Washington, DC, USA, IEEE Computer Society (2007) 7

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