# A Taint Based Approach for Smart Fuzzing

Sofia Bekrar VUPEN Security Montpellier, France sofia.bekrar@imag.fr Chaouki Bekrar VUPEN Security Montpellier, France bekrar@vupen.com Roland Groz Grenoble University Grenoble, France roland.groz@imag.fr Laurent Mounier Grenoble University Grenoble, France Laurent.Mounier@imag.fr

*Abstract*— Fuzzing is one of the most popular test-based software vulnerability detection techniques. It consists in running the target application with dedicated inputs in order to exhibit potential failures that could be exploited by a malicious user. In this paper we propose a global approach for fuzzing, addressing the main challenges to be faced in an industrial context: large-size applications, without source code access, and with a partial knowledge of the input specifications. This approach integrates several successive steps, and we mostly focus here on an important one which relies on *binary-level dynamic taint analysis*. We summarize the main problems to be addressed in this step, and we detail the solution we implemented to solve them.

Keywords-component: vulnrability detection; smart fuzzing; taint analysis; dynamic analysis.

### I. INTRODUCTION

Software security has become an important issue these last years because of the serious damages that vulnerabilities may cause when they can be exploited by attackers. Currently, different techniques for vulnerability detection exist [1, 2] but fuzzing remains the most efficient way for detecting vulnerabilities in closed-source software. The technique is today widely deployed in industry. It is also adopted, additionally to other advanced methods, by VUPEN Security research team. VUPEN Security is a leading vulnerability research company providing advanced technical analysis and exploitation of security vulnerabilities which enable companies to protect against cyber attacks. In 2010, VUPEN discovered 147 critical vulnerabilities, allowing code execution, in prominent software such as: Microsoft Office and Adobe Acrobat/Reader. Many of those vulnerabilities were discovered thanks to fuzzing. Despite all the flaws found, fuzzing efficiency should be improved especially in industry where we have to face real life constraints namely, the huge target sizes, large inputs, and time constraints. For that, we propose to enhance fuzzing with advanced approaches, notably coverage and taint analysis techniques, to create an innovative approach for zero-day vulnerability detection taking into account those constraints. This approach targets both file processors (for HTML, DOC, PDF formats) and packet processors (TCP/IP). Because the source code of software is often not available to an end user, the method is based on binary analysis. We assume a partial knowledge of the target because input (files, packets) structures are generally known, but not the implementation details. First, we introduce fuzzing and categorize its main strategies. Then, we present our general approach and focus on an important issue, namely dynamic taint analysis.

The paper is organized as follows: *Section II* introduces fuzzing and its main categories. In *Section III*, we discuss our motivations and challenges. The proposed approach is described in *Section IV*. *Section V* and *Section VI* present respectively results to date and ongoing works. Finally, conclusions and future works are given in *Section VII*.

#### II. FUZZING

In this section, we briefly introduce the field of fuzzing. We categorize the various types of fuzzing and put in perspective different strategies of data generation.

Nowadays, fuzz testing is the main security testing approach for detecting serious security vulnerabilities in large software [1, 2]. Many definitions exist in the literature [3, 4, 5, 6]. We adopt here the following synthetic definition: fuzzing is an automatic approach based on injecting invalid or random inputs into a program at execution in order to obtain an unexpected behavior and identify potential vulnerabilities. There is no precise methodology for fuzzing. It depends on the target, and the input format. Different ways to implement this approach are explained in more details later in this paper.

# A. Fuzzing phases

Fuzzing generally respects the following basic steps no matter the target and input format type [2].

1) Target identification: The user identifies the application to test, according to which a tool or an approach is chosen. This target can be a web application, a file processor, a network protocol, etc. The fuzzing approach can vary greatly from one target to another. It is entirely dependent on the target application (file processor or network protocol), the format of the data being fuzzed and their sizes, the number of different inputs that should be tested, and execution time that is required to test the target.

E.g.: Microsoft Word.

2) Input identification: Input type depends directly on the chosen target.

E.g.: To test Microsoft Word, the input will be a DOC file.

3) Fuzzed data generation: Once the input type identified (files, packets, etc), fuzzed data are generated. This can be done in many ways: randomly, by modifying existing valid data, or by modeling the target. This step is the most important in the fuzzing process.

E.g.: Generating inputs to test Microsoft Word may be realized by modifying some bytes on a valid DOC file.

4) Fuzzed data execution: The target is run with new generated test cases. This step may vary depending on the target.

E.g.: Run Microsoft Word by opening the new generated file.

5) *Faults identification:* Observing target under test is a crucial step because it helps identifying and analyzing errors that cause system crashes. A debugger is generally attached to the program under test in order to detect triggered exceptions.

6) *Exploitability evaluation:* Each detected crash should be analyzed to determine whether it is due to an exploitable vulnerability. This step is usually performed manually.

#### B. Fuzzing methods

Generating relevant test cases is the key of fuzzing. A test case can be created either by applying mutations on existing well-formed inputs, or by generating them from scratch according to the internals of the software. The first method is known as Blackbox fuzzing because it does not require any knowledge of the target behavior contrary to the second, Whitebox fuzzing, which assumes a complete knowledge of the application code and behavior to model the target for instance. Blackbox techniques suffer from a lack of precision. Indeed, test cases submitted to the system are not refined and generally not pertinent. Recent works show that it can be more efficient to assume a complete knowledge of the tested application but this is not always possible because sources are generally not available. This limits the use of this approach. Several works based on Whitebox techniques have been proposed. We can mention in particular SAGE (Scalability, Automated, and Guided Execution) [7], a tool to test file formats on Windows. Other tools such as Flayer [8] and Bunny [9] also implement this technique. Gravbox fuzzing, for which we assume a partial knowledge of the target, stands between the two previous methods aiming to take advantage of both. Combining fuzzing and inference is a good example of Graybox fuzzing. The technique aims to get a partial knowledge of the target by providing inputs, observing behavior and analyzing outputs. This knowledge will be used to improve fuzzing by adjusting test cases. Works in this direction have been recently proposed by [10, 11, 12]. They show that this technique gives interesting results in the absence of software source code.

# C. Strategies of Generating Fuzzed Data

There are three main ways to generate test cases.

1) Random: The first and most traditional strategy for creating input data for fuzzing is to use completely random data, without any intelligence or special knowledge of the application. Several vulnerabilities, especially in the past, have been found thanks to this technique [13, 14, 15, 16, 17]. Although it is fast, low cost, and relatively simple to implement, this method usually attacks only the applications surface and thus it provides poor coverage. In [18], it was described as the worst test case design methodology. Another disadvantage is that this method is less efficient when cheksums are involved. This is explained in [19].

2) Mutation of valid data: It is based on modifying existing data i.e. mutating some fields in the original correct input. This technique is theoretically more effective than random generation and is relatively inexpensive to implement (mutations on different fields). Note that its effectiveness depends on the number of valid data available. Some protocol fuzzers use this technique. They apply mutations directly on the network traffic.

3) Model based: Test cases are created according to a previously described model of the target. Test cases correspond to specifications of the application. It can be effective if the specification has been well modeled but can be expensive. This method is implemented in several frameworks: Autodafé [20], Peach [21], Sulley [22], etc.

A comparison of the three strategies is shown in table 1. Mutation based approach seems to be the most appropriate method when software knowledge is not completely assumed.

TABLE I.	FUZZING STRATEGIES COMPARISON

	Random	Mutation based	Model based
Advantages	- Simple - Quick - Low cost	- Relatively simple - Reusable across different software	- Potentially efficient (if the target is well modeled )
Dis- -advantages	- Attacks only the application surface - Useless with checksums - Poor coverage	- Needs numerous valid inputs to get a good coverage	-Time- consuming to set up - Requires knowledge of the format/ protocol - Reusable only with the same format

### III. CHALLENGES

One of the major challenges that we encounter is working at binary level. This is due to the low-level semantics,

diversity, and complexity of assembly instructions. Another difficulty is to identify potentially dangerous executions i.e. the definition of vulnerability patterns that are useful to detect potential vulnerabilities. Generating relevant inputs able to trigger such vulnerabilities is also a challenging issue. In fact, fuzzer efficiency depends on its ability to create effective fuzzed inputs and to produce reliable verdicts. We also need to measure and maximize coverage. In fact, evaluating the amount of code exercised by a fuzzer in the whole target is an important metric for measuring the effectiveness of the fuzzing technique. Those challenges have to be overcome, notably in industry, where we have to deal with numerous constraints namely, the huge software sizes, huge inputs, the absence of source code, the absence of format-specific knowledge, and time constraints. Additionally to these main challenges, we also need to address many other difficulties coming from the techniques used within this general approach. Indeed, for coverage analysis, it appears that none of the existing tools we considered was able to satisfy our needs. As a consequence, we implemented our own techniques and tools. This is also the case with taint analysis for which various frameworks exist but none of them fully satisfy our requirements: the ability to operate without any recompilation (nor access to source code) and portability considerations. Most of the existing implementations do not support the Windows platform. Temu[36] and Dytan[23] are two taint analysis frameworks. The first one is based on PIN [24], a dynamic instrumentation framework running on both Linux and Windows. The implementation of Dytan working on Windows is not useful vet. The second one is a framework based on a virtual machine emulator and cannot be easily integrated into our approach.

# IV. OUR APPROACH

In this section, we present the approach that we proposed [25] which is illustrated by the tool architecture presented on Figure 1. Then, a description of the main principles used and the motivations behind their use are given.

We propose to start by identifying potentially vulnerable sequences of code within the binary code using vulnerability patterns that have to be defined first. The target is then executed taking into account information obtained in the previous step and analyzed dynamically thanks to taint analysis approach. Untrusted data are marked as tainted and their propagation at runtime is tracked. In this way, information flow between sources and sinks are identified. This helps to recognize which parts of the input should be fuzzed in order to generate pertinent test cases and audit the most dangerous paths of the application under test. Fuzzing effectiveness is evaluated through coverage analysis techniques. A method to detect faults is used to monitor executions. A fault does not necessary mean that an exploitable vulnerability exist that is why potential exploitability of each detected fault should be evaluated.

#### A. Vulnerability Pattern

The first step is to identify dangerous functions in the binary by defining vulnerability models, also called "patterns", based on both VUPEN Security expertise in finding and exploiting security vulnerabilities in binaries, and also on already discovered vulnerabilities. A vulnerability pattern represents a model at assembly level that can potentially be the cause of a fault. The function « strcpy » is a well-known example of a vulnerable function at source code which may lead to buffer overflow. In fact, this function can allow the attacker to write outside the bounds of the array, hence overwriting the current return address in the stack to replace its value by the address of a shellcode.

### B. Taint Analysis

This technique is combined with vulnerability pattern detection to identify the most interesting fields of the input that should be fuzzed, and therefore to create the most promising test sequences able to trigger potential vulnerabilities (restricting the test space). Combining those two techniques and applying them at assembly level is an important challenge.

# C. Test Generation

The choice of test values is crucial in fuzzing. For that reason, generating fuzzed data is the most important step in the process. This step can be conducted in different ways as already discussed. In order to generate the most promising test sequences able to trigger particular paths that might reveal faults, we apply a taint analysis at the assembly level to gather information about potentially dangerous data.

# D. Coverage Analysis

Fuzzing effectiveness is evaluated along the process using coverage analysis techniques. Evaluating fuzzing is measuring how well the program is tested and identifying the modification necessary to expand the coverage and try to maximize it.

#### E. Property Checking

Monitoring the program for possible faults at execution is necessary in the fuzzing process. Disassembling and debugging features facilitate such monitoring if they are attached to the program at execution. Here also, working at assembly level makes the task harder because of assembly code complexity.

#### *F. Exploitability Pattern*

Evaluating potential exploitability of a fault is also a challenge. Analyzing after-crash information gathered from the binary, like the stack or the heap content, helps to determine whether this crash could lead or not to an exploit.



Figure 1. Proposed approach.

#### V. RESULTS TO DATE

Based on the approach presented in the above section, we first proposed, developed and evaluated two block-level code coverage techniques able to operate at binary level (without need of the source code), addressing the software under test in a whole (including dynamic libraries), and taking into account the main constraints that have to be faced off in an industrial context: large-sized applications with a wide input domain, unavailability of source code, and minimal knowledge of the input formats and specifications. We analyzed coverage of AcroRd32.dll (+20,000 kilobytes), the most important library in Acrobat Reader, with the same PDF file in the same environment using our two tracers and two others: PinCov [26], ccovtrace [27]. We selected a very large file including some animations and text for our experiments. We repeated this experiment several times in order to calculate the average time required to trace the target with each tool. Exerimental results comparing average execution time showed that the difference in execution time between our two tools and their competitors is important. In fact, our tools have reduced by 50% the required execution time. This can reduce considerably the overhead time when analyzing large suite sizes. Additionally to the fact that they are fast, our tools offer other important features which will be described in more details in another paper.

Our coverage analysis approach can improve fuzzing in two ways: it is used to estimate fuzzing efficiency by measuring how well the program has been tested and identifying the modifications necessary to expand the coverage. It can also be used to maximize coverage and reduce fuzzing time. In fact, it is useful before test generation to find the minimal subset that has an equivalent coverage as the large set of test cases. Therefore, a set providing more important coverage can be tested in minimal time. Coverage is a metric on which fuzzing performance highly depends. That is why reducing test suite size before fuzzing without reducing target coverage is important. For that, we used test suite reduction algorithms. We implemented and evaluated four of the well-known algorithms classically used for test suite reduction according to three criteria, different from the ones considered in most previous studies, and particularly relevant in a fuzzing context: the execution time, the percentage of suite size reduction, and the rate of testing coverage after reduction. All our experiments were performed with a real large application, namely Acrobat Reader.

We implemented a tool to monitor binaries at execution for system exceptions and systematically identify violations at runtime and gather information such as the exception type, and the address of the instruction that causes the exception, to avoid missing hidden vulnerabilities.

Currently, we are working on defining a dynamic taint analysis approach to enhance fuzzing. This approach is presented in the following section.

#### VI. ONGOING WORK: TAINT ANALYSIS

Taint analysis has caught the attention of the security community these last few years [28, 29, 30]. This is highlighted by an increasing use of this technique in the software security domain, and the existence of numerous available frameworks. Taint analysis is a full software vulnerability detection technique which can be performed either statically or dynamically. The key idea behind this analysis is to mark data originating from untrusted sources (E.g.: user input) as *tainted* and determine, at each location in the program, which variables can be influenced by tainted data Determining whether a variable is tainted consists in identifying its possible input dependencies. There are two possible dependencies:

• Data flow dependencies, which correspond to direct assignments between variables in the program. For example, if x is tainted and y is not tainted, the result of the addition x+y is tainted.

```
// x is tainted
y = 2;
z =x+ y;
// Result z is tainted
```

• Control flow dependencies, which correspond to controlling values of program variables through the

control flow determined by conditional statements of the program. For example, a non tainted variable is modified if a tainted variable satisfies a conditional statement.

//x is tainted
if (x > 1) y = 1 else y = 2;
// y is tainted because it is influenced by x

# A. Taint Analysis

We can distinguish two taint analysis approaches:

- Static Taint Analysis is performed mostly at source level, thus covering all the possible execution paths [32, 33]. This approach allows in principle a complete analysis and deals with all possible runtime cases. However, most of the time, application sources are not available, and operating at the binary-level raises serious difficulties. Another general problem is that decidability is generally achieved by means of abstract interpretation techniques, leading to over-approximate the program behavior, which may generate a lot of false positives.
- Dynamic Taint Analysis consists in analyzing code during its execution. Each object from user input (e.g.: network, files, etc) is marked as insecure. This taint allows us to track the influence of tainted objects along the execution of the program. This taint can be performed without access to application sources. Although in theory the approach should give interesting results, it is very complex to implement. In addition, implicit flows are not available at runtime. Therefore, it is not possible to take advantage of the full control flow information. Moreover, the program needs to be executed with specific inputs and hence cannot cover all possible executions [34, 35, 36, 37].

# B. Proposed Dynamic Taint Analysis Approach for Smart Fuzzing

#### *1) Taint analysis for improving test case quality*

As already discussed, there are three main fuzzing strategies: *random*, *generation* and *mutation*. A generation based fuzzer is a self-contained program that generates its own invalid inputs based on the target model. A mutation fuzzer takes a valid input and mutates the sample to create many invalid sessions. The main problem is how to know which parts of the valid input should be mutated in order to create relevant test cases able to trigger faults. Vulnerability is exploitable if its execution can be triggered by user but how to detect parts of the input that *influence* the target. The idea behind using taint analysis to improve fuzzing is to identify the specific offsets that taint a specific program scope (interesting functions) and then to mutate only those interesting parts.

Despite the widespread usage of taint analysis, there has been little effort to apply it to binaries and to summarize the critical issues that arise when these techniques are performed at assembly level. Indeed, we do not only have to deal with the open taint analysis challenges, but also with the complexity of assembly language. Different errors can occur in taint analysis. First, marking a variable as tainted when it is not derived from a tainted source. This will typically result in generating false positives. Second, missing the information flow from a source to a sink, and then generating false negatives.

#### a) Taint propagation rules

The main issues to address when using taint analysis to analyze a program execution are the following: new taint introduction, taint propagation as instructions execute, taint checking during execution, and taint elimination. At the binary level, two objects can be tainted: memory locations and registers, as shown in figure 2.



Figure 2. Taint propagation.

The quality of information obtained thanks to taint analysis depends on the markings that can be either bit-precision or byte-precision. Although the higher degree of precision provided with bit-precision markings, the level of details provided with byte-precision markings are accurate enough for our purposes.

- Memory locations (memory addresses): to taint memory locations we have to keep track of
  - The initial address of the memory location
  - The size of memory location to be tainted
- Registers: we have to keep track of all registers (32, 16, and 8 bits registers)
  - Register name
  - Register content
  - Register taint value (tainted/untainted)

Everything that is controlled by user is tainted.

• Object X is tainted if X is influenced by the value of a tainted object Y. We say: Y tainted X and write X→ t(Y)

E.g. mov eax, ecx

- // ecx tainted => eax tainted
- If an object is influenced by an object derived from a tainted object, the first one is tainted because of the transitivity

$$X \rightarrow t(Y)$$
 and  $Y \rightarrow t(Z) \Longrightarrow X \rightarrow t(Z)$ 

An object is tainted if it was:

• assigned from an unstruted source

E.g.: mov eax, userbuffer[ecx]

- assigned from a tainted object
- E.g.: add eax, eax (eax tainted)

Taint should be deleted if the object was:

- assigned from an untainted object
   E.g.: mov eax, ecx (ecx not tainted)
   assigned from a constant
- E.g.: mov eax, 0x10
- assigned from a tainted object but the assignment result is a constant.
   E.g.: xor eax, eax

Applying dynamic taint analysis at binary level is a challenging task because of the variety of cases that we have to deal with. To implement the technique, we have to automatically:

> • Identify all operands of each instruction Difficulty: An instruction can have from one to four operands.

• Identify each operand type (source/destination)

Difficulty: An operand can be:

- -A register: there are nine 32-bit registers, four 16-bit registers, and eight 8-bits registers: eax (32 bits), ax (16 bits), al and ah (8 bits).
- -An address: [ebp+4], or [eax+edx-4] -A value
- Track each tainted object

Difficulty: deal with all propagation types and taint in all the following cases: memory-register, register-register, and register-memory.

• Understand the semantic of each instruction Difficulty: There are Hundreds of x86 different instructions (mov, add, xch, etc), and at least one propagation rule for each instruction.

#### b) Implementation and preliminary results

We first run the target with a well-formed input and monitor execution to identify how the program uses user inputs. These data are marked as tainted and their propagation tracked along the execution of the program. At each execution a taint trace is gathered. It contains information about taint propagation during program execution i.e. all tainted objects, and all input fields that tainted these objects at each instruction of the program. Figure 3 is a simple representation of how dynamic taint analysis maps file offsets to instructions influenced by tainted data. We associate to each variable, not only, a boolean value (tainted\untainted), but also offsets in the input instance that tainted this variable, monitoring how input data influences the program. Consequently, input locations that could affect potentially dangerous locations such as strcpy(), or memcpy() functions are automatically identified. We tested our tool with small executables containing different functions. The output generated with our dynamic taint analysis program contains all executed instructions at runtime with the address of each instruction, its operands, and all tainted registers at this instruction. A list of tainted memory locations is also maintained at each instruction and updated if necessary. In this way, all tainted objects and offsets that taint them are identified at each instruction. A simple example is the memcpy () example. If memcpy(), which is considered as a dangerous function, is called with a tainted source, taint is propagated to destination, and sensitive fields are automatically identified . The second step of the approach is to mutate sensitive fields to generate relevant new inputs. Mutating the offsets associated to the source argument of the memcpy() function could generate an interesting test case able to trigger a fault at this instruction.



Figure 3. Mapping file offsets to tainted instructions/functions.

Once the mapping is available, fuzzing is driven using this mapping, and offsets that taint specific functions identified as potentially dangerous (thanks to vulnerability patterns) are fuzzed, generating in turn new inputs that are fed to the target in order to trigger faults. This process is given in figure 4.



Figure 4. Mapping file offsets to tainted instructions/functions.

#### 2) Taint analysis for exploitability

Dynamic taint analysis not only helps to create relevant test cases able to trigger faults, but also helps to evaluate the exploitability level of each detected fault because a detected

fault does not necessary mean that an exploitable vulnerability exists. Dynamic taint analysis is thus also involved in the "exploitability pattern" step of our global approach. Determining the root cause of a fault in order to understand the problem and exploit the vulnerability is today a very hard and time consuming task. Taint information provided by dynamic taint analysis can be very useful to alleviate this problem. In fact, an offline data flow analysis of the collected taint trace of an execution leading to a program fault can be performed automatically to determine the origin of the fault. Potentially dangerous inputs are tainted, and taint is propagated along the execution of the program thanks to dynamic taint analysis which also records trace files. The idea is to implement a technique for backward slicing of traces to parse and analyze those trace files and extract conditions leading to a fault. Detailed information about each executed instruction in a program, starting from the input and going up to a crash, should be extracted. Backward slicing can be considered as a complementary step to data tainting: data tainting taints and propagates taint from an attacker-controlled input to determine what it affects and generate associated inputs, and slicing starts analyzing from the crash to understand its causes. Currently, there is few tools, such as the binary analysis tool BitBlaze [36], and !exploitable [38] to help determining whether a crash is caused by a potentially exploitable vulnerability or not. However these tools do not satisfy our needs. BitBlaze is based on a virtual machine which makes its use complex and !exploitable does not always provide relevant information because it assumes that all data are tainted from the state where the fault is triggered.

# VII. CONCLUSION AND FUTURE WORK

In this paper, first we introduced the field of fuzzing and present briefly our general approach [25]. Then we focused on an important issue, namely dynamic taint analysis, that we are currently exploring in order to improve fuzzing efficiency. Dynamic taint analysis aims to map some locations in the program to associated specific fields of the user input. It also helps to determine causes of a fault and thus to determine its exploitability. This works takes place in a larger perspective whose objective is to set up a fuzzing environment for software vulnerability detection according to the general approach that we discussed in [25]. Our objective is first to experiment the approach on real and large-sized applications with a wide input domain. We also plan to combine dynamic taint analysis and static analysis in order to further improve the mutation step. And, finally, we will integrate the results obtained into our fuzzing environment.

#### AKNOWLEDGEMENT

This work is supported by VUPEN Security and University of Grenoble. We would like to thank VUPEN Security for funding our work and for allowing us to publish these results.

# References

- [1] A. Takanen, J. DeMott, and C. Miller, "Fuzzing for software security testing and quality assurance". Artech House Publishers, 2008.
- [2] M. Sutton, A. Greene, P. Amini. "Fuzzing: Brute Force Vulnerability Discovery". United States. Addison–Wesley Professional, 2007.
- [3] Virus.org Mailing List Archive, 2007. http://lists.virus.org/fuzzing-0703/msg00014.html.
- [4] P. Ochlert, "Violating assumptions with fuzzing," Security & Privacy, IEEE, vol. 3, no. 2, pp. 58–62, 2005.
- J. Koziol, "Ultimate fuzzers," 2005. http://www.infosecinstitute.com/blog/2005/12/fuzzers-ultimate-list.html.
- [6] G. Zhao, W. Zheng, J. Zhao, and H. Chen, "An heuristic method for web-service program security testing," in ChinaGrid Annual Conference, 2009. ChinaGrid '09. Fourth, 2009, pp. 139–144.
- [7] P. Godefroid, M. Y. Levin, D. A. Molnar, "Automated Whitebox Fuzz Testing". In NDSS, The Internet Society 2008.
- [8] W. Drewry, T. Ormandy, "Flayer: exposing application internals". In 1st USENIX Workshop on Offensive Technologies (WOOT '07), August 6, 2007, Boston, MA, USA, 2007.
- [9] Bunny the Fuzzer. http://code.google.com/p/bunny-the-fuzzer/.
- [10] G. Shu et D. Lee. Testing security properties of protocol implementations – a machine learning based approach. In ICDCS. IEEE Computer Society, 2007.
- [11] G. Shu, Y. Hsu, D. Lee. "Detecting communication protocol security flaws by formal fuzz testing and machine learning". FORTE 2008, pages 299–304, 2008.
- [12] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, J. Röning, "Experiences with Model Inference Assisted Fuzzing". In Dan Boneh, Tal Garfinkel, and Dug Song, editors, WOOT. USENIX Association, 2008.
- [13] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," Communications of the ACM, vol. 33, no. 12, pp. 32–44, 1990.
- [14] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services". Citeseer, 1995.
- [15] J. Forrester and B. Miller, "An empirical study of the robustness of windows nt applications using random testing," in Proceedings of the 4<sup>th</sup> conference on USENIX Windows Systems Symposium-Volume 4, 2000, pp. 6–6.
- [16] N. Kropp, P. Koopman, and D. Siewiorek, "Automated robustness testing of off-the-shelf software components," in Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on. IEEE, 1998, pp. 230–239.
- [17] B. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," in Proceedings of the 1st International workshop on Random testing. ACM, 2006, pp. 46–54.
- [18] G. Myer, "The art of software testing," A Willy-Interscience Pub, pp. 1979.
- [19] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in Security and Privacy (SP), 2010 IEEE Symposium on, May 2010, pp. 497–512.
- [20] M. Vuagnoux. Autodafé: an Act of Software Torture. In 22th Chaos Communication Congress, Berlin, Germany, 2005. http://autodafe.sourceforge.net/docs/autodafe.pdf.
- [21] Peach Fuzzing Platform. http://peachfuzzer.com/.
- [22] Sulley. http://code.google.com/p/sulley/.
- [23] J. A. Clause, W. Li, A. Orso, "Dytan: a generic dynamic taint analysis framework". ISSTA 2007:196-206.
- [24] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized

program analysis tools with dynamic instrumentation", In Sarkar and Hall, pages 190-200.

- [25] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 427–430.
- [26] http://code.google.com/p/pincov/.
- [27] http://www.ruxcon.org.au/assets/Presentations/ben-nagy.prospectingfor-rootite.2010.pdf.
- [28] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.
- [29] D. Zhu, J. Jung, D. Song, T. Kohno, D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking." Operating Systems Review (SIGOPS) 45(1):142-154 (2011)
- [30] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," Acm Sigplan Notices, vol. 40, pp. 213-223, Jun 2005.
- [31] D. M. Koushik Sen, Gul Agha, "CUTE: A concolic unit testing engine for C," 13th ACM SIGSOFT 2005.

- [32] D. Ceara, L. Mounier, M-L Potet: Taint Dependency Sequences: A Characterization of Insecure Execution Paths Based on Input-Sensitive Cause Sequences. ICST Workshops 2010:371-380.
- [33] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2006, pp. 269-278.
- [34] E. J. Schwartz, T. Avgerinos, D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". IEEE Symposium on Security and Privacy 2010:317-331
- [35] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," 2009, pp. 474-484.
- [36] D. Song, et al., "BitBlaze: A New Approach to Computer Security via Binary Analysis," Information Systems Security, Proceedings, vol. 5352, pp. 1-25 307, 2008.
- [37] M. Kang, S. McCamant, P. Poosankam, D. Song: DTA : Dynamic Taint Analysis with Targeted Control-Flow Propagation. NDSS 2011.
- [38] http://msecdbg.codeplex.com/.