ISSN : 0976-8491 (Online) | ISSN : 2229-4333 (Print)

An Overview of Software Vulnerability Detection

[1]Yunfei Su, [2]Mengjun Li, [3]Chaojing Tang, [4]Rongjun Shen
[1,2,3,4]School of Electronic Science and Engg., National University of Defense Tech., Changsha China

Abstract
Software vulnerability is the main cause of computer security problems and software vulnerability detection is a research hotspot recently. A lot of research has already been done regarding detection techniques, models, tools and all of them are covered in literatures. The main purpose of this paper is to provide a comprehensive survey and analysis of past and current research directions, including static analysis, fuzzing, taint analysis, symbolic execution and hybrid methods. Besides, this paper also provides an analysis and comparison of different tools and talks about the future direction of this field.

Keywords
Software Vulnerability Detection, Static Analysis, Fuzzing, Taint Analysis, Symbolic Execution

I. Introduction
With the development of technology, information systems are widely used in our life, and software security tends to be of great concern. The prevalence of software enforces the software industry to think of how to build quality in. Software quality is most related to the knowledge and experience of the developers. Unfortunately, developers make mistakes that lead to vulnerable and defect software. Exploited security vulnerabilities can cause drastic costs, e.g., system crash or the modification of data. A high proportion of all software security incidents is caused by attackers who exploit vulnerabilities.
Software vulnerability is defined as a flaw in software systems which causes a computer software or system to crash or produce invalid output or to behave unintended way. Vulnerability detection [1] is the process of confirming if a system contains flaws that could be leveraged by an attacker to compromise the security of the system or that of the platform the system runs on. In comparison to other approaches to security, such as intrusion detection and prevention, the focus of vulnerability detection is on identifying and eventually correcting flaws, rather than detecting and blocking attacks that exploit a flaw.
Identifying vulnerabilities and patching them is a widely applied measure to evaluate and improve the security of software. Due to the openness of modern software-based systems, applying appropriate security techniques is of growing importance and essential to perform effective and efficient vulnerability detecting. Therefore, an overview of vulnerability detection techniques is of high value both for researchers to evaluate and refine the techniques and for practitioners to apply and disseminate them. This paper fulfills this need and provides an overview of recent vulnerability detection techniques. For this purpose, it first summarizes the required background of vulnerability detection. Then, basics and recent developments of vulnerability detection techniques applied, i.e., static analysis, fuzzing, taint analysis, symbolic execution, as well as hybrid methods are discussed. Finally, the tools released for vulnerability detection are compared with their detection techniques in a table.

II. Static Analysis
Static analysis is a way of analyzing the source code (or the binary code) without actually executing programs, thus avoiding risks linked to the execution of malicious programs. Static analysis techniques can analyze all control flows of a program. Therefore, static analysis approaches achieve, compared to dynamic test approaches, a significant higher coverage of program under analysis and, thus, produce a significant lower false negative rate. In other words, if there is vulnerability in the application under test, in most cases the analysis is able to find it. Mostly, static analysis tools detect vulnerabilities by scanning the program source code, a significant part of efforts in static vulnerability detection have been directed towards analyzing software written in some high-level language, such as C, C++, C#, Java, or PHP. It is a very effective method for detecting programing related vulnerabilities early in the software development life cycle.
Static analysis techniques make the detection process fast, repeatable and can deal with various vulnerabilities. However, the approximate nature of the results provided by static analysis makes it difficult to eliminate false positives.
The techniques that can be used in static vulnerability detection include lexical analysis, data flow analysis, abstract interpretation, model checking.

A. Lexical Analysis
Lexical analysis is also called as grammar structure analysis [2] or pattern matching or syntactic analysis. Flawfinder [3], ITS4 [4], PMD [5], RATS [6] and Findbugs [7] are based on lexical analysis. lexical analysis divides the program into a tokenized stream and searches for a predefined set of vulnerable functions or patterns. For examples, lexical analysis can detect the use of potentially insecure C functions, like strcpy(), strcat() etc.
The speed of lexical analysis is simple and fast. But, its drawback is, this method may produce a massive amount of false positives. It is because this method does very simple analysis and ignores the flow of data through the program.

B. Data Flow Analysis
Data flow analysis is used in compilers to optimize programs. It uses a control flow graph to check the possible set of values calculated at various program points. Data flow analysis can also be used in vulnerability detection. Jlint [8], Findbugs [7], Parfait [9] are based on data flow analysis.

C. Abstract Interpretation
Abstract interpretation is introduced by Patrick Cousot and Radhia Cousot[10] in1978. The abstract interpretation relies on the notion of approximation. It is alsosometimes so called as a theory of semantics approximation. According to this theory, all possible values a variable can take on a certain program point can be approximated by a set that can be compactly represented as an interval. The notion of approximation in abstract interpretation is defined by Galois connection and extrapolation is used for ensuring the termination of cyclic systems [11]. Astree [12] and Frama-C [13] are based on abstract interpretation.

D. Model Checking
The model checking is the automatic technique which helps

to check if the property holds for the given state of the model. Usually, the inputs for model checkers which are expressed as formulas of temporal logic are analyzed and checked to see if the program properties are retained. In practice, sometimes it becomes infeasible to check all the system states, for commercial software having millions LOC and a state-explosion problem may arise. CBMC [14], Java Pathfinder [15], SLAM [16], BLAST [17] are based on model checking.

III. Fuzzing

Fuzzing or fuzz testing is a program testing technique that is based on the idea offeeding random inpputs to a program until it crashes. It was proposed in 1990 by Barton Miller at the University of Wisconsin [18]. Since then, fuzz testing has been proven to be an effective technique for finding vulnerabilities in software.

Data generation is the key to fuzzing, according to the data generation methods, fuzzing can be categorized as random fuzzing, mutation-based fuzzing, generation-based fuzzing and direction-based fuzzing.

Random fuzzing is the simplest fuzz testing technique, a stream of completely random input data is send to the program under test. The input data can be sending as command line options, events, or network packets. This type of fuzzing is, in particular, useful for test how a program reacts on large or invalid input data. While random fuzzing can find already severe vulnerabilities, modern fuzzers do have a detailed understanding of the input format that is expected by the program under test.

Mutation-based fuzzing is one type of fuzzing in which the fuzzer has some knowledge about the input format of the program under test: based on existing data samples, a mutation-based fuzzing tools generated new variants, based on a heuristics, that it uses for fuzzing. The mutation algorithm is the key to improve the efficiency of fuzzing.

Generation-based fuzzing generates program inputs according to some specifications. Compared to pure random-based fuzzing, generation-based fuzzing achieves usually a higher coverage of the program under test, in particular if the expected input format is rather complex and has checksums.

Direction-based fuzzing use the program control flow to direct the fuzzing, also called testcase generation fuzzing. SAGE [19] is the type of Direction-based fuzzing. First, it constructs an initial and valid input IN0, sends the input into program P, and symbol execution engine observes P's processes on IN0 and a path constraint that is in the form of logical formulas; secondly, it negates the path constraint encountered during execution, solves new constraint by a constraint solver, and create a new input IN1 whose execution path is different from IN0's; finally, it processes IN1 in the same way with IN0 and repeats the previous three procedures.

There are lots of research [20] and tools on fuzzing, such as Sulley [21], SPIKE [22], Peach [23], Bestorm [24], Spider Pig [25] and so on. State-of-the-art testing of large distributed systems often relies in practice on fuzzing. Unfortunately, this approach suffers from the fact that the space of possible inputs is extremely large and the efficiency is low.

IV. Taint Analysis

During the process of taint analysis, no matter the data is malicious or not but all the input data that comes from unknown and untrusted sources are marked as tainted and traced to check if the tainted input data is used at sink point, such as an API that converts string data into executable code. A significant portion of today's security vulnerabilities are string-based code injection vulnerabilities, which enable the attacker to inject data into dynamically executed programming statements, which leads to full compromise of the vulnerable execution context. Examples for such vulnerabilities include SQL Injection and Cross-Site Scripting. The taint analysis has two forms: dynamic or static.

A. Dynamic Taint Analysis

The approach used in dynamic taint analysis is to mark the data originating from untrusted input as tainted. The analysis keeps track of all the tainted data in the memory and when such data is used in a dangerous situation, a possible bug is detected. This approach offers the capabilities to detect most of the input validation vulnerabilities with a very low false positive rate. However there are some disadvantages when using dynamic taint analysis. The execution of the program is slower because of the necessary additional checks and the problems are detected only for the executions path that have been executed until now (not for all executable paths) which can lead to false negatives.

BitBlaze [26] a binary analysis platform which combines static analysis techniques with dynamic analysis techniques, mixing concrete and symbolic execution, system emulation and binary instrumentation. One of the dynamic techniques implemented by BitBlaze is taint analysis used for detecting overwrite attacks.

BuzzFuzz [27] an automated white box fuzzing tool which, unlike standard fuzzing tools, uses dynamic taint tracing to automatically locate regions of original input files that influence values used at key program attack points. New input files are generated by fuzzing the identified taint regions. Because it uses taint analysis to automatically discover and exploit information about the input file format, it is especially appropriate for testing programs that have complex input file formats.

B. Static Taint Analysis

Static taint analysis is the technique used for detecting the over-approximation of the set of instructions that are influenced by user input. This set of tainted instructions is computed statically only by analyzing the sources of the program. The main advantage for static taint analysis is that it takes into account all the possible execution paths of the program. On the other hand the analysis may not be so accurate as the one performed dynamically because the static analyzer does not have access to the additional runtime information of the program.

Parfait [9] is a static multi-layered program analysis framework. It uses static taint analysis in its preprocessing stages. The approach used by Parfait is to reduce the taint analysis to a graph reachability problem.

V. Symbolic Execution

Symbolic execution [28] is a technique used to exercise various code paths through a target system. Symbolic execution works as follows: instead of running the target system with concrete input values, a symbolic execution engine replaces the inputs with symbolic variables, that are initially allowed to be anything, and then runs the target system. Whenever the system execution branches based on a symbolic value (that depends on symbolic inputs), the symbolic execution engine forks, following each branch and adding constraints on the symbolic variable in the branch node. Thus, each execution path through the target system will have associated a set of constraints on the symbolic inputs that need to be satisfied in order for the execution path to be feasible. The set of constraints can be "solved", generating a set of concrete

inputs that would exercise the respective path.
While symbolic execution the number of feasible paths in a program grows exponentially with an increase in program size which finally leads to path explosion [29]. Another problem degrading symbolic execution is environment interactions. Programs interact with their environment by performing system calls, receiving signals, etc. Consistency problems may arise when execution reaches components that are not under control of the symbolic execution tool [30].
Concolic execution[19, 31-32] is a hybrid program execution that performs symbolic execution, along with a concrete execution (with particular inputs) path. Starting with a concrete input, concolic execution symbolically executes the program, gathering input constraints from conditional statements encountered along the way.
KLEE [29] is an open source symbolic execution tool to analyze programs and automatically generate system input sets that achieve high levels of code coverage. KLEE is specifically designed to support the testing of applications that interact with their runtime environment. KLEE was used to test the GNU Coreutils suite of applications, which form the basis of the user environment on many different UNIX like systems. KLEE's symbolic execution engine accepts programs that have been compiled to Low Level Virtual Machine (LLVM) byte code which it then symbolically executes with two goals. First, it attempts to touch every line of executable code in the program. Second, at each potentially dangerous operation, such as memory dereferencing, if any of the possible input values can cause an error.
MAYHEM [33] is tool for automatically finding exploitable bugs in binary programs in an efficient and scalable way. MAYHEM introduces a novel hybrid symbolic execution scheme that combines the benefits of existing symbolic execution techniques (both online and offline) into a single system. Index-based memory modeling is proposed in MAYHEM, a technique that allows MAYHEM to discover more exploitable bugs at the binary-level.
S2E[30] is a platform based on symbolic execution for analyzing the properties and behavior of software systems. The S2E platform reuses parts of the QEMU virtual machine, the KLEE symbolic execution engine, and the LLVM tool chain. S2E currently runs on Mac OS X, Microsoft Windows, and Linux, it can execute any guest OS that runs on x86, and can be easily extended to other CPU architectures, like ARM or PowerPC. In S2E the path explosion and environment interactions problems are alleviated by selective symbolic execution.
AEG [34] introduced a fully automatic end-to-end approach for automatic vulnerability exploit generation. A novel preconditioned symbolic execution technique and path prioritization algorithms for finding and identifying exploitable bugs are developed. AEG analyzed 14 open-source projects and successfully generated 16 control flow hijacking exploits, including two zero-day exploits for previously unknown vulnerabilities.

VI. Hybrid Method

Static analysis, such as lexical analysis, data flow analysis, abstract interpretation, static taint analysis and model checking, can be utilized to detect vulnerabilities in code without code execution. It is a fast and scalable technique for scanning millions of lines of code during the analysis and get a high coverage of code, but it suffers the high false positives. Dynamic analysis, such as fuzzing, symbolic execution, dynamic taint analysis, needs the code to be run during analysis. It suffers state-explosion, high cost and low coverage and efficiency problems, but it gets a low false positives. It is a natural idea to combine them to complement each other.
Rawat et al. [35] present a hybrid approach for buffer overflow detection in C code. The approach makes use of static and dynamic analysis of the application under investigation. The static part consists in calculating Taint Dependency Sequences (TDS) between user controlled inputs and vulnerable statements. This process is akin to program slice of interest to calculate tainted data- and control-flow path which exhibits the dependence between tainted program inputs and vulnerable statements in the code. The dynamic part consists of executing the program along TDSs to trigger the vulnerability by generating suitable inputs with a fitness function.
SANTE (Static ANalysis and TEsting) [36] is tool for verification of C programs. In SANTE heterogeneous techniques such as abstract interpretation, dependency analysis, program slicing, constraint solving and test generation are combined within one tool. It can be used to detect the risks of division by zero, out-of-bounds array access and some cases of invalid pointers.
FLINDER-SCA [37] proposed combined verification approach to detect recent vulnerabilities at the source code level with reasonable amounts of efforts and computing time. It includes three steps. First, abstract interpretation and taint analysis are used to detect potential vulnerabilities (alarms), then program slicing is applied to reduce the initial program, and finally a testing step tries to confirm detected alarms by fuzzing on the reduced program. We describe the proposed approach and the tool, illustrate its application for the recent OpenSSL/ HeartBeat Heartbleed vulnerability.
SMASH [38] presented a unified framework for compositional may-must program analysis and a specific algorithm. SMASH was implemented using predicate abstraction for the may part and using dynamic test generation for the must part. The key technical novelty of SMASH is the tight integration of may and must analyses using interchangeable not-may/must summaries. Results of experiments with 69 Microsoft Windows Vista device drivers show that SMASH can significantly outperform may-only, must-only and non-compositional may-must algorithms.
Driller[39] is a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs. By combining the strengths of the two techniques, driller mitigates their weaknesses,avoiding the path explosion inherent in concolic analysis and the incompleteness of fuzzing. Driller evaluated 126 applications released in the qualifying event of the DARPA Cyber Grand Challenge and shown its efficacy by identifying the same number of vulnerabilities, inthe same time, as the top-scoring team of the qualifying event.

VII. Comparison of Tools in Vulnerability Detection

In Table 1 we list the main tools developed for vulnerability detection and mark the techniques they used, the code form they detect. LA, DFA, AI, MC, F, TA, SE, SC, BC separately stands for Lexical Analysis , Data Flow Analysis, Abstract Interpretation, Model Checking, Fuzzing, Taint Analysis, Symbolic Execution, Source Code and Binary Code.

Table 1: Comparison of Tools in Vulnerability Detection

Tools	LA	DFA	AI	MC	F	TA	SE	SC	BC
ITS4	√							√	
SPLINT[40]		√						√	
PMD	√							√	
FindBugs	√	√						√	
RATS	√							√	
FlawFinder	√							√	
Jlint		√						√	
Parfait		√				√		√	
Astree			√					√	
Frama-C			√					√	
CBMC				√			√	√	
JPF				√				√	
SLAM			√	√				√	
BLAST			√	√	√			√	
SPIKE					√				√
Sulley					√				√
Peach					√				√
Bestorm					√				√
Spider Pig					√				√
BitBlaze						√	√		√
BuzzFuzz					√	√		√	
DART[31]					√		√	√	
CUTE[32]					√		√	√	
EXE[41]							√	√	
SAGE					√		√		√
KLEE							√	√	
AEG							√	√	
MAYHEM						√	√		√
S2E							√		√
SANTE			√				√	√	
FLINDER-SCA			√			√	√	√	
SMASH			√				√	√	
Driller					√		√		√

VIII. Conclusion and Future Direction

In this paper, we provided an overview of vulnerability detection techniques. For this purpose, we first summarized the required background on software vulnerability detection. Then we discussed the typical vulnerability detection techniques such as Lexical Analysis, Data Flow Analysis, Abstract Interpretation, Model Checking, Fuzzing, Taint Analysis, Symbolic Execution and hybrid methods. At last, we compared the different tools with their detection techniques employed.

From the survey, we can conclude that different techniques have different advantages and disadvantages, single technique is not sufficient for vulnerability detection, combining different techniques and complementing each other is the main direction in future.

References

[1] Cova, M., Felmetsger, V., Banks, G., Vigna, G.,"Static detection of vulnerabilities in x86 executables", In Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual, pp. 269-278. IEEE, 2006.

[2] L. Peng, C. Baojiang,"A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools", IEEE, Beijing, China, 2010.

[3] Flaw Finder: [Online] Available: http://www.dwheeler.com/flawfinder

[4] Viega, John, Jon-Thomas Bloch, Yoshi Kohno, Gary McGraw,"ITS4: A static vulnerability scanner for C and C++ code", In Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference, pp. 257-267. IEEE, 2000.

[5] PMD: [Online] Available: http://pmd.sourceforge.net/

[6] RATS: [Online] Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[7] Findbugs Hovemeyer, David, William Pugh,"Finding bugs is easy", ACM Sigplan Notices 39, No. 12 (2004), pp. 92-106.

[8] Jlint [Online] Available: https://sourceforge.net/projects/jlint/

[9] Cifuentes, Cristina, Bernhard Scholz,"Parfait: designing a scalable bug checker", In Proceedings of the 2008 workshop on Static analysis, pp. 4-11. ACM, 2008.

[10] Cousot, Patrick, Radhia Cousot,"Abstract interpretation: past, present and future." Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). ACM, 2014.

[11] Viladrosa, Robert Clarisó,"Abstract interpretation techniques for the verification of timed systems", PhD diss., Universitat Politècnica de Catalunya, 2005.

[12] Cousot, Patrick, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival,"The ASTRÉE analyzer." In European Symposium on Programming, pp. 21-30. Springer Berlin Heidelberg, 2005.

[13] Kirchner, Florent, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowski,"Frama-C: a software analysis perspective", Formal Aspects of Computing 27, No. 3, pp. 573-609, 2015.

[14] Kroening, Daniel, Michael Tautschnig,"CBMC–C bounded model checker", International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2014.

[15] Havelund, Klaus, Thomas Pressburger,"Model checking java programs using java pathfinder." International Journal on Software Tools for Technology Transfer 2.4 (2000), pp. 366-381.
[16] Ball, Thomas, Sriram K. Rajamani,"The SLAM project: debugging system software via static analysis", ACM SIGPLAN Notices. Vol. 37. No. 1. ACM, 2002.
[17] Beyer, Dirk, et al.,"The software model checker Blast." International Journal on Software Tools for Technology Transfer 9.5-6, pp. 505-525, 2007.
[18] Miller, Barton P., Louis Fredriksen, Bryan So. "An empirical study of the reliability of UNIX utilities", Communications of the ACM 33.12 (1990), pp. 32-44.
[19] Godefroid, Patrice, Michael Y. Levin, David A. Molnar. "Automated Whitebox Fuzz Testing." NDSS. Vol. 8. 2008.
[20] Munea, Tewodros Legesse, Hyunwoo Lim, Taeshik Shon,"Network protocol fuzz testing for information systems and applications: a survey and taxonomy." Multimedia Tools and Applications, pp. 1-13, 2015.
[21] Amini, Pedram, and Aaron Portnoy. "Sulley-Pure Python fully automated and unattended fuzzing framework".
[22] Aitel, D."An Introduction to SPIKE." In The fuzzer creation kit, presented at the BlackHat USA Conference. 2002.
[23] Peach, M. Eddington, Peach Fuzzing Platform (peachfuzzer.com)
[24] Bestorm, [Online] Available: http://www.beyondsecurity.com/bestorm.html
[25] Spider Pig: [Online] Available: http://code.google.com/p/spiderpig-pdffuzzer
[26] Song, Dawn, et al.,"BitBlaze: A new approach to computer security via binary analysis", International Conference on Information Systems Security", Springer Berlin Heidelberg, 2008.
[27] Ganesh, Vijay, Tim Leek, Martin Rinard,"Taint-based directed whitebox fuzzing", In Proceedings of the 31st International Conference on Software Engineering, pp. 474-484. IEEE Computer Society, 2009.
[28] King, James C.,"Symbolic execution and program testing", Communications of the ACM 19, No. 7, pp. 385-394, 1976.
[29] Cadar, Cristian, Daniel Dunbar, Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008.
[30] Chipounov, Vitaly, Volodymyr Kuznetsov, George Candea."S2E: A platform for in-vivo multi-path analysis of software systems." ACM SIGPLAN Notices 46.3, pp. 265-278, 2011.
[31] Godefroid, Patrice, Nils Klarlund, Koushik Sen,"DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.
[32] Sen, Koushik, Darko Marinov, Gul Agha,"CUTE: A concolic unit testing engine for C." ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 5. ACM, 2005.
[33] Cha, Sang Kil, Thanassis Avgerinos, Alexandre Rebert, and David Brumley,"Unleashing mayhem on binary code", In 2012 IEEE Symposium on Security and Privacy, pp. 380-394. IEEE, 2012.
[34] Avgerinos, Thanassis, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, David Brumley,"Automatic exploit generation." Communications of the ACM 57, No. 2, pp. 74-84, 2014.
[35] Rawat, Sanjay, Dumitru Ceara, Laurent Mounier, Marie-Laure Potet,"Combining Static and Dynamic Analysis for Vulnerability Detection", arXiv preprint arXiv:1305.3883 (2013).
[36] Chebaro, Omar, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, Boris Yakobowski,"Behind the scenes in SANTE: a combination of static and dynamic analyses." Automated Software Engineering 21, No. 1, pp. 107-143, 2014.
[37] Kiss, Balázs, Nikolai Kosmatov, Dillon Pariente, Armand Puccetti,"Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed." In Haifa Verification Conference, pp. 39-50. Springer International Publishing, 2015.
[38] Godefroid, Patrice, Aditya V. Nori, Sriram K. Rajamani, Sai Deep Tetali,"Compositional may-must program analysis: unleashing the power of alternation", In ACM Sigplan Notices, Vol. 45, No. 1, pp. 43-56. ACM, 2010.
[39] Stephens, Nick, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna,"Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In Proceedings of the Network and Distributed System Security Symposium. 2016.
[40] Evans, David, David Larochelle,"Improving security using extensible lightweight static analysis", IEEE software 19, No. 1, pp. 42-51, 2002.
[41] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler, EXE: Automatically generating inputs of death, ACM Transactions on Information and System Security, Vol. 12(2), pp. 10, pp. 1–38, 2008.