BLENDING FUZZING AND SYMBOLIC EXECUTION FOR MALWARE ANALYSIS

A Dissertation
Presented to
The Academic Faculty

by

Addison Amiri

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in Computer Science in the
School of Computer Science

Georgia Institute of Technology
May 2017

BLENDING FUZZING AND SYMBOLIC EXECUTION FOR

MALWARE ANALYSIS

Approved by:

Dr. Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Simon P. Chung
School of Computer Science
Georgia Institute of Technology

Date Approved: May 5, 2017

ACKNOWLEDGEMENTS

I would like to thank my fiancée for her support and love.

TABLE OF CONTENTS

SUMMARY

Malware infections have grown at least five-fold in the past five years [1]. With an increase in IoT devices that are lacking built-in security, this problem is likely to only continue growing. Malware analysis, meanwhile, is becoming ever more challenging. Where manual analysis, symbolic execution, or fuzzing alone are overly time consuming or unfruitful, a combination of these techniques may offer promising solutions. This paper suggests a combination of fuzzing and symbolic execution to reverse engineer malware. A framework is described to tie these components together, producing test cases that call all functionality of a malware binary. These test cases show researchers the protocol used by the malware, as well as its capabilities, and allow for a reconstruction of the C&C server as desired. The goal of this work is to allow researchers to better understand malware and how to effectively combat it.

CHAPTER 1. INTRODUCTION

Malware—malicious software installed on a user's machine without the user's knowledge—can compromise user security and privacy, result in significant financial loss and "compromise large numbers of machines that are linked together" to form botnets [2]. Last year, nearly "one million new malware threats were released each day" [3], highlighting the importance of doing research in this field.

To effectively combat malware, researchers and security professionals must use reverse engineering to determine how the malware works, what functions it performs, and under what circumstances these functions are executed. This practice is known as malware analysis. "Malware analysis is the process of determining the behavior and purpose of a given malware sample" [2]. Malware analysis becomes continuously more challenging as malware authors anticipate their malware being analyzed and use what is known as binary obfuscation. Binary obfuscation uses a variety of techniques, to make a binary (in this case malware) much harder to analyze. Binary obfuscation is used legitimately by many industries to protect trade secrets such as algorithm optimizations that could be read and copied from the binary, inhibit attackers from analyzing the binary for vulnerabilities, and in the form of DRM, protect against copyright infringement. Tools used to do this kind of obfuscation are freely available online and are becoming common practice among malware distributors.

In response to these obfuscation techniques, researchers have focused their malware analysis efforts on dynamic analysis. Dynamic analysis consists of letting a malware

sample run in an isolated environment while analyzing its execution [4]. Fuzzing and symbolic execution are two techniques that have proven useful for this purpose.

Fuzzing and symbolic execution are two techniques commonly used for reverse engineering. Fuzzing aids this process by bombarding a binary with different inputs as possible and tracking how the program's execution changes. Through trial and error, a fuzzer will correlate how changes in the input effect the execution of the program and using this information try to intelligently drive the program to expose more of its functionality. Symbolic execution instead analyzes the compiled bytecode of the program and finds branches in its execution. For each branch, it finds how the branch is determined and sets the necessary variables to force execution down each path.

Each of these techniques suffer from their own weaknesses when used independently. The computational complexity of fuzzing increases dramatically as the set of potential inputs increases. For instance, programs that communicate over the internet can rely on timing to determine execution. A fuzzer with no knowledge of how timing influences execution will have to try all variations of timings for network packets to find the effect on execution. Symbolic execution, on the other hand, can become near-impossible when a binary has complex branching, exemplified by a technique known as packing.

When used in conjunction with each other, the weaknesses of symbolic execution and fuzzing can be mitigated. This paper suggests that using both fuzzing and symbolic execution together for malware analysis can help address the shortcomings of each

technique, allowing researchers to perform more effective reverse engineering to better analyze, and combat, malware.

CHAPTER 2. LITERATURE REVIEW

2.1 Current Uses of Fuzzing and Symbolic Execution

Research efforts in fuzzing and symbolic execution have primarily focused on their use as tools for identifying security vulnerabilities and bugs [5] [6]. In this case, they will drive a binary through as many execution paths as possible, to test all functions of the program for potential crashes.

Fuzzing does this through creating and permuting sets of inputs for the program. These sets of inputs are called test cases. For identifying vulnerabilities and bugs, a fuzzer's effectiveness is determined by its ability to produce test cases which cause a program to exhibit peculiar behavior, such as hanging or crashing. A researcher will then comb through the test cases of interest and determine the cause of the peculiar behavior and whether the behavior is exploitable.

Symbolic execution, instead, replaces all the inputs of a program with symbolic values. Initially the symbolic values can be anything, but as the program runs, the symbolic executor will find branch conditions which restrict the input values for a certain execution path. For any path, the symbolic values can be evaluated to a concrete test case using a constraint solver.

Fuzzing and symbolic execution can thus be useful techniques for malware analysis. Instead of manually reverse engineering a malware sample, a combination of fuzzing and symbolic execution would allow researchers to focus on the functionality of the sample.

While symbolic execution has already been leveraged for automatic reverse engineering [7] [4] [2], fuzzing is seldom used for this purpose.

2.2 Limitations of Fuzzing and Symbolic Execution

Fuzzing has many limitations, with the largest being the speed at which it can generate and run test cases. In cases where there is not enough information about the binary to produce good test cases, fuzzing "usually provides low code coverage and can miss security bugs". It can also be "ineffective if most generated inputs are rejected at the early stage of program running" [8]. Like brute-forcing an encryption key, fuzzing has a large element of randomness. While information about the input syntax can often be given to the fuzzer to guide its test case generation, in many cases no information about the expected input is known by the researcher. In these cases, the fuzzer is tasked with trying random input until the program takes an execution path not before seen. However, even with this information given to the fuzzer, the randomness of the method often results in less code coverage than other methods.

Symbolic execution, on the other hand, typically provides good code coverage. However, it is expensive in terms of processing power [6] and suffers from a variety of other challenges relating to path explosion, constraint solving, memory modeling and handling concurrency [9]. In addition, binary obfuscation techniques seriously impact symbolic execution times [10].

2.3 Malware Analysis Frameworks

A variety of frameworks for malware analysis have been suggested and developed [11], including frameworks for fuzzing and symbolic execution [12] [13]. However, "fuzzer frameworks typically require a considerable investment of time and resources to model tests for a new interface, and if the framework does not offer ready-made inputs for common structures and elements, efficient testing also requires considerable expertise in designing inputs that are able to trigger faults in the tested interface" [14].

2.4 Existing Efforts That Blend Fuzzing and Symbolic Execution

While symbolic execution systems have "proven to highly improve the effectiveness of traditional fuzzing tools" [8], research blending fuzzing and symbolic execution is not widespread. Whitebox fuzzing, first implemented in SAGE, "consists of symbolically executing the program under test dynamically, gathering constraints on inputs from conditional branches encountered along the execution" [5]. It "extends the scope of systematic dynamic test generation from unit testing to whole-application testing" [6].

TaintScope is one example of "an automatic fuzzing system using dynamic taint analysis and symbolic execution techniques" applied to security vulnerability identification [15]. This symbolic-execution-based fuzzing tool "can symbolically evaluate a trace, reason about all possible values that can execute the trace, and then detect potential vulnerabilities on the trace" [15].

Driller, a tool developed at UC Santa Barbara recently, leverages the strengths of fuzzing and symbolic execution, and mitigates their respective weaknesses. Fuzzing, for example, mitigates the path explosion problem of symbolic execution [16]. Driller was also developed with vulnerability discovery in mind.

These efforts all provide evidence that using both fuzzing and symbolic execution can mitigate the weaknesses of both. However, they were all developed with the goal of finding vulnerabilities. While the process is the same, using them for malware analysis would require modification to the tools as code coverage in malware analysis is less important than potentially identifying a malware's family, source, or target. These can potentially be identified in the early stages of analysis by comparing the execution signature of a binary against previously tested malware.

CHAPTER 3. METHODS

Fuzzing and symbolic execution use different methods to try to force all functionality out of a binary—aiming to get every function of the binary to be executed at least once. Using these techniques together, the fuzzer can first create test cases that a symbolic executor can then use. This paper describes a framework that leverages both fuzzing and symbolic execution together, mitigating the shortcomings of each when used independently, for reversing obfuscated binaries.

3.1 Framework Constraints

While designing the framework, constraints outside of integrating the fuzzer and symbolic executor were identified. Over 120 million new malware samples have been identified each year since 2014 [1]. Due to this rapid growth, it is infeasible to expect researchers to perform a manual analysis of each binary. Thus, one of our primary goals was to automate the majority of the framework.

To leverage the capabilities of fuzzing and symbolic execution together, there must be some degree of awareness regarding their progress and success while analyzing a binary. For example, when a fuzzer reaches a point at which it is no longer successfully increasing code coverage, this must be recognized, and the test cases should be handed to the symbolic executor. When the action of the binary becomes too complex for the symbolic executor, this must also be recognized, and the test cases should be handed back to the fuzzer. The framework must also be able to recognize when neither solution is functioning effectively and stop its current analysis, potentially moving on to the next sample in a queue.

3.2 The Fuzzing and Symbolic Execution Framework

The proposed framework consists of a symbolic executor, a fuzzer, and an orchestrator. The symbolic executor must be able to load and save arbitrary program states. It is not expected to be able to analyze these states itself, but is expected to symbolically execute from that point forward. The symbolic executor must also be able to communicate with the orchestrator, including its progress in terms of code coverage and the parameters that influence execution branches.

The orchestrator must then be able to accept the parameters identified by the symbolic executor and produce effective test cases for the fuzzer. It must also be able to record all relevant program states produced by the symbolic executor and hand these off to the fuzzer, reducing the time needed for each test case's execution. Finally, the orchestrator must monitor the progress of the symbolic executor and as it detects a slowdown in code coverage, signal that the symbolic executor should switch to fuzzing.

The fuzzer can then take the test cases and program states provided by the orchestrator and symbolic executor and start fuzzing. It should monitor its code coverage and report this back to the orchestrator. Should the code coverage rate slow, it will be up to the orchestrator whether to do another round of symbolic execution or whether to move onto another malware sample.

The point of the orchestrator in this case is to provide a mechanism for easy replacement of both the symbolic executor and fuzzer if desired. Depending on the binary, different symbolic executors will perform more or less effectively. If the orchestrator supports them, the most effective symbolic executor can be selected for use in generating

test cases. Likewise, multiple fuzzers could be implemented and run simultaneously, sharing information with the orchestrator and receiving information from the orchestrator, speeding up the fuzzing process. This architecture also allows a single orchestrator to potentially manage the analysis of multiple binaries concurrently, or a single binary across multiple machines.

At the end of the process, ideally with 100% code coverage, the framework could minimize the test cases to ensure there is no duplication. After this process, the test cases should fully cover the functionality of the malware binary and could be used to construct a functional command and control server for simulation.

3.3 Proposed Implementation

As there are many pieces to the framework, the work will have to be split up into steps, with each step implementing a part of the framework, eventually building up to a fully functional framework.

1. Create a basic orchestrator with dummy symbolic executors and dummy fuzzers to facilitate communication between the two.

2. Integrate a symbolic executor with the orchestrator in a meaningful way.

3. Integrate a fuzzer with the orchestrator and patch its test case format.

4. Implement test case generation in the orchestrator, including test case minimization.

5. Implement work control to determine when an analysis job should be terminated.

After these steps, a fully functional framework implementation will be finished, and further improvements to the orchestrator, or further support in different symbolic executors and fuzzers, can be added.

CHAPTER 4. DISCUSSION

Before a framework like this can be implemented, work needs to be done on both fuzzers and symbolic executors to make them compatible with the concept of an orchestrator.

The first thing that needs to be achieved is standardizing an API that allows interfacing with fuzzers and symbolic executors. The orchestrator must be able to send instructions and receive results from both the fuzzer and symbolic executor to function correctly. The APIs must also be compatible enough that the output from one can easily be passed to the other for further work. Standardization would also allow for fuzzers and symbolic executors to be used interchangeably, allowing for greater flexibility of the tools being used. For instance, an unpacker which supports this API could coordinate with the orchestrator, aiding significantly in symbolic execution efforts. This interchangeability would also allow orchestrators to move from specialized fuzzers and symbolic executors to more generalized solutions when a binary isn't handled well by a specialized solution.

Developing a method for saving the execution state of a binary and sharing it with other analysis tools would also help the orchestrator significantly. If a fuzzer and a symbolic executor could somehow share program states, it would allow each to easily build on the progress of the other without having to reestablish the program state each time. This solves a problem typically faced by malware analysts, as malware binaries commonly receive instructions from a command and control server. During analysis, the server may no longer exist or may not be accessible from the sandboxed environment running the malware.

With these two problems addressed, the implementation of the framework described in this paper becomes much easier and a proof of concept could likely be implemented in a much more reasonable amount of time. This proof of concept could be the starting point for a range of tools aiding malware analysts in their efforts. With this framework, malware samples could be collected, analyzed, and categorized before a human analyst is needed. By the time an analyst is needed for a malware sample, the malware's toolkit and command and control servers may already be known. The users targeted by the malware, as well as the malware's behaviors in different scenarios can also be recognized prior to human intervention. With such a solution, malware analysts would be better able to leverage their time and efforts. Malware analysts would be able to confirm the findings of the framework, and analyze malware too advanced for the framework to process, as the majority of common malware variants should be recognized by the framework after basic processing.

CHAPTER 5. CONCLUSION

Based on the rate at which malware is increasing, it is clear that we must investigate new ways of analyzing and combating its spread. This paper proposes a framework that would use fuzzing and symbolic execution in conjunction to reverse engineer malware samples. Such a framework could significantly improve analysis of malware and aid in attribution of malicious attacks.

REFERENCES

[1] AV-TEST, "Malware Statistics & Trends Report," AV-TEST, 20 03 2017. [Online]. Available: https://www.av-test.org/en/statistics/malware/. [Accessed 03 04 2017].

[2] A. Moser, C. Kruegel and E. Kirda, "Exploring multiple execution paths for malware analysis," in *2007 IEEE Symposium on Security and Privacy*, Oakland, 2007.

[3] V. Harrison and J. Pagliery, "Nearly 1 million new malware threats released every day," CNN, 14 04 2015. [Online]. Available: http://money.cnn.com/2015/04/14/technology/security/cyber-attack-hacks-security/. [Accessed 10 04 2017].

[4] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, Springer, 2008, pp. 65-88.

[5] P. Godefroid, M. Y. Levin and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM,* vol. 55, no. 03, p. 40, 01 03 2012.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, K. Tillmann and W. Visser, "Symbolic execution for software testing in practice: preliminary

assessment," in *33rd international Conference on Software Engineering*, Honolulu, 2011.

[7] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song and H. Yin, "Bitscope: Automatically dissecting malicious binaries," *School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-07-133,* 18 March 2007.

[8] T. Wang, T. Wei, G. Gu and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, Oakland, 2010.

[9] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM,* vol. 56, no. 2, pp. 82-90, 02 2013.

[10] M. I. Sharif, A. Lanzi, J. T. Giffin and W. Lee, "Impending Malware Analysis Using Conditional Code Obfuscation," in *Network and Distributed System Security Symposium*, San Diego, 2008.

[11] M. Egele, T. Scholte, E. Kirda and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys,* vol. 44, no. 2, pp. 1-42, 01 02 2012.

[12] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*, Oakland, 2010.

[13] S. Khurshid, C. S. Păsăreanu and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, 2003.

[14] A. Takanen, J. D. Demott and C. Miller, Fuzzing for software security testing and quality assurance, Norwood, MA: Artech House, 2008.

[15] T. Wang, T. Wei, G. Gu and W. Zou, "Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution," *ACM Transactions on Information and System Security,* vol. 14, no. 2, pp. 1-28, 01 09 2011.

[16] N. Stephens, J. Grosen, C. Salls, A. dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, K. Christopher and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network and Distributed System Security Symposium*, San Diego, 2016.