2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

MARVEL: A Generic, Scalable and Effective Vulnerability Detection Platform

Xiaoning Du
Nanyang Technological University, Singapore

***Abstract*—Identifying vulnerabilities in real-world applications is challenging. Currently, static analysis tools are concerned with false positives; runtime detection tools are free of false positives but inefficient to achieve a full spectrum examination. In this work, we propose MARVEL, a generic, scalable and effective vulnerability detection platform. Firstly, a lightweight static tool, LEOPARD, is designed and implemented to identify potential vulnerable functions through program metrics. LEOPARD uses complexity metrics to group functions into a set of bins and then ranks functions in each bin with vulnerability metrics. Top functions in each bin are identified as potentially vulnerable. Secondly, a directed grey-box fuzzer is designed to take the results from LEOPARD for further confirmation. Our design stands out with the ability to automatically group adjacent functions and orchestrate both the macro level function directed fuzzing and the micro level path-condition directed fuzzing. LEOPARD is evaluated to cover 74.0% of vulnerable function when identifying 20% of functions as vulnerable and outperforms the baseline approaches. Further, three applications are proposed to demonstrate the usefulness of LEOPARD. As a result, we discovered 22 new bugs and eight of them are new vulnerabilities.**

I. INTRODUCTION

Vulnerabilities are one of the major threats to software security. Hunting vulnerabilities in real-world applications is challenging, as the applications acn accept innumerable inputs from a specific domain and are implemented with complex code in a large scale. Since only a few vulnerabilities are scattered across a large amount of code, vulnerability hunting is comparable to finding "a needle in a haystack" [19]. Existing dynamic and static detection tools either explore the input space (e.g., black-box fuzzing [14] and grey-box fuzzing [1], [13], [4], [11]) or analyze the internal program state space (e.g., symbolic execution [9], [2] and metrics/pattern-based analysis [12], [17], [10], [16], [5], [15]), hence both require optimized search strategies to identify hidden vulnerabilities.

Static analysis tools try to explore the internal program states by analyzing the source code or binary code. Most static scanners work to flag potential vulnerable codes or functions and can hardly provide concrete triggering inputs to validate the feasibility of the vulnerability unless symbolic execution is integrated. Unfortunately, scalability of symbolic execution is extremely restrained by the path exploration problem and the capability of SMT solvers. Inevitably, vulnerabilities reported by static scanners could have high false positives. Currently, input-oriented fuzzing techniques gain great popularity with the ability to produce confirmed vulnerabilities with triggering inputs. Different from the purely blind black-box fuzzing, grey-box fuzzing take as feedback the coverage on internal states to improve the testing efficiency. However, it still fails to generate inputs to examine the full spectrum of states in the program, especially states buried deeply. Instead, researchers design fuzzers (AFLGo [3] and Hawkeye [7]) that can be directed to fuzz designated parts of the program, assuming they are aware of targets that are most likely vulnerable. The merit behind directed grey-box fuzzing (DGF) relies on narrowing down the search space and putting more time and efforts to where it is needed most.

In order to take the advantages of both static and dynamic techniques, we propose to integrate static vulnerability identification with DGF to achieve a *generic*, *scalable*, and *effective* vulnerability detection platform, named MARVEL. For the static analysis phase, we propose a lightweight approach, LEOPARD, to identifying potential vulnerable functions through program metrics analysis. We first use complexity metrics to group functions into a set of bins. Then, vulnerability metrics are developed to rank the functions in each bin and the top ones are recognized as potentially vulnerable. For the DGF phase, we present a design which aims to overcome the problem with current directed fuzzers [3], [7] of supportting only a few targets in a particular fuzzing task. Besides, we plan to enhance DGF with some fine-grained path condition penetration techniques, like local taint analysis or local symbolic execution, in order to improve the efficiency while keep good scalability. LEOPARD is evaluated on 11 real-world projects and demonstrated to achieve an average 74.0% recall when identifying 20% of functions as potentially vulnerable, which outperforms the state-of-the-art baseline approach, machine learning(ML)-based approaches and pattern-based scanners. We further validate the usefulness of LEOPARD in manual code review and fuzzing, through which we discovered 22 new bugs in real applications like `PHP`, `radare2` and `FFmpeg`, and eight of them are new vulnerabilities. More details can be found in [8].

II. OUR APPROACH

An overview of the MARVEL framework is shown in Fig. 1. Given the source code of a C/C++ application, the vulnerable function identification component, LEOPARD, works to generate a list of potential vulnerable functions through a program metrics based binning-and-ranking method. These functions are then fed to the DGF component as targets to further examine whether they can be confirmed vulnerable with concrete triggering inputs. Adjacent targets are grouped to fuzz in a parallel and efficient way. Function directed and path-

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00056

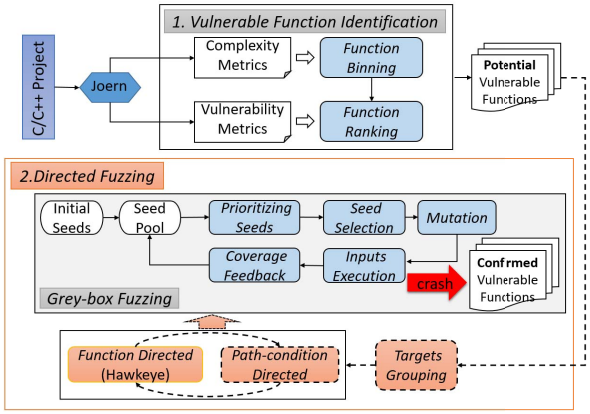

Fig. 1. An Overview of the MARVEL Framework

condition directed strategies interweaves to reach the target sites quickly. Note that completed functional modules in the framework are drawn with solid lines, and ongoing ones are marked with dashed lines.

A. Potential Vulnerable Function Identification

Binning. Four program complexity metrics are devised to characterize the structural complexity of functions. Intuitively, they are used to approximate the number paths in a function by combining cyclomatic complexity and loop-related metrics. We compute a complexity score for each function by adding up the values of these complexity metrics and group functions with the same score into the same bin.

Ranking. Memory management errors and missing checks on sensitive variables forms a majority of critical types of vulnerabilities in C/C++ programs. Based on this observation, eleven vulnerability metrics are developed to capture the characteristics of general causes of vulnerabilities, which are grouped in three dimensions: 1) two dependency metrics to characterize the dependency relationship of a function with other functions; 2) three pointer metrics to capture the manipulation of pointers, e.g., the number of variables used in pointer arithmetics; and 3) six control structure metrics to capture the vulnerability resulting from the highly coupled and dependent control structures, e.g., the maximum nesting level of control structures and the number of *if* structures without *else* statement. Basically, larger values of vulnerability metrics indicate a function is involved in complicated computations both internally and externally, and is potentially with memory management problems, resulting the function hard to follow and analyze, thus most likely to contain vulnerabilities.

The binning-and-ranking strategy incorporates both complexity metrics and vulnerability metrics to identify vulnerabilities at all levels of complexity without missing low-complexity ones. Also it requires no prior knowledge about known vulnerabilities compared to ML-based approaches.

B. Directed Grey-box Fuzzing

The vanilla grey-box fuzzing initially takes a set of seeds, and loops over to prioritizes the seeds, selects one to do the mutations, executes the newly generated inputs, checks their coverage and keeps good ones to supplement the seed pool.

We design DGF based on Hawkeye, the state-of-the-art DGF tool, and with two main enhancements. Firstly, the target functions are grouped based on their distance on the call graph. Adjacent targets are fuzzed within the same fuzzing task, thus more targets (than Hawkeye) can be handled in a parallel scheme to save computing resources. Secondly, we desire a combination of both macro and micro guidance to extensively felicitate the directed fuzzing process. Hawkeye provides mainly the macro guidance on exploring a feasible call chain leading to the target function, and we call it function directed fuzzing. However, it is lack of micro guidance to break through some complex path conditions more efficiently, and technically we call it path-condition directed fuzzing. Function directed fuzzing can easily get stuck at some complex and tricky path conditions, especially when the searching space is restricted by the target sites and these hard conditions becomes the only way to make progress. To address this, we suggest to combine the function directed fuzzing with the path-condition directed fuzzing, also with an optimized interplay between the two techniques. We plan to design a runtime scheduler to orchestrate the two techniques dynamically.

III. EVALUATION

We implement LEOPARD in 11k line of Python code and use Joern [16] to extract the program metrics.

Effectiveness. We evaluate the effectiveness of LEOPARD in identifying vulnerable functions on 11 real-world applications, including `Binutils`, `FFmpeg`, `SQLite`, `Wireshark` and `Linux Kernel`. The benchmarks are from various application domains and include large-scale ones. On average, when identifying 15%, 20%, 25% of functions as vulnerable, we achieve recalls of 64%, 74%, and 78%. Compared to the baseline approach [18], 9.2%, 10.3%, and 7.4% improvement is achieved. LEOPARD also outperforms ML-based approaches, an open-source static scanner and a commercial static scanner by a substantial margin.

Usefulness. To further evaluate the usefulness of LEOPARD, three applications are designed to intake its results for vulnerability assessment, including manual auditing, directed fuzzing and seed prioritization. For manual auditing, in order not to overwhelm the security expert, top 1% of vulnerable functions from LEOPARD are presented to him with detailed values for each metrics. For the directed fuzzing, we manually select dozens of functions reported by LEOPARD and group them based on the architectural modules. These targets are fed to Hawkeye for bug hunting. For the seed prioritization, we score all the functions based on its ranking given by LEOPARD and guide grey-box fuzzing to favor seeds that can exercise more vulnerable paths. Paths are scored via accumulating the scores of functions on that path. A prototype has been built upon FOT [6], an extensible fuzzing framework. Case studies are conducted on `FFmpeg`, `PHP`, and `radare2`, with which we discovered 22 new bugs and eight of them are new vulnerabilities. Remarkably, six vulnerabilities are unveiled with directed fuzzing, which also inspires us to investigate a more efficient and fully automated DFG design.

REFERENCES

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017.

[2] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22, 2011.

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *CCS*, pages 2329–2344, 2017.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *CCS*, pages 1032–1043, 2016.

[5] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *FSE*, pages 678–689, 2016.

[6] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *ESEC/FSE*, pages 867–870, 2018.

[7] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS*, pages 2095–2108, 2018.

[8] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. *arXiv preprint arXiv:1901.11479*, 2019.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[10] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27(C):504–518, 2015.

[11] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[12] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[13] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *SP*, pages 497–512, 2010.

[14] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *CCS*, pages 511–522, 2013.

[15] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *ICSE*, pages 462–472, 2017.

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *SP*, pages 590–604, 2014.

[17] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li. Combining software metrics and text features for vulnerable file prediction. In *ICECCS*, pages 40–49, 2015.

[18] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology*, 27(1):1, 2018.

[19] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST*, pages 421–428, 2010.