Boosting Symbolic Execution for Heap-based Vulnerability Detection and Exploit Generation

Haoxin Tu
Singapore Management University, Singapore
haoxintu.2020@phdcs.smu.edu.sg

***Abstract*—Heap-based vulnerabilities such as *buffer overflow* and *use after free* are severe flaws in various software systems. Detecting heap-based vulnerabilities and demonstrating their severity via generating exploits for them are of critical importance. Existing symbolic execution-based approaches have shown their potential in the above tasks. However, they still have some fundamental limitations in path exploration, memory modeling, and environment modeling, which significantly impede existing symbolic execution engines from efficiently and effectively detecting and exploiting heap-based vulnerabilities. The objective of this thesis is to design and implement a boosted symbolic execution engine named HEAPX to facilitate the automatic detection and exploitation of heap-based vulnerabilities. Specifically, a new path exploration strategy, a new memory model, and a new environment modeling solution are expected to be designed in HEAPX, so that the new boosted symbolic execution engine can detect heap-based vulnerabilities and generate working exploits for them more efficiently and effectively.**

***Index Terms*—Software security, software reliability, vulnerability detection, automatic exploit generation, symbolic execution**

I. Motivation and Problem Statements

Heap-based vulnerabilities such as *buffer overflow* and *use after free* are severe flaws in software systems [1], [2]. To alleviate security risks induced by heap-based vulnerabilities, a plurality of advanced approaches have been devoted to automatically detecting them [3]–[5] and exploiting them by generating exploits for the vulnerabilities to decide which should be fixed first [6]–[8]. Among the existing approaches, symbolic execution-based ones have shown their potential in vulnerability detection and exploitation benefiting from the versatile capabilities of automatic path exploration and test case generation in symbolic execution [9]–[11].

Despite the success of symbolic execution applications, existing symbolic execution engines still suffer from fundamental limitations in (1) handling the state explosion problem during path exploration, (2) modeling complex memory objects, and (3) modeling run-time environment with high fidelity [12], [13]. These limitations significantly impede existing symbolic execution engines from efficiently and effectively detecting and exploiting heap-based vulnerabilities.

To overcome the above limitations, this thesis aims to investigate the following three main research questions (***RQs***):

RQ1. *How to effectively guide path exploration toward detecting more heap-based vulnerabilities?*

Previous studies target various coverage-based heuristics to alleviate the path explosion challenge [9], [14], [15]. However, they are not vulnerability-oriented. Since pointer operations have a high risk to induce vulnerabilities, a new path search strategy with the help of pointer checking is needed for effective path exploration toward vulnerability detection.

RQ2. *How to effectively represent memory objects toward detecting more heap-based vulnerabilities?*

Existing modeling of memory addresses are usually represented as concrete values [9], [16]–[18]. However, such modeling is not effective enough as the existing modeling of heap allocation is static (i.e., *concrete*) but the heap-based addresses in actual execution should be dynamic (i.e., *symbolic*). Therefore, a new memory model that has more precise modeling of heap-based memory objects is needed.

RQ3. *How to effectively model the run-time environment of heap allocation towards automatic exploit generation?*

Generating working exploits usually requires run-time information such as run-time addresses to demonstrate the validity of the exploits [6]–[8]. However, existing symbolic execution engines do not consider such an important factor and do not support either native stack or heap addresses for successfully launching and validating generated exploits.

II. Proposed Solutions and Expected Contributions

This thesis aims to design and implement a boosted symbolic execution engine named HEAPX to answer the above questions. Specifically, three new proposed solutions (***SOLs***) to be designed in HEAPX are described as follows.

SOL1. The observation is that, instead of a random or coverage-guided search, a new search towards the more likely vulnerable paths can be effective for vulnerability detection. To do so, a type inference system is combined with symbolic execution to classify the type (*safe* or *unsafe*) of pointer operations before symbolic execution. Here, the *unsafe* means the path that is more likely to be vulnerable. Then, an *unsafe-pointer-oriented* search strategy can be applied to guide the path exploration during symbolic execution, which means execution paths with more *unsafe* pointer operations will be the top priority and be explored first.

SOL2. The intuition is that, instead of modeling memory addresses with concrete ones, a new memory model of symbolic memory addresses with efficient symbolic memory read/write operations can be more helpful for vulnerability detection. Therefore, a new memory model which not only models memory addresses using symbolic values rather than

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00059

concrete ones but also supports efficient symbolic memory read/write is proposed in HEAPX. Specifically, a new solution of concretely mapped symbolic memory locations is designed to support intractable symbolic memory read/write and thus help detect more heap-based vulnerabilities.

SOL3. The key insight is that, rather than using simulated memory addresses from heap allocation, run-time information such as heap native addresses can be useful for the validation process of exploit generation. So, a new solution that supports native information of run-time environment is leveraged in HEAPX. Specifically, dynamic analysis infrastructure such as OASIS [19] which supports user-space program runs in native addresses is planned to be integrated into HEAPX to replace the simulated heap addresses in existing symbolic execution engines with the new native heap addresses.

This thesis is expected to bring helpful insights and optimizations for symbolic execution to facilitate automatic vulnerability detection and exploit generation. The expected contributions are summarized as follows:

- A new boosted symbolic execution engine HEAPX to facilitate the automatic detection and exploitation of heap-based vulnerabilities. Developers can take HEAPX as an automatic tool to analyze their projects for further improving the reliability and security of the projects.
- Three new techniques are designed in HEAPX to overcome the limitations in path exploration, memory modeling, and environment modeling in symbolic execution, respectively.

III. RELATED WORK

A. Automatic Vulnerability Detection

Existing studies in this field can be broadly classified into three categories: static, dynamic, and symbolic execution-based approaches. Coccinelle [20], Cppcheck [21], TscanCode [22], and Frama-C [23] are well-known static-based approaches that rely on certain existing vulnerability patterns to detect heap-based vulnerabilities. Dynamic approaches either apply dynamic binary translation to collect required information via instrumenting test program at run-time (e.g., Valgrind [24], Purify [25], and Dr. Memory [26]) or leverage compile-time instrumentation which adds memory safety checks while the program compiles (e.g., Mudflap [27] and AddressSanitizer [28]) to detect potential vulnerabilities. For the symbolic execution-based ones, KLEE [9] is the leading symbolic execution engine that can detect many heap-based vulnerabilities by using a concrete memory model and certain heuristic path exploration strategies. Then, new search strategies [14], [15] are designed to alleviate the path explosion challenge, and new memory models [17], [18], [29], [30] are also proposed to facilitate the detection of heap-based vulnerabilities. Other approaches adopt hybrid analysis that combines dynamic and symbolic execution to detect vulnerabilities [3], [4].

Unlike existing approaches, we aim to detect more heap-based vulnerabilities based on a new path exploration strategy and memory model: we propose an *unsafe-pointer-oriented* path exploration to explore the paths that are more likely to have vulnerabilities and a new memory model with concretely mapped symbolic locations to improve the vulnerability detection capability of existing symbolic execution engines.

B. Automatic Exploit Generation

Early work on AEG focused on the exploitation of stack-based buffer overflows in user space programs. Avgerinos et al. proposed two symbolic execution-based systems (AEG [6] and Mayhem [7]) that both search for stack-based buffer overflow and generate exploits for them. Repel et al. [31] demonstrated the first approach to AEG for heap overflows. Revery [8] uses a mix of fuzzing and symbolic execution to build exploits. Originating from discussions from Revery, Gollum [11] adopts the idea of "one-gadget" payloads to exploit the corruption of data used by the application itself. KOOBE [10] studies the exploit generation of kernel out-of-bounds write vulnerabilities.

Different from existing approaches, we focus on supporting heap native environments in symbolic execution to facilitate the exploitation of heap-based vulnerabilities. Specially, we support native heap address space in HEAPX rather than simulated one to facilitate the validation of generated exploits.

IV. EVALUATION CRITERIA AND PRELIMINARY RESULTS

Evaluation Criteria. To evaluate the effectiveness of the three new proposed solutions, we plan to separately test them. For *SOL1* and *SOL2*, we plan to compare HEAPX with state-of-the-art approaches in terms of code coverage (i.e., can HEAPX cover more code?) and vulnerability detection capability (i.e., can HEAPX detect more vulnerabilities?), over widely-used benchmarks included heap-based vulnerabilities such as GNU Coreutils [32]. For *SOL3*, we plan to use the exploitability (i.e., can HEAPX generate working exploits?) to measure the effectiveness of HEAPX over well-known CGC benchmarks [33] or other important software such compilers.

Preliminary Results. We have two preliminary results on the first two solutions. First, we proposed FASTKLEE [34] and have shown the advantage of combing a type inference system with symbolic execution in terms of performance, achieving up to 9.1% (5.6% on average) speedups compared with state-of-the-art approach KLEE. In the near future, we plan to extend FASTKLEE and apply the *unsafe-pointer-oriented* path exploration strategy in our future work. Second, we designed a new memory model and leveraged concretely mapped symbolic memory locations to alleviate the complex memory modeling challenge. The preliminary evaluation results show that our approach can detect 8%-64% more heap-based vulnerabilities than existing memory detectors.

Timeline Plans. This year before August, we plan to continue the investigation of RQ1 and finalize the work on RQ2. For the rest of this year and the first six months of the next year, we are going to investigate RQ3: we may first empirically study the existing exploits patterns and learn what are the common requirements (e.g., the support of native addresses or the use of specific exploitable patterns) to launch the working exploits. Then, based on the empirical results, we plan to design our new strategies for automatically generating working exploits for heap-based vulnerabilities. After that, I will start to work on preparing the Ph.D. thesis, which is expected to be defended by December 2024.

Acknowledgment

I appreciate my supervisors very much for their continuous support and patience as well as the anonymous reviewers for their insightful comments. This article is partially supported by the National Research Foundation (NRF) Singapore and the National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) award number NSOE-TSS2019-04.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 48–62.

[2] R. M. farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 1037–1054.

[3] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 1–16.

[4] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 745–761.

[5] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards bug-driven hybrid testing," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1580–1596, 2019.

[6] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy (SP)*, 2012, pp. 380–394.

[8] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1914–1927.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[10] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities," in *29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 1093–1110.

[11] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1689–1706.

[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, 2018.

[13] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[14] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *SIGPLAN Not.*, vol. 48, no. 10, pp. 19–32, 2013.

[15] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 2526–2540.

[16] L. Borzacchiello, E. Coppa, D. Cono D'Elia, and C. Demetrescu, "Memory models in symbolic execution: key ideas and new thoughts," *Software Testing, Verification and Reliability*, vol. 29, no. 1–8, p. e1722, 2019, e1722 stvr.1722.

[17] D. Trabish, S. Itzhaky, and N. Rinetzky, "A bounded symbolic-size model for symbolic execution," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 1190–1201.

[18] D. Trabish and N. Rinetzky, "Relocatable addressing model for symbolic execution," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 51–62.

[19] J. Hong and X. Ding, "A novel dynamic analysis infrastructure to instrument untrusted execution flow across user-kernel spaces," in *IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 402–418.

[20] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, "Coccinelle: tool support for automated cert c secure coding standard certification," *Science of Computer Programming*, vol. 91, pp. 141–160, 2014.

[21] D. Marjamäki, "Cppcheck: a tool for static C/C++ code analysis," 2023.

[22] Tencent, "A fast and accurate static analysis solution for C/C++, C#, Lua codes," 2023. [Online]. Available: https://github.com/Tencent/TscanCode

[23] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, 2012, pp. 233–247.

[24] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.

[25] R. Hastings, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of 1992 Winter USENIX Conference*, 1992, pp. 125–136.

[26] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of IEEE Symposium on Code Generation and Optimization (CGO)*, 2011, pp. 213–223.

[27] F. C. Eigler, "Mudflap: Pointer use checking for c/c+," *Proceedings of the First Annual GCC Developers' Summit*, pp. 57–70, 2003.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 1–28.

[29] M. Nowack, "Fine-grain memory object representation in symbolic execution," in *Proceedings of the 34th ACM/IEEE Automated Software Engineering (ASE)*, 2019, pp. 912–923.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[31] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, p. 25–35.

[32] G. Coreutils, "A collection of core utilities which are expected to exist on every operating system," 2022. [Online]. Available: https://www.gnu.org/software/coreutils/

[33] Cb-multios, "DARPA Challenges Sets for Linux, Windows, and macOS," 2022. [Online]. Available: https://github.com/trailofbits/cb-multios

[34] H. Tu, L. Jiang, X. Ding, and H. Jiang, "FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1741–1745.