RESEARCH STATEMENT

Hang Zhang (hzhang420@gatech.edu)

1 Motivation and Overview

My research focuses on security analysis and vulnerability discovery in complex yet essential software systems (*e.g.,* Operating Systems). Due to the modern world's heavy dependence on software, timely pinpointing and understanding their security flaws is critical. However, it remains a significant challenge to automate this process, especially for intricate tasks such as discovering subtle vulnerabilities in the large code base.

I set a critical goal for my research to systematically and automatically reason about the software for security loopholes. Following this direction, I have discovered and investigated real-world security issues [12, 13, 1], developed techniques to automatically and accurately pinpoint known-but-unpatched vulnerabilities [11, 14], and unknown complex vulnerabilities of certain types [10]. My past work span different software layers (*e.g.,* source code, IR, and binary), explore diverse techniques (*e.g.,* both static and dynamic program analysis), and cover various vulnerability categories (*e.g.,* both known and unknown, including multiple types such as memory corruption, denial-of-service, information leak, *etc.*). My research has led to multiple papers published on top venues in related areas (*e.g.,* ACM CCS, USENIX Security, ACM ICSE), open-source tools attracting interest from and applied in both academia and industry, and various discovered vulnerabilities fixed, acknowledged, and sometimes bountied by the community.

2 Past and Current Work

My journey of security research started with a personal hobby of reverse engineering and hacking (especially PC game hacking), which led my path to bug hunting and security analysis of large software systems (§2.1). During this process, I found that regardless of the availability of techniques and tools for automatic bug finding, many in-depth and systematic security analyses are still manually performed by experienced human experts, as well as the identification of subtle security issues (*e.g.,* my own work in §2.1). I then began to ask myself whether, how, and to what degree the machine could be programmed to accomplish the same tasks. These questions have been motivating my subsequent research, more specifically, I have then developed the techniques to identify the yet-to-patch vulnerabilities at the binary level, as accurately and flexibly as human experts but fully automated (§2.2), and to discover particular unknown vulnerabilities with complicated control/data flows, for which existing tools have difficulty recognizing (§2.3). More recently, I have worked in this same direction to support discovering a broader range of complex vulnerabilities. In the remaining section, I detail three significant lines of my research work.

2.1 Security Analysis of Software Systems

In this line of work, I conduct security assessments of the Android ecosystem and pinpoint some severe security flaws, including:

(1) Study of Android One-Click Root Ecosystem. Android one-click root ecosystem comprises end users who want to obtain the root privilege of their mobile devices (*e.g.,* for better system customization) and the providers who provide convenient one-click root services. In this work, we conduct a first-of-its-kind comprehensive study of this ecosystem with a thorough reverse engineering of many major root service providers [12]. We surprisingly find that these services are backed by a sophisticated set of dangerous but poorly protected root exploits, which can be (and have been) easily abused by malicious actors. We have reported our findings to the community and got acknowledged, as well as the media coverage [2]. Moreover, the large set of root exploits we extract during this research is utilized in follow-up work to detect in-the-wild exploitation attempts [4], providing extra safeguards for Android device users.

(2) Security Analysis of Android ION Subsystem. In this project, we perform an in-depth security analysis of Android ION memory management subsystem [13], which reveals multiple high-profile security loopholes that can lead to severe consequences like sensitive user information leakage and denial-of-service attacks. We further develop a novel static analysis method to help developers pinpoint these issues early in the development cycle. We responsibly report our findings to the vendors, which have been assigned multiple CVEs and bountied by Google.

I have also participated in other work along the same line, including a discrepancy analysis between the app-side and the web-side security mechanisms [1], and a large-scale study of the app promotion fraud [3].

2.2 Automatic Identification of Unpatched Vulnerabilities

After manually identifying security issues in my previous work, I start to consider how to catch them automatically. What first comes to my mind is the known but unfixed software vulnerabilities (*i.e., N-Day vulnerabilities*) due to the delay in the patching process. In this line of work, we focus on the problem of how to identify these unfixed vulnerabilities automatically and precisely at the binary level, since it is often the case that the source code may be unavailable for downstream software distributions, regardless of their open-source upstream. Inspired by the workflow of human experts, I developed a novel static analysis tool named FIBER [11] that can recognize the most significant code changes (*e.g.,* apparent control flow alternation, extra security checks, *etc.*) introduced by a security patch and then generate the corresponding binary level signatures, which can then be used to detect the unfixed vulnerabilities in the target software distribution. FIBER achieves high accuracy and flexibility, attracting interest from both academia and industry. For example, it has been adopted by car manufacturers to detect unfixed vulnerabilities in Android systems hosted by automobiles.

We further improve FIBER and apply it in a large-scale measurement study of the Android patch ecosystem [14]. Thanks to FIBER's capability of working at the binary level, our study gains a much broader scope than previous work. It results in many insightful findings and suggestions to help reduce the delays in the patch propagation process.

2.3 Automatic Discovery of Unknown Complex Vulnerabilities

Besides the known vulnerabilities, I have also been undertaking research to automatically and principally discover unknown vulnerabilities, which may need complex control/data flows to trigger and thus are primarily occasionally identified (*e.g.,* by random testing or manual inspection). One type of such vulnerability is the cross-entry taint style vulnerability, where the taint flow from (malicious) user input to the sensitive sink crosses multiple entry function (*e.g.,* system calls of the Linux kernel) invocations, relayed by global states. To effectively discover such vulnerabilities, I develop SUTURE [10] a high-precision static analyzer capable of constructing high-order cross-entry data flows, by efficient and on-demand query of individual function summaries. SUTURE has successfully discovered multiple cross-entry taint vulnerabilities in the Linux kernel, which can cause severe security consequences. Besides SUTURE, I also take part in projects detecting other vulnerability types, such as use-before-initialization bugs in the kernel [9].

Currently, I have been working on devising new techniques based on SUTURE to discover the complex cross-entry use-after-free vulnerabilities, where the use and free sites can be reached in separate entry functions and are connected by complicated alias relationships. To discover such issues, I enhance SUTURE with a precise cross-entry alias analysis and a systematic false positive filtering framework taking into account multiple aspects (*e.g.,* locks, sanity checks, *etc.*). Our preliminary evaluation has identified some complex UAF cases in the Linux kernel, showing the tool's potential.

Other Relevant Work. Outside the above significant lines of research work, I am involved in projects where we apply the static analysis techniques developed in my primary research to help perform various security analysis tasks, often in combination with different approaches (*e.g.,* dynamic analysis). For example, to help understand and address the dependency challenge commonly faced by fuzzers (*e.g.,* the satisfaction of a condition check relies on some global states that can be modified in different entry functions) [5], to reveal more profound security consequences of vulnerabilities discovered with fuzzing (*e.g.,* a seemingly less severe over-read vulnerability may cause more dangerous over-write after the crash site where the fuzzing process stops) [15], and to identify the propagation of sensitive cryptography information (*e.g.,* the keys) for its further protection with hardware security mechanisms [6].

3 Future Research Directions

There are still many open problems in my research area, I plan to continue my research lines and pursue the following interconnected directions in the future.

3.1 Principal Discovery of a Broader Range of Complex Vulnerabilities

Many complex vulnerabilities are discovered by chance (*e.g.,* manual inspection and random testing) instead of a principal method like static analysis, which can provide a more systematic scan of the target software, resulting in a more robust security guarantee. The question is why existing static analyzers fall short in finding these complex vulnerabilities and what we can do to improve them, which is the research direction of my interest. In my previous work, I have developed effective static analysis techniques to automatically discover complex vulnerabilities of certain types (§2.3) in the Linux kernel, in the future, I plan to extend my scope to more vulnerability types (*e.g.,* the stealthy side channels and subtle logical errors), programming languages (*e.g.,* Rust), and target software (*e.g.,* the browsers).

3.2 Improve the Warning Review Process of Static Analyzers

During my practice of static bug discovery [10, 9], The arduous and time-consuming warning review process is one major problem preventing static analyzers from better adoption. On the one hand, static bug discovery approaches inherently suffer from false alarms - the reviewers may need to inspect and reject many similar false warnings repeatedly. On the other hand, even for warnings eventually proved to be valid, the reviewers often still need to make lots of efforts to confirm them (*e.g.,* ensure that there are not any overlooked sanity checks by the analyzer), especially for vulnerabilities with complex and lengthy control/data flows. Such difficulties motivate me to think about ways to ease the review process and make static bug discovery more practical and valuable. One concrete idea is to develop flexible and efficient warning grouping and ranking systems (with program analysis or even machine learning-based approaches), so that the reviewer can quickly recognize and exclude similar false alarms as the already identified ones, by matching their control/data flow characteristics. Analogously, for a valid confirmed warning, other similar ones can also be identified and promoted so that actual bugs can be found more quickly. Moreover, I would like to devise methods to explicitly express the uncertainty of the static analysis tools (*e.g.,* over-estimation of a variable value, skipped functions, *etc.*) and precisely assess their impact on the generated warnings (*e.g.,* which variable or code snippets need to be manually checked to confirm the bug), this way, the reviewer will have a better-informed review experience regarding the analyzer's limitations, enabling a more targeted and efficient warning confirmation.

3.3 Automatic Analysis of Bugs Reported by Dynamic Testing

One of the most common complaints among the developers and maintainers for the bug reports generated by dynamic testing (*e.g.,* fuzzing) is the lack of root cause analysis, aside from the superficial symptoms (*e.g.,* crash at a specific location) and simple traces (*e.g.,* the call stack), making it difficult to reproduce the issue, assess the severity, and develop fixes [7, 8].

In past research, we have developed methods to automatically assess the severity of certain types (*e.g.,* buffer over-read) of fuzzer-reported security issues [15], which is an initial step in this research direction. I plan to continue the exploration with the following potential goals:

(1) Automatic Root Cause Inference. The root causes can vary for different vulnerability types. For example, for taint style vulnerabilities, we should figure out the taint source (*e.g.,* a particular user-controlled function argument) and the missing or insufficient sanitization checks enabling the taint flow to the sink. For race vulnerabilities, it is essential to understand the specific thread interleavings leading to the race so we can adequately synchronize them to fix the issues. I will try first to define the root causes and critical components of different types of vulnerabilities, and then apply or devise dedicated program analysis methods to automatically infer them from the limited information attached to the dynamic testing reports, which will significantly accelerate the analysis and fix of the identified bugs.

(2) Automatic Impact Analysis. As revealed in our previous work [15], though a dynamic testing can effectively find bugs, it usually lacks a full understanding of the possible impact of the bug beyond the initial symptoms (*e.g.,* crash, sanitization failure, *etc.*). I want to extend the scope of our previous work to support a more comprehensive and deeper impact analysis, for instance, beyond the over-read or -write, whether the bug can eventually cause a privilege escalation or leak specific types of sensitive information. Such an analysis is complementary to the root cause analysis - while the latter is more of a "backward" analysis to trace the origin of the bug, the impact analysis focuses more on the "forward" part to reveal the bug consequences, which can help the developers better assess the found security issues and prioritize the high-profile ones for patch development.

3.4 Improve Static Analysis with Multi-Approach Combination

A large part of my previous work employ static analysis techniques to solve different security problems (*e.g.,* bug finding) due to its potential in systematic program reasoning, during this process, I have gradually developed a strong interest in improving the static analysis method itself, regarding both its accuracy and efficiency. Although it is fundamentally an undecidable problem to statically infer any non-trivial program properties (*e.g.,* the existence of specific bugs), there is still lots of space for improvement. One promising direction is to combine multiple different approaches with static analysis. For example, dynamic analysis can augment static analyzers with efficient and accurate instruction parsing (*e.g.,* for resolving complex pointer arithmetics). The dynamically collected runtime information (*e.g.,* concrete variable values) can also aid static analyzers for result verification and probabilistic state explosion control. The machine learning approaches can help the analyzer make a better decision regarding the sensitivity (*e.g.,* path-sensitivity) selection for different code regions based on their characteristics, achieving the desired balance between precision and performance in a more flexible way. Moreover, the human-in-the-loop approach is also worth an exploration - by developing ways to uniformly obtain, express, and convey domain knowledge to the static analyzers, we can eliminate many uncertainties and generate more reliable analysis results.

References

[1] Alavi, A., Quach, A., Zhang, H., Marsh, B., Haq, F. U., Qian, Z., Lu, L., and Gupta, R. Where is the weakest link? a study on security discrepancies between android apps and their website counterparts. In *International Conference on Passive and Active Network Measurement* (2017), Springer, pp. 100–112.

[2] ArsTechnica. How a few legitimate app developers threaten the entire Android userbase. `https://arstechnica.com/information-technology/2015/10/how-a-few-legitimate-app-developers-threaten-the-entire-android-userbase/`.

[3] Du, S., Zhao, M., Hua, J., Zhang, H., Chen, X., Qian, Z., and Zhong, S. Who moves my app promotion investment a systematic study about app distribution fraud. *IEEE Transactions on Dependable and Secure Computing* (2021).

[4] Gasparis, I., Qian, Z., Song, C., and Krishnamurthy, S. V. Detecting android root exploits by learning from root providers. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 1129–1144.

[5] Hao, Y., Zhang, H., Li, G., Du, X., Qian, Z., and Sani, A. A. Demystifying the dependency challenge in kernel fuzzing. In *44th International Conference on Software Engineering (ICSE 2022)* (2022).

[6] Jin, X., Xiao, X., Jia, S., Gao, W., Gu, D., Zhang, H., Ma, S., Qian, Z., and Li, J. Annotating, tracking, and protecting cryptographic secrets with cryptompk. In *IEEE Symposium on Security and Privacy (S&P)* (2022).

[7] LWN. A fuzzy issue of responsible disclosure. `https://lwn.net/Articles/904293/`.

[8] LWN. Filesystem fuzzing. `https://lwn.net/Articles/637151/`.

[9] Zhai, Y., Hao, Y., Zhang, H., Wang, D., Song, C., Qian, Z., Lesani, M., Krishnamurthy, S. V., and Yu, P. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 221–232.

[10] Zhang, H., Chen, W., Hao, Y., Li, G., Zhai, Y., Zou, X., and Qian, Z. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 811–824.

[11] Zhang, H., and Qian, Z. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 887–902.

[12] Zhang, H., She, D., and Qian, Z. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1093–1104.

[13] Zhang, H., She, D., and Qian, Z. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1663–1674.

[14] Zhang, Z., Zhang, H., Qian, Z., and Lau, B. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 3649–3666.

[15] Zou, X., Li, G., Chen, W., Zhang, H., and Qian, Z. {SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 3201–3217.