Research Statement

Xiaoyu Sun

Never before has any OS been so popular as Android. Existing mobile phones are not simply devices for making phone calls and receiving SMS messages, but powerful communication and entertainment platforms for web surfing, social networking, etc. Even though the Android OS offers powerful communication and application execution capabilities, it is riddled with defects (e.g., security risks, and compatibility issues), new vulnerabilities come to light daily, and bugs cost the economy tens of billions of dollars annually. For example, malicious apps (e.g., backdoors, fraud apps, ransomware, spyware, etc.) are reported [7] to exhibit malicious behaviours, including privacy stealing, unwanted programs installed, etc. To counteract these threats, our research fellows proposed plenty of works that rely on static analysis techniques to detect such issues. However, the static techniques are not sufficient to detect such defects precisely, which will likely yield false positive results as it has to make some trade-offs when handling complicated cases (e.g., object-sensitive vs. object-insensitive). In addition, the static analysis will also likely suffer from soundness issues because some complicated features (e.g., reflection, obfuscation, and hardening) are difficult to be handled [18, 14].

My research is among the first to tackle these problems directly: I develop several tools that attempt to automatically integrate program analysis techniques with dynamic testing techniques for detecting Android defects in real-world applications. My works are demonstrated to be more effective and precise in detecting more security issues than state-of-the-art program analysis tools. Specifically, my works highlight several security/compatibility aspects of the Android framework and unveil the discovery of various security/compatibility threats that were never been investigated before. My work on Android defects has received positive recognition from leading companies such as Alibaba Group. Moreover, I am currently collaborating with researchers from institutions all over the world, including **Google**, **Alibaba Group**, **Tencent AI Lab**, **University of Luxembourg**, **Washington State University**, **Swinburne University of Technology**, etc. In this statement, I highlight my previous works and my future research plans.

Research Contributions.

1. Detecting Compatibility Issues in Android Applications.

   - **JUnitTestGen** Our research fellows proposed various approaches to automatically detect compatibility issues in Android. Unfortunately, these approaches have only focused on detecting a limited range of compatibility issues (i.e., signature-induced compatibility issues), leaving other equally important types of compatibility issues unresolved. To this end, we propose a novel prototype tool, JUnitTestGen [17], to fill this gap by mining existing Android API usage to generate unit test cases. After locating Android API usage in given real-world Android apps, JUnitTestGen performs inter-procedural backward data-flow analysis to generate a minimal executable code snippet(i.e., test case). Experimental results on 10,000 real-world Android apps show that JUnitTestGen is effective in generating 22,805 distinct unit tests and locating 3,488 compatibility issues, 3,050 of which have never been discovered before.

   - **LazyCow** My thread of research continued to investigate other compatibility issues caused by the diversity of customized OS versions and Android models. I propose a novel, lightweight, crowdsourced testing approach, LAZYCOW, which enables the possibility of taming Android fragmentation through crowdsourced efforts. Specifically, crowdsourced testing is an emerging alternative to conventional mobile testing mechanisms that allow developers to test their products on real devices to pinpoint platform-specific issues. Experimental results on thousands of test cases on real-world Android devices show that

LAZYCOW is effective in automatically identifying 161 APIs with vendor-specific compatibility issues and 47 model-specific compatibility issues. In addition, after investigating the user experience through qualitative metrics, users' satisfaction provides strong evidence that LAZYCOW is useful and welcome in practice.

My thread of research moved from compatibility issues to security risks that might steal privacy from users.

2. Detecting Security Issues in Android Applications.

   - **DroidRA** DroidRA [18] is a state-of-the-art tool that aims at resolving reflection in Android apps. However, it suffers from low precision and incompleteness. Thus, I introduce a probabilistic-like approach to strengthen DroidRA, aiming at providing a more sound and comprehensive reflection analysis so that it can be used for analysing large-scale applications in practice. My main research contributions for this project are that I have improved DroidRA by 1) resolving more reflective targets, 2) reaching more reflective calls that have been overlooked and 3) reducing computational costs. We evaluated it over 1,000 Android apps and the number of reached and resolved reflective calls (i.e., 13,073 and 11,646, respectively (or 89.1% resolving rate)) is larger than that of the original version, which is respectively 11,952 and 9,375 (or 78.4% resolving rate). This result experimentally demonstrates the effectiveness of our improvements for DroidRA towards resolving reflective calls in Android apps.

   - **SEEKER** Sensor data are known to be leaked out because they are not protected by any permissions in Android. Thus, I design and implement a novel prototype tool, SEEKER [16], that extends the famous FlowDroid [3] tool to detect sensor-based data leaks in Android apps. SEEKER conducts sensor-focused static taint analyses directly on the Android apps' bytecode and reports not only sensor-triggered privacy leaks but also the sensor types involved in the leaks. Experimental results using over 40,000 real-world Android apps show that SEEKER is effective in detecting 1,964 sensor leaks in Android apps, and malicious apps are more interested in leaking sensor data than benign apps. Moreover, our improvement for supporting field sources detection has been merged to the original version FlowDroid via pull #385 on GitHub[20]), which we believe could be adapted to analyze other sensitive field-triggered leaks.

   - **HiSenDroid** My thread of research continued to investigate other malicious behaviours. We find that malware writers regularly update their attack mechanisms to hide malicious behaviour implementation avoiding detection. Motivated by this situation, I propose a static approach specifically targeted at highlighting hidden sensitive operations, mainly sensitive data flows. The prototype version of HiSenDroid has been evaluated on a large-scale dataset of thousands of malware and goodware samples on which it successfully revealed anti-analysis code snippets aiming at evading detection by dynamic analysis. We further experimentally show that, with FlowDroid, some of the hidden sensitive behaviours would eventually lead to private data leaks. Those leaks would have been hard to spot either manually among the large number of false positives reported by the state-of-the-art static analyzers, or by dynamic tools. Overall, by putting the light on hidden sensitive operations, HiSenDroid helps security analysts in validating potential sensitive data operations, which would be previously unnoticed.

Future Work

My long-term research goal is to do a comprehensive program analysis to ensure the security and reliability of software systems. I am committed to solving difficult software problems using practical and elegant solutions. As an independent principal investigator, I plan to develop more impactful tools for automatically detecting or solving software defects so that I can help to make a difference in our community both for end users and app developers.

Specifically, since the state-of-the-art static analysis techniques are not sufficient to find various defects in real-world applications. This would leave a bunch of defects undetected, causing serious problems for both users and developers. For example, attackers are reported to use complicated language features (e.g., reflection, obfuscation, and encryption) to hide malicious operations. So I have the ambition to propose new approaches to do code unification to perform much more sound and comprehensive results. Specifically, I plan to unify native code, and javascript with the bytecode in Android applications to achieve this. After enhancing the static analysis techniques, I plan to further apply such techniques to help detect vulnerabilities or software defects which are overlooked before. This line of thinking admits a number of opportunities for future research.

1. **React Native Code Unification** Android code unification to perform comprehensive static analysis of Android apps: In Android apps, dex bytecode cohabits with JavaScript code which can be used through the React Native Interface. Due to the challenge presented to analyze JavaScript code, it is most of the time overlooked by existing approaches. This limitation is a severe threat to validity since malicious behaviour can be implemented in JavaScript code. Therefore, I have the ambition to propose a model unifying both the bytecode and the JavaScript code in Android apps. I plan to propose a first step toward this direction at the call-graph level and with more granularity at the statement level relying on heuristic-based defined statements.

2. **Typestate Misuse Detection in Android** Android frameworks often provide complex functionality in order to support rich state transitions, which is a non-trivial task for programmers to understand and adopt correctly [2]. One essential aspect of correct API use is *typestate* [12], which defines legitimate sequences of operations that can be performed upon a given state type. A *typestate property* is a finite state machine [1], representing a sequence of behavioural type refinements such as "method A must be called after method B returns successfully". However, programmers are reported frequently ignore such disciplines [8], causing Android defects and lost productivity when typestate APIs are misused in practice [2]. Even worse, applications that fail to follow correct state transitions (i.e., typestate misuses) may also be infected with vulnerabilities [10, 11, 13, 22, 19], including data corruption [21, 5], privacy leaks [9, 15] and hijacking attacks [4, 6].

   To address this problem, I plan to design and implement a prototype tool to automatically detect typestate API misuse in real-world Android apps. Specifically, I am going to statically model the correct typestate usage based on the API comments in Android official documents, indicating the explicit disciplines of how typestate APIs should be performed. After that, I plan to perform intra- and inter-procedural data-flow analysis to extract the typestate API usage in real-world applications. After that, I will compare the typestate API usage with the correct typestate rules for pinpointing the violations. I believe that my tool can help app developers identify typestate misuse cases at the development stage to avoid these problems from being distributed into the community.

3. **Dynamic Software Testing** Given that the state-of-the-art works targeting software defects detection leverage static analysis techniques to achieve their purpose. However, as known to the community, the software detects cannot statically detect under every circumstance, introducing false positives and false negatives. To mitigate this problem, I plan to leverage dynamic program testing techniques to perform defects analysis as dynamic malware analysis methods generally provide better precision than purely static methods.

   To trigger malicious behaviours, I plan to develop a tool for automatically generating inputs that will trigger certain sensitive APIs (e.g., the APIs are protected by permission) in the source code. By dynamically triggering the target APIs, I am able to determine whether there exist malicious operations. In addition, I will then integrate my tool with state-of-the-art static taint analysis tools (e.g., FlowDroid [3]) to offer better precision.

References

[1] Finite-state machine. https://en.wikipedia.org/wiki/Finite-state_machine. Last updated: 2022-07-13.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, 2009.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[4] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

[5] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[6] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2017.

[7] Google. The google android security team's classifications for potentially harmful applications. https://developers.google.com/android/play-protect/phacategories. Online; accessed October 2022.

[8] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 47(11):2382–2400, 2019.

[9] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[10] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872, 2022.

[11] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 396–409, 2016.

[12] Ashish Mishra, Aditya Kanade, and YN Srikant. Asynchrony-aware static analysis of android applications. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 163–172. IEEE, 2016.

[13] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. Static asynchronous component misuse detection for android applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 952–963, 2020.

[14] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: a step towards android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1232–1244. IEEE, 2022.

[15] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[16] Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. Characterizing sensor leaks in android apps. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 498–509. IEEE, 2021.

[17] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. Mining android api usage to generate unit test cases for pinpointing compatibility issues. *arXiv preprint arXiv:2208.13417*, 2022.

[18] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. Taming reflection: An essential step toward whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–36, 2021.

[19] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[20] Xiaoyu Sun. Improve FlowDroid to support Field Sources(develop branch) . `https://github.com/secure-software-engineering/FlowDroid/pull/385`. Online; accessed 12 September 2021.

[21] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[22] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54, 2017.