On the Robustness of *Stochastic Stealthy Network* Against Android App Repackaging*

Ravshanbek Norboev
Temple University, and
North American University

Zakia Hossain
Temple University

Lannan Luo
University of South Carolina

Qiang Zeng
Temple University

***Abstract*—An unethical developer can download a mobile application, repackage it after making modifications, and then re-distribute it; the process is called *application repackaging*. Such attacks are rather common in Android markets, posing a severe threat to both app companies and users. Existing defenses against app repackaging attacks are mostly centralized and based on app similarity comparison. Recently, a novel defense, called *Stochastic Stealthy Network* (SSN) [1], was proposed. Unlike most existing defenses, it performs client-side detection of app repackaging. Our work analyzes the robustness of SSN by analyzing whether and how SSN can be bypassed by attackers. We have identified five different methods of bypassing SSN and present them in this paper. In addition, we give guidelines on designing a robust client-side defense against app repackaging.**

***Keywords*-App repackaging attacks; evasion attacks.**

I. Introduction

With the wide use of mobile devices, mobile app markets have grown quickly. Meanwhile, app piracy has been on the rise, and *application repackaging* is one of the most common and dangerous forms of infringement, as attackers not only make use of it to make profits, but also disseminate malware by inserting code into repackaged apps. According to a study [2], 86.0% of 1260 malware families were repackaged from legitimate apps, indicating application repackaging is a commonly used vehicle for mobile malware propagation.

Due to the importance and urgency of the problem, many defenses have been proposed. But most of them are based on app similarity comparison [3], [4], [5], [6], [7], [8]. They tend to be imprecise when handling obfuscated apps, and usually rely on a centralized trusted party to conduct detection, which is not very scalable considering the huge number of apps. Moreover, there are many alternative app markets, but their quality and commitment in repackaging detection are questionable [9]. Finally, users may download apps from places other than any app markets, such as FTP, and install them, bypassing the centralized defenses.

Recently, a novel precise and scalable defense, called *Stochastic Stealthy Network* (SSN) [1], was introduced to tackle the app repackaging problem. Instead of relying on a centralized party for defense deployment, SSN aims to prevent repackaged apps from working on user devices. Despite its novelty, its resilience to evasion attacks is not discussed and analyzed yet. *The goal of this work is to dissect the defense mechanisms of SSN, and analyze whether and how the defense can be bypassed*. In addition, based on the experiences gained from the analysis, we provide guidelines for future researches on building robust client-side defenses against app repackaging.

SSN is a compile-time enhancement, which inserts repackaging detection and response code into the app being protected, so that the inserted code runs interleaved with the original app code to detect whether the host app has ever been repackaged and respond to detected attacks. SSN captures a well-known fact that each app developer (or company) has a unique public key, which is part of an app, specifically, the app certificate. When an app is repackaged by another developer (i.e., the hacker), a different public key has to be used. SSN thus detects repackaging by comparing the original public key hidden in the code against the one contained in the app certificate; if they differ, an repackaging attack is detected.

While the repackaging detection method of SSN is straightforward, how to make the inserted detection and response operations resilient to attacks is challenging. SSN adopts multiple measures trying to overcome the challenge. For instance, the call for retrieving the public key of the app, i.e., `getPublicKey()`, is not issued directly but through *reflection*, such that text search cannot precisely pinpoint code containing this call; in addition, the detection operation is invoked probabilistically based on the return value of `rand()`, which explains "*stochastic*" in the name of the SSN defense. Given these measures, we examine whether SSN can be bypassed systematically and how. Our main approach is to employ various program analysis techniques that can be used to disable or delete the repackaging detection nodes, which all rely on invoking the API, *getPublicKey()*; regardless

*Technical report, Temple University, 2017. *Ravshanbek Norboev* joined this project when participating in the 2017 NSF Research Experiences for Undergraduates program at Temple University.

of the other intricate designs of SSN, as long as we can disable, delete, or calls to this API, SSN can be successfully bypassed. In addition, we propose attacks that modify the stochastic detection to be deterministic, in order to find the detection nodes.

We have identified the following distinct ways to bypass SSN:

- **Fuzzing**: during fuzzing, the attacker can make use of symbolic execution or manipulate the return value of `rand()`, in order to reveal as many detection nodes as possible. Both *blackbox fuzzing* and *whitebox fuzzing* can be applied.
- **Backward program slicing:** by extracting the program slice backward starting from a suspicious function call and then executing the slice, the attacker can determine the destination of a reflection call.
- **Code instrumentation:** the attacker can insert code before each reflection call to dynamically examine whether a reflection call leads to `getPublicKey()`.
- **Virtual function table hijacking:** the attacker may insert code into the repackaged app to manipulate the vtable or the vtable entry for `getPublicKey()`, such that such calls return fake values.

While some of the attacking methods require specific program analysis skills of attackers, others are actually trivial to conduct and can disable the detection nodes completely. The experiences gained in the process of analyzing the robustness of SSN comprehensively lead to valuable insights into designing client-side defenses against repackaging attacks. We thus give some guidelines on proposing more resilient defenses.

We made the following contributions.

- We analyze the robustness of SSN, a novel client-side defense against app repackaging attacks, and identify multiple evasion attacks that can be used to bypass SSN.
- To the best of our knowledge, this is the first work that exemplifies how to comprehensively examine the robustness of a client-side defense against app repackaging. We believe the analysis approaches can be adapted to analyzing other client-side defenses to evaluate their robustness.
- We share our insights with regard to devising more robust client-side defenses. The insights may help researchers avoid the same or similar pitfalls. It can also work as guideline for security researchers working on this problem.

The remainder of the paper is organized as follows. We briefly summarize the design of SSN in Section II, and describe the threat model against SSN in Section III. The evasion attacks that bypass SSN are detailed in Section IV, and we share our guidelines for designing such client-slide repackaging attack defenses in Section V. The paper is concluded in Section VI.

```
1 if(rand() < 0.01) {
2   funName = recoverFunName(obfuscatedStr);
3   // The reflection call invokes getPublicKey
4   currKey = reflectionCall(funName);
5   if(currKey != PUBKEY)
6     // repackaging detected!
7 }
```

Listing 1. Application Repackaging Detection in SSN.

II. The Design of SSN

There are mainly two motivations behind mobile app repackaging. First, an attacker repackages an app under their name in order to make profits, causing a financial loss to honest developers. Second, attackers, when repackaging a popular app, may insert malicious payloads, e.g., sending out users' private information and purchasing apps without users' awareness; and leverage the popularity of the original app to accelerate the propagation of the malicious one. Clearly, to maintain the health of the app ecosystem as well as for the security of mobile users, app repackaging detection is a critical problem to be addressed.

SSN is a defense technique that can be used by legitimate developers during compile time to build repackage detection and response capabilities into their apps, such that repackaging attack can be detected when the repackaged apps are run on user devices, and, if repackaging is detected, the response code prevents the apps from working normally.

SSN detects repackaging by detecting the change of the public key contained in the app certificate. Each developer or app company has its own public/privte key pair. An Android app, no matter it is legitimate or repackaged, has to be digitally signed using the private key before it is released, and the public key contained in the certificate, which itself is part of the app, is used to verify the signature. Thus, the public key contained in a certificate, which is a unique identification of an app developer, can be leveraged to determine whether an app has been repackaged by an attacker. SSN inserts many detection nodes in to an app, and each node checks the change of the public key to detect repackaging. Once repackaging is detected, the response code will be activated to prevent the repackaged app from working properly on user devices.

Listing 1 illustrates a repackaging detection node inserted by SSN. It detects repackaging by comparing the app's current public key with the original public key, `PUBKEY`, which is embedded into the code inserted by SSN; the current public key is retrieved through a call to the Android system service API `getPublicKey`. Plus, a hash can be used on both `PUBKEY` and `currKey`, such that an attacker cannot search the literal value `PUBKEY`

to locate repackaging detection nodes. In order to hide the call from attackers, SSN proposed the following measures: (1) repackaging is only invoked probabilistically, as shown in Line 1, to hide the repackaging detection nodes from attackers running the app; (2) the function name "*getPublicKey*" is obfuscated, so attackers cannot find the word in the code; the call is issued through *reflection* (Line 4), which requires the function name to be recovered (Line 2), though. (3) after repackaging is detected, instead of responding to it immediately, the response is delayed to confuse the attacker who analyzes the anomalies.

III. Threat Model

We then discuss possible adversary attacks against SSN. Not only can the threat model be used to examine SSN, but may also be adopted to evaluate the resilience of other client-side repackaging detection.

Text search. An attacker may search for specific text patterns, such as "*getPublicKey*", to locate repackaging detection code. In the case of SSN, it hides calls to `getPublicKey` through reflection calls and transform some normal calls into reflection calls as well, so it is resilient to such attacks. Note that text search for `PUBKEY` (Line 5) will fail, since instead of using the original public key, SSN can derive a value from `PUBKEY` using a custom hash to eliminate the literal key value from the code.

Manual running and debugging. An attacker may install the repackaged app and run it on an emulator or a real device. Whenever suspicious symptoms arise, the attacker may use a debugger to trace back to the repackaging detection and response code. Such dynamic analysis works only when repackaging detection is executed. An attacker may try to *intercept* critical calls the repackaging detection code relies on. For instance, an attacker may hook calls to `getPublicKey` in order to locate the repackaging detection code. However, running a protected app manually in order to trigger all or most of the repackaging nodes is too costly, so we regard SSN resilient to such attacks.

Blackbox fuzzing. An attacker may use blackbox fuzzing to run the repackaged app by providing a large number of random inputs to trigger as many logic bombs as possible [10], [11]. For every activated bomb, the attacker can trace back and disable it. We will show in Section IV that, by manipulating the return value of `rand()`, we can turn the probabilistic activation of detection nodes into deterministic ones, such that whenever a path containing a detection node is executed, the node can be surely revealed by an attacker. That is, the design goal of "stochastic" activation of detection nodes in SSN fails.

Whitebox fuzzing. Various techniques have been proposed to explore execution paths in a program. A dynamic analysis based approach is to *explore multiple paths* during execution [12]. *Symbolic execution* has been widely applied to discovering inputs that execute program along specific paths [13], [14], [15], [16], [17]; it uses symbolic inputs to explore as many execution paths as possible, and resolves the corresponding path conditions to find the concrete inputs. Recent research has shown that symbolic execution is an effective approach to discovering conditional code and identifying trigger conditions [16]. When symbolic execution is applied to SSN, Line 1 cannot stop symbolic executor from exploring (and hence exposing) the path containing repackaging detection.

Backward program slicing. An attacker may simply circumvent trigger conditions and execute payloads directly. E.g., given a line of suspicious code, an attacker may perform *backward program slicing* starting from that line of code, and then execute the extracted slices to uncover the payload behavior [18]. Or, the attacker may apply *forced execution* to directly execute the code that looks suspicious [19]. Take SSN as an example: an attacker can circumvent Line 1 to execute the following code; thus, SSN is vulnerable to such attacks.

Code instrumentation. An attacker may modify code to assist attack. In SSN, e.g., the attacker can insert code right before a suspicious reflection call to check the destination of the call, i.e., `if(funName=="getPublicKey")`, such that the inserted code can reveal repackaging detection dynamically when running on the user side.

Vtable hijacking. If a defense relies on specific API calls, an attacker may perform vtable hijacking attack to return fake values. Such attacks can fool SSN by returning fake values when `getPublicKey` is invoked.

In short, while SSN is immune to *text search* and resilient to *manual running and debugging*, it is vulnerable to five other types of attacks, which are discussed in Section IV in detail.

IV. Bypassing SSN

While SSN takes multiple measures to make repackaging detection code stealthy, there are still some weaknesses that allow attackers to bypass the defense. To analyze the effectiveness of SSN, we became the attackers and tried to weaken the SSN defense. Our observations include (1) SSN relies on a specific Android API (i.e., `getPublicKey()`) to be effective, so it involves a single point of failure, and (2) the probabilistic repackaging-detection feature is meant to expose only a small portion of detection nodes to attackers; however, by manipulating the return value of the random number generation function, the probabilistic detection can be turned to be deterministic, such that all detection nodes encountered during execution are exposed. This section presents in-depth details on adversary techniques that can bypass SSN.

A. Blackbox Fuzzing

A blackbox fuzzer generates a massive number of random or semi-random inputs to execute a program under test. According to the design of SSN, as a repackaging-detection (RD) node is only activated at a very low probability (see Line 1 in Listing 1), even an execution path encounters a RD node is encountered, it is probably not revealed to the attacker who runs the app to find RD nodes. Considering that the attacker, who feeds a large number of random inputs in order to reveal RD nodes, cannot afford too much time to analyze an app, it is expected that a large number of RD nodes survive the attacker's analysis.

However, the approach that SSN achieves probabilistic activation is based on checking the return value of a random number generation function (see Line 1 in Listing 1). Thus, if an attacker intercepts the invocation of `rand()` and returns a fake value (e.g., 0), the RD node is activated deterministically.

There are many random input generators for the Android system, such as Monkey [20], PUMA [21], AndroidHooker [22], and Dynodroid. It is worth noting that the attack that manipulates `rand()` calls does not depend on blackbox fuzzing. For instance, an attacker may apply manual running in order to invoke all or most of the functionalities of the app, aiming to reveal the RD nodes involved in the related execution paths.

On the other hand, given a *large* app, it is difficult to achieve a high code coverage through manual running or blackbox fuzzing. If an execution path is never explored during the adversary analysis, the RD node in that path can survive. Therefore, this evasion attack method can only be regarded as semi-effective.

B. Whitebox Fuzzing

Whitebox fuzzing is a form of automatic dynamic test generation based on symbolic execution and constraint solving, designed for security testing of large applications [23]. The idea of whitebox fuzzing is to use a fixed input, which symbolically executes the program, gathering input constraints from conditional statements encountered along the way. The collected constraints are then systematically negated and solved with a constraint solver, yielding new inputs that exercise different execution paths in the program.

Thus, whitebox fuzzing can be leveraged to explore as many execution paths as possible; and during the path exploration, it can reveal the destination of the suspicious reflection calls (see Line 4 in Listing 1). There has been many symbolic executors that have been built to analyze Android apps, including [24], [25], [26], [27], [28]. For example, Jensen et al. proposed to use concolic execution to build summaries of individual event handlers and then generate event sequences backward, in order to find event sequences that reach a given target line of code in the Android app [26].

```
8  if(rand() < 0.01) {
9    funName = recoverFunName(obfuscatedStr);
10   // The reflection call invokes getPublicKey
11   currKey = reflectionCall(funName);
12   if(funName == ``getPublicKey'')
13     currKey = PUBKEY;
14   if(currKey != PUBKEY)
15     // repackaging detected!
16 }
```

Listing 2. Bypassing SSN based on code instrumentatioin.

While it is well known that whitebox fuzzing improves code coverage compared to blackbox fuzzing, it does not guarantee complete path exploration. Thus, we do not claim this attacking method can completely bypass SSN.

C. Backward Program Slicing

Program slicing is a decomposition technique that extracts statements relevant to a particular computation [29]. It is usually used in debugging to locate source of errors easily and faster. There are two forms of slicing: backward slicing and forward slicing. A backward slice is constructed from a target in the program, and all data flows in this slice end of the target. It is a version of the original program that can be executed. An important property of any backward slice is that it preserves the effect of the original program on the variable chosen at the selected point of interest within the program. Typically, it can assist a developer to locate the parts of the program containing a bug.

We propose to apply backward program slicing to bypassing SSN. Specifically, we take a suspicious reflection call (Line 4 in Listing 1) as a target and extracts the backward slice, so that we can compute the value of `funName` from the slice to check whether it is equal to *"getPublicKey()"*. Since SSN uses relatively simple logic to hide the value of `funName`, it should be easy to get the backward slice and reveal its value.

D. Code Instrumentation

Code instrumentation is performed by adding statements to software in order to monitor performance and operation of the software during runtime. To bypass SSN, we can insert code to check the destinations of all reflection calls and "fool" SSN.

Listing 2 shows a concrete example how to perform code instrumentation on the app protected by SSN. By inserting two simple lines of code (Lines 12 and 13) after each reflection call, on the user device side, it can check the destination of the reflection call and manipulate the variable used to store the return value (that is, `currKey`) of the call to `getPublicKey`. Therefore, in order to

bypass SSN, the attacker simply insert such code into an app before releasing the repackaged app.

E. Vtable Hijacking

A virtual table (vtable) is a lookup table function used to resolve function calls in a dynamic binding manner. An object in a Java program contains one pointer towards a vtable, which contains function pointers to the implementation of the methods associated with the object.

By manipulating the whole vtable or the vtable entry for `getPublicKey`, an attacker can arbitrarily control the return value of calls to `getPublicKey`. The opensource project, ARTDroid [30], illustrates how to manipulate a targeted vtable entry, but it requires the root privilege to launch the attack. Even without the root privilege, an attacker should be able to insert crafted native code into an app to manipulate the vtable entry. Note that the paper of SSN also briefly points out the threat of vtable hijacking attacks.

Summary: We have presented five different methods to bypass SSN. while **blackbox fuzzing** and **whitebox fuzzing** cannot guarantee bypassing SSN completely, the other three methods indeed can. Among the three, the attacking method based on **code instrumentation** is trivial to launch. The analysis demonstrates that *SSN, despite its novelty, is a very weak defense against app repackaging detection*.

V. Guidelines for Designing Client-Side Repackaging Detection Defenses

Guideline 1: the main challenge in proposing a robust repackaging detection technique is *how to protect repackaging detection code from attacks*; to examine the robustness of the technique, the designers should take into account various adversary analysis, such as static analysis, dynamic analysis, and combined analysis methods. Nowadays, many advanced analysis techniques have become mature. For example, backward slicing [18] and symbolic execution [16] are considered in our analysis. Thus, a comprehensive examination of the resilience of the proposed detection technique is necessary. For instance, if the detection code demonstrates itself with some patterns, it will be trivial to identify and eliminate the related code. As another example, if the detection capability relies on specific system APIs and the call sites of those APIs can be located, an attacker can either modify the API calls or manipulate the return values. Moreover, rather than regarding the repackaging detection as secret, researchers should propose a technique that is resilient under whitebox analysis, because attackers may propose novel custom analysis once the repackaging detection technique is known.

Guideline 2: While SSN only uses the change of the public key as the indication of repackaging attacks, a repackaged app demonstrates itself with various explicit modifications, such as icons, code hash values, and company names. They all can be used as features to detect repackaging attacks. Moreover, SSN triggers the detection logic probabilistically; however, the probabilistic mechanism can be easily manipulated by attackers through controlling the generation of random numbers. *How to keep the probabilistic trigger advantage meanwhile avoiding the easy manipulation of attackers is an interesting problem but also a challenge.*

Guideline 3: self-modifying code has been proven to be a very effective obfuscation approach, and this obfuscation technique can be used to hide the detection code. Specifically, the detection code can be transformed into native code, which is then encrypted into data and is only recovered when being executed. In order to prevent the attackers from simply deleting suspicious repackaging detection code, a vtable way is to stitch together the detection code and the original function code (i.e., the detection code is weaved into the original app code). This can defeat attacks that try to delete suspicious code.

Guideline 4: it is important to not break the original app and make sure the overall overhead due to the repackaging detection small and ideally negligible.

Our recent work [31] has proposed a novel and robust client-side defense technique against app repackaging attack, following all the guidelines. We propose a creative use of *logic bombs*, which are regularly used in malware, to conquer the main challenge described in **Guideline 1**. A novel bomb structure is invented and used: the *trigger conditions* are constructed to exploit the differences between the attacker and users, such that a bomb that lies dormant on the attacker side will be activated on one of the user devices, while the repackaging detection code, which is packed as the bomb *payload*, is kept inactive until the trigger conditions are satisfied. Moreover, the repackaging detection code is *woven* into the original app code and gets *encrypted*; thus, attacks by modifying or deleting suspicious code will corrupt the app itself. We have implemented a prototype, named BombDroid [31], that builds the repackaging detection into apps through bytecode instrumentation, and the evaluation shows that the technique is effective, efficient, and resilient to various adversary analysis including symbolic execution, multi-path exploration, and program slicing.

VI. Conclusion

Repackaging attacks have become a severe threat to the Android ecosystem, infringing the IP of honest developers and disseminating malicious code. Among the many defenses against repackaging attacks, SSN is a novel client-side defense. Our work, however, has shown that the technique actually can be bypassed by attackers in multiple ways, and some of them are actually easy to conduct, for example, code instrumentation to detect calls to specific APIs. Our analysis demonstrates that SSN is

not resilient to evasion attacks. By no means this work is to negate the contribution and novelty of SSN, but it demonstrates the extraordinary difficulties in designing a new resilient defense technique.

Based on our analysis experiences, we have gained valuable insights into the problem. The guidelines based on such insights hopefully can inspire other researchers who work on tackling the critical repackaging attack problem. In addition, we have briefly introduced our recent work, BOMBDROID, which follows the guidelines in its design and stands for a new state-of-the-art client side defense against app repackaging.

REFERENCES

[1] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *DSN*, 2016.

[2] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *S&P*, 2012.

[3] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *ESORICS*, 2012.

[4] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *WiSec*, 2014.

[5] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among Android applications," in *DIMVA*, 2013.

[6] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *ICSE*, 2014.

[7] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *CODASPY*, 2012.

[8] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *In Engineering Secure Software and Systems*, 2012.

[9] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "Andradar: fast discovery of android applications in alternative markets," in *DIMVA*, 2014.

[10] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013.

[11] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *FSE*, 2013.

[12] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *S&P*, 2007.

[13] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "BitScope: Automatically dissecting malicious binaries," in *Tech. Rep. CMU-CS-07-133*, 2007.

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

[15] J. R. Crandall, G. Wassermann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, "Temporal search: Detecting hidden malware timebombs with virtual machines," in *ACM Sigplan Notices*, 2006.

[16] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards detecting logic bombs in android applications," in *S&P*, 2016.

[17] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 225–238.

[18] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *NDSS*, 2016.

[19] J. Wilhelm and T. cker Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *International Workshop on Recent Advances in Intrusion Detection*, 2007.

[20] UI/Application Exerciser Monkey, 2017, http://developer.android.com/tools/help/monkey.html.

[21] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014.

[22] AndroidHooker, 2016, https://github.com/AndroidHooker.

[23] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[24] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," in *Software Engineering Notes*, 2012.

[25] S. Anand, M. Naik, H. Yang, and M. J. Harrold, "Automated concolic testing of smartphone apps," in *FSE*, 2012.

[26] C. S. Jensen, M. R. Prasad, and A. Moller, "Automated testing with targeted event sequence generation," in *ISSTA*, 2013.

[27] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: automated system input generation for android applications," in *ISSRE*, 2015.

[28] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1043–1054.

[29] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[30] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime." in *IMPS@ ESSoS*, 2016.

[31] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized android application repackaging detection using *Logic Bombs*," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2018.