Secure Repackage-Proofing Framework for Android Apps using Collatz Conjecture

Haoyu Ma, Shijia Li, Debin Gao, Chunfu Jia*

Abstract—App repackaging has been raising serious concerns about the health of the Android ecosystem, and repackage-proofing is an important mitigation against threat of such attacks. However, existing app repackage-proofing schemes were only evaluated against trivial adversaries simulated using analyzers for other purposes (e.g., disclosing privacy leakage vulnerabilities), hence were shown "effective" mainly because their key programming features were not even supported by those toolkits. Furthermore, existing works have also neglected dynamic adversaries capable of manipulating victim apps at runtime, making them vulnerable against such stronger opponents. In this paper, we propose a novel repackage-proofing framework, which deploys distributed detection and response sites into the subject app's native partition to cross-verify all its code files. The detection sites transmit obtained integrity metrics to response sites via secure communication channels built on the subject app's own control flows using a specialized obfuscation technique based on Collatz conjecture, turning the repackage-proofing process into complicated implicit flows that are intrinsically difficult to be resolved due to the conjecture's nonlinear dynamical behaviors. We evaluated our framework using sophisticated Android data-flow analyzers. Results showed that our prototype effectively impeded analyses aiming to trace the information flows of its cross-verification.

Index Terms—App repackaging, repackage-proofing, code obfuscation, Collatz conjecture.

1 Introduction

Mobile devices, especially smart phones, have become important players in ubiquitous computing, while Android is currently considered to be the most popular operating system for mobile devices [1], [2]. This unsurprisingly drew all kinds of malicious activities against the platform, with app repackaging being an important means. A typical purpose of app repackaging is to re-publish a victim app either as a new one or a mimic, which seems to be the authentic version but in fact tampered in a way to fulfill certain collateral malicious purposes, e.g., to deprive economic benefits via compromised in-app purchases and/or advertisements, to allow piggybacked malicious payload to be executed. To this end, the adversary usually modifies internal logic of the victim app first, then packages the compromised instance with a new signing key so that after it is published, unwitting users might be lured into using it. Past studies showed that 5% to 13% of apps were plagiarisms in Android markets [3], and among 1,260 malicious apps which were comprehensively investigated, 86% were propagated via app repackaging [4]. More recent studies [5], [6] further showed that more and more sophisticated tricks from traditional desktop malware samples have now emerged in app repackaging cases, including adding hooks, hiding malicious payload within resource files, mounting obfuscation, and VM-aware mechanisms. App repackaging today has even started to challenge machine-learning-based detection techniques [7].

- *Haoyu Ma is with the School of Computing and Information Systems, Singapore Management University, Singapore and the School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, China. E-mail: hyma@xidian.edu.cn.*
- *Debin Gao is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: dbgao@smu.edu.sg.*
- *Shijia Li and Chunfu Jia are with the College of Cyber Science, Nankai University and the Tianjin Key Laboratory of Network and Data Security Technology, Tianjin, China. E-mail: cfjia@nankai.edu.cn.*
- *Chunfu Jia is the corresponding authors of this paper.*

One countermeasure against app repackaging inherits the idea of software tamper-proofing, and aims to construct Android apps with embedded capability of fighting off integrity violations, which is widely known as *repackage-proofing* [8]. To the best of our knowledge, five representative repackage-proofing schemes have been made public as of 2020, namely Droidmarking [9], Stochastic Stealthy Network (SSN) [8], [10], AppIS [11], BOMBDROID [12], [13], as well as Self-Defending Code (SDC) [14]. Unfortunately, these works either neglected certain intrinsic flaws which make them vulnerable against targeted attacks, or relied on programming tricks which appeared to be secure merely because existing analysis tools never intended to support those tricks. The theoretical security basis of these existing schemes, however, is much weaker than claimed.

In this paper, we propose `AppWarder`, a novel and secure Android repackage-proofing framework. Given a subject app, the proposed framework enforces repackage-proofing protection with a payload built in the form of distributed code snippets consisting of

- a collection of *detection sites* built to obtain the the subject app's integrity status, and
- a separate collection of *response sites* built to verify readings returned by the detection sites and react to potential integrity violations.

These code snippets are deployed at various randomly selected positions of the subject app's native partition, and they are designed to provide protection independently for

additional resilience against removal attacks. The embedded detection and response sites communicate according to complex interdependencies, allowing AppWarder to work with a delayed and probabilistic repackage responding strategy. Instead of retrieving well-known integrity metrics from an app's static files, the detection sites of AppWarder are designed to parse loaded code sections of the app and verify their in-memory status, making AppWarder capable of providing protection against dynamic adversaries. The retrieved integrity metrics are passed to the response sites through communication channels implemented on top of an existing Collatz-conjecture-based control flow obfuscation approach [15]. Collatz conjecture provides a unique program structure that *unfolds according to a unique orbit when initiated with a distinct positive integer, whereas for all other positive integers in general, it unfolds in a pseudo-random manner*. This unique orbit, when built into the communication channels of AppWarder, turns its parameter passing processes (i.e., the propagation of obtained integrity metrics) into complicated implicit flows, making them intrinsically difficult to be analyzed via data-flow analysis. In this way, AppWarder manages to integrate its payload into the subject program's own semantics (as part of the obfuscated control transfers) so that attempting to compromise its protective logic risks damaging the subject program as well.

To assemble a complete solution, AppWarder adopts different repackage responding strategies to form a stochastic verification mechanism which works either locally or in cooperation with a remote server. Specifically, our remote repackage responding strategy turns the response behavior into a proprietary protocol of which the consequence is controlled by a remote entity, making it easier to disguise such consequences into irrelevant errors.

We have evaluated the effectiveness of AppWarder on both aspects of defending against runtime deceiving attacks launched by dynamic adversaries and against off-line attacks based on sophisticated software analysis. Put in short:

- with a comparative analysis on the source of integrity metrics adopted by AppWarder and existing competitors, we showed that our method could resist most types of runtime deceiving attacks to which other existing method are vulnerable; and
- further simulations showed that data flow analysis using a state-of-the-art static analysis framework for Android, namely Argus-SAF[1], was also thwarted by AppWarder, whereas the generic design adopted by SSN was shown to be in fact traceable once Argus-SAF was extended with a customization to support tainting a specific resource of Android.

In addition, by studying low-level details of Argus-SAF in processing apps protected with AppWarder, we confirmed that complexity of the control and information flows of such protected apps had been significantly increased by our communication channel design.

The rest of this paper is organized as follow. In Section 2, we briefly introduce related works on app repackaging detection and repackage-proofing, as well as the threat model and assumptions considered in this work. Following that, in Section 3, we present the detailed design of AppWarder. In Section 4, we explain the implementation of our repackage-proofing framework. Section 5 presents our evaluation on its effectiveness. We further discuss some implications on this new repackage-proofing framework in Section 6, including its compatibility with new security mechanisms to be added in future Android distributions. Finally, we conclude the paper in Section 7.

2 Background

2.1 Repackage Proofing

In essence, repackage proofing is software tamper proofing that targets Android apps specifically. As such, the task of a repackage-proofing scheme includes detecting and responding to integrity violations occurred to both the subject app and the payload of itself. To this end, it is necessary for a repackage-proofing scheme to protect its payload from being resolved by the potential adversaries via program analyses.

Early Android apps were written in Java only. Consequently, many of the existing repackage-proofing schemes, including those mentioned in Section 1, protect their payload using two obfuscation approaches:

- transforming key API calls of the repackage-proofing payload into recorded reflections, with class and method names of their callees ciphered before invocations; and
- encoding large sections of the repackage-proofing payload using self-decrypting code based on one-way functions [16].

Both of the above approaches exploit features which Android inherited from Java, and the idea is to turn key elements required for resolving semantics of the repackage-proofing payload into secrets to be released on-the-fly such that the adversaries cannot rely on static analysis to reveal and compromise the protected payload.

Meanwhile, another repackage-proofing design proposed in the past, namely SSN [8], adopted a "delayed responding" strategy by turning its payload into distributed components so that the integrity measuring and verifying operations are carried out separately, hence increasing the complexity of dependencies between the two operations. Furthermore, SSN transmits data among its distributed components via "stealth communication channels" to prohibit attempts of tracing, which are implemented by exploiting the R class feature of Android[2].

Unfortunately, the aforementioned defensive strategies had underestimated the strength of app repackaging adversaries. To begin with, note that during a repackaging attack, the adversary is assumed to have full access and control over all resources contained in the subject app's APK. As a result, it's possible for the adversary to manipulate the subject app's code sections (bytecode or native) in the off-line phase to hijack any API calls that are considered suspicious. Existing studies have already

1. https://github.com/arguslab/Argus-SAF.

2. The R class is a dynamically generated class that reflects the various values defined in an app's resource files. See https://developer.android.com/reference/android/R for its definition.

pointed out that both the encoded reflection and the self-decrypting code can be defeated by a dynamic adversary who rewrites the related key instructions (e.g., invocations to `java.lang.reflect.Method.invoke` and `mprotect`) to redirect control to a customized handler, where the true destination of reflection calls can be revealed and the dynamically decrypted code sections can be compromised right before execution [17], [18].

The previously proposed design of stealth communication channels [8], [10] is also weak mainly due to the lack of solid security basis. In fact, this design was only tested against laboratory toolkits originally built for the purpose of detecting information leakages, in which tracing the R class is not supported merely because it was considered unnecessary for the targeted application scenario. For a determined adversary who is willing to pay effort to extend a data-flow analysis toolkit [19]–[22] to cover features like the R class, the protection added by existing communication channels can be trivially broken. We demonstrate this in our evaluation given in Section 5.

In summary, existing Android app repackage-proofing schemes are either weak on their security basis or are constructed on top of flawed protection mechanisms. In this paper, we propose `AppWarder` which leverages a well-accepted theoretical limitation of data-flow analysis. Unlike the existing schemes, the integrity measuring mechanism of the proposed framework also takes the resilience against dynamic adversaries into consideration.

2.2 Control Flow Obfuscation using Collatz Conjecture

Collatz conjecture, or the *3x+1 problem*, is an unsolved conjecture of great importance in mathematics[3]. The problem involves computing the so-called *Collatz function*, a mapping $\theta : \mathbf{N}^* \to \mathbf{N}^*$ where for any $n \in \mathbf{N}^*$,

$$\theta(n) = \begin{cases} n/2 & (if\ n \equiv 0\ mod\ 2), \\ 3n+1 & (if\ n \equiv 1\ mod\ 2). \end{cases} \tag{1}$$

Let $\theta^0(n) = n$, and let

$$\theta^k(n) = \underbrace{\theta \circ \cdots \circ \theta(n)}_{k \text{ times}} = \theta(\theta^{k-1}(n)), \tag{2}$$

the conjecture asserts that there always exists a $\delta_n \in \mathbf{N}^*$, such that $\theta^{\delta_n}(n) = 1$. Meanwhile, the procedure of computing $\theta^{\delta_n}(n)$ leaves behind a particular number sequence $\Lambda(n) = \{n, \theta(n), \theta^2(n), \cdots, 1\}$, which is called the *Hailstone sequence* of n. The assertion of Collatz conjecture also implies that, given $n_1, n_2 \in \mathbf{N}^*$, $\Lambda(n_1) = \Lambda(n_2)$ holds only if $n_1 = n_2$, i.e., the number of distinct Hailstone sequences is as many as that of members in $\mathbf{N}^*$. This characteristics makes Collatz conjecture "a deterministic process that simulates 'random' behavior" [23] — *while the Hailstone sequence of each particular number in $\mathbf{N}^*$ is both determined and unique, behavior of the conjecture is in general hard to predict*. Moreover, the implementation of the conjecture involves an unrolling loop controlled by a symbolic value, making it intrinsically difficult to resolve using state-of-the-art program analysis tools based on symbolic execution.

This characteristics has already been exploited as the basis of control flow obfuscation [15]. Specifically, given a to-be-protected conditional branch, this obfuscator surrounds it with a Collatz conjecture loop while introducing the natural number (again, let it be denoted by n) being computed by the conjecture into the conditional logic of the branch, such that the modified logic is functionally unchanged only if $n = 1$. As a result, at runtime, the obfuscated branch would always work correctly because Collatz conjecture asserts that n will eventually become 1; to static program analysis tools, on the other hand, n can only be considered as a symbolic value, thus the Collatz conjecture loop surrounding the protected branch could effectively prohibit the program analysis via path explosion [24].

In this paper, we employ the same characteristics of Collatz conjecture with n being (dependent on) a new and reliable integrity metric, and build implicit information flows from it to form an ideal security fundamental of the secure communication mechanism in `AppWarder`. We further introduce our detailed design (which is built on top of the aforementioned obfuscation) in Section 3.4.

2.3 Threat Model and Assumptions

As mentioned above, existing repackage-proofing schemes are ineffective mainly because these designs underestimated the capability of their opponents. Therefore, contrary to the previous works, the threat model considered in this paper takes both the static and dynamic adversaries into account. Be more specific, both types of adversaries are assumed to have full control to the subject app's APK, namely

- they are assumed to be able to analyze the APK offline with any possible means; and
- they could rewrite code and/or other resource files contained in the APK during the process of repackaging.

The difference between a static and a dynamic adversary is that *code rewriting done by a dynamic adversary involves injecting payload to further manipulate the subject app's behavior on-the-fly*. Specifically, the injected payload could analyze and/or modify the app's code/data at runtime, with the timing of such tampering controlled by the dynamic adversary.

On the other hand, compared to the assumed adversaries, the capability of `AppWarder` is limited within the reasonable margin of normal third-party apps. That is,

- the proposed framework cannot rely on any feature that requires root privilege; and
- it should not leverage any hidden APIs since they could be deprecated without notice.

Taking the rapid evolving pace of the platform into consideration, we believe that supporting the old Dalvik virtual machine that had deprecated since Android 5.0 is no longer worth the effort. As an evidence, according to the statistics provided by Android Studio[4], as of July 2020, the distribution of DVM based Android systems (i.e., Android 4.4 and

3. As a well-known mathematical problem, a detailed introduction of this conjecture can be found at https://en.wikipedia.org/wiki/Collatz_conjecture

4. Readers could refer to Google's official explanation on "Distribution dashboard" at https://developer.android.com/about/dashboards.

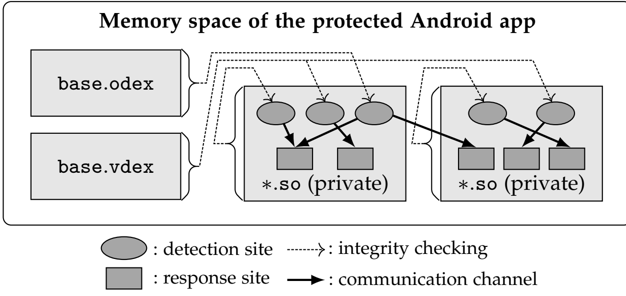


Fig. 1: Overview of the proposed verification framework of AppWarder.

lower) is only 6.6%. Therefore, AppWarder is designed to target only the currently in-use Android runtime environment, namely the ART runtime.

3 METHOD

In this section, we begin with an overview on the architecture of AppWarder, and then further explain in detail the design of key components of AppWarder.

3.1 Overview

Figure 1 illustrates the architecture of AppWarder. Similar to AppIS and SSN, the payload of AppWarder consists of a large amount of distributed detection and response sites to establish a cross-verification network. The difference is that components of AppWarder are deployed only in the subject app's native partition. Note that our framework is not the first to utilize native level repackage-proofing payload. However, unlike existing schemes [11], [18] in which the motivation of this design choice is mainly to increase the difficulty of program analysis, AppWarder focuses more on reaping another benefit of native code — the availability of pointers. Being capable of manipulating pointers allows AppWarder to check the in-memory status of code files belonging to the subject app, making it capable of detecting potential runtime code tampering conducted by dynamic adversaries.

More specifically, AppWarder deploys a payload of distributed detection and response sites, in which

- a *detection site* obtains the subject app's (partial) integrity status; and
- a *response site* verifies readings returned by a subset of the detection sites and reacts to potential integrity violations.

To achieve good resilience, AppWarder utilizes different detection sites to measure different code files of the subject app while having each code file of the app cross-checked by multiple detection sites distributed over the app's different execution paths. Meanwhile, each individual response site of AppWarder verifies the integrity metrics retrieved from multiple detection sites. These together create complex interdependencies between components of our framework which operate in a probabilistic manner. To further obscure the behavior of AppWarder, the response sites are set to operate according to one of two strategies, which, upon detecting a potential anomaly, either stops the app directly with a local crash or cooperates with a remote entity to deny service or initiate code repairing. Details on the code file measuring approach of our detection sites are further explained in Section 3.2, while the local and remote response strategies of our framework are presented in Section 3.3.

Most importantly, AppWarder's detection sites communicate the integrity metrics with its response sites via a group of conditional control transfers selected from the subject app and transformed them using specialized Collatz-conjecture-based obfuscation [15]. Such obfuscated control structures integrate AppWarder's semantics with that of the subject app as implicit flows that make AppWarder's data flow difficult to be traced, providing a security basis that is technically strong against both types of assumed adversaries. We explain details of this communication channel design in Section 3.4.

3.2 Repackage Detection

Intuitively, a repackage-proofing defense needs all code partitions of the subject app to undergo its integrity verification. As stated in Section 3.1, AppWarder deploys native level payload so that code of the subject app can be verified after being mapped into its address space.

Earlier Android apps consist mainly of components written in Java or Kotlin (a general-purpose programming language announced by Google). With the emerging of Android NDK, however, more and more apps today also contain native components written in C/C++. When released in the form of APK,

- the Java/Kotlin components of an app are typically compiled into Dalvik bytecode and encoded in one or more DEX files;
- the C/C++ components, on the other hand, are compiled into private native libraries.

When an app is started in the ART runtime, both its DEX files and private libraries are mapped into its memory space. Before Android 8.0, ART compiles the app's DEX files at install time and merges the resulting native instruction stream together with the DEX files into an OAT file (**O**ptimized **A**head of **T**ime) named base.odex, which is mapped into memory upon app starting. Starting from Android 8.0, however, ART stopped merging the DEX files into base.odex, and instead introduced a new VDEX format (**V**erified **D**alvik **EX**ecutable) specifically for the purpose of verifying and merging DEX files. As a result, an additional base.vdex file is generated at install time and is mapped alongside base.odex on app starting. Furthermore, although ART claims ahead-of-time compiling as one of its key features, our real-device tests showed that bytecode inside the DEX files is not guaranteed to be fully compiled at install time. Based on these observations, and in order to ensure the capability of properly inferring the subject app's integrity at all times, the detection sites of AppWarder are designed to measure the app's bytecode from its base.odex or base.vdex file (depending on the version of Android system), as well as the native code located in its private libraries; see Figure 1.

3.2.1 Integrity of DEX files

Recall that regardless the specific versions of the ART runtime, all DEX files of an app are merged into one particular

file when being mapped into the address space. Specifically, before Android 8.0, DEX files are integrated into the app's OAT file named base.odex and located in a read-only .oatdata segment together with the OAT header. For higher versions of Android, DEX files are, instead, encoded into the app's VDEX file named base.vdex, which consists of only a single read-only section. A DEX file by definition carries a checksum within its header, which also exists after its integration into the OAT/VDEX file. The ART runtime does not allow incorrect DEX checksums, and will assert that on-the-fly with a strict file legitimacy verification process. As such, AppWarder directly checks the embedded DEX checksums from the subject app's OAT/VDEX file to infer the integrity of its DEX files. This requires our detection sites to perform on-the-fly special-purpose parsing on the OAT/VDEX files.

To give an illustrative example, assume that the goal is to measure the first DEX file of an app, which is embedded as a data section in its base.odex file built by an Android 7.0 platform. According to the format definition of OAT, a group of OatDexFile structures are maintained right after the OAT header and a key_value_store section. Each OatDexFile structure corresponds to one of the app's DEX files, with its offset within the OAT file specified by a dex_file_offset field. Let d_1 and d_2, respectively, denote the offset of the demanded OatDexFile structure within the base.odex and that of the dex_file_offset field to the beginning of this OatDexFile, and further let d_3 be the offset of the targeted DEX file's checksum field to the beginning of its header, then the process of indexing from the beginning of base.odex to this DEX checksum is as demonstrated in Figure 2. Here, if we assume that the beginning of base.odex is indexed by a pointer p, then the checksum of the target DEX file can be found at the address $*(*p+(d_1+d_2))+d_3$, where "$*$" indicates pointer dereferencing. Note that offsets $\{d_1, d_2, d_3\}$ can be calculated off-line and will remain unchanged across app re-installation because

- for a fixed APK, the ART runtime always merges its DEX files according to a fixed sequence; and
- once the Android version is known, the adopted OAT/VDEX format can be safely assumed.

Therefore, AppWarder's detection sites only need to obtain p at runtime in order to correctly perform the demanded parsing. As shown above, the functionality of the parser here consists of no more than a few pointer operations, making it ultra-light-weight and thus easy to hide or obfuscate. Last but not the least, the process of checksum retrieving for Android 8.0 and above is very much the same as in the aforementioned example, except that the detection sites need to parse the subject app's base.vdex file instead.

According to project LIEF[5], format of the DEX header and the OatDexFile structure has never been changed since Android 6.0. Although the format of OAT header had been modified once, the change occurred in Android 8.1 where OAT no longer contains DEX files and is therefore not the target of our detection sites anymore. For the format of VDEX files, on the other hand, our study on recent versions of Android source code found one major modification on its

5. This is an open-source project which supports parsing, modifying, and abstracting executable formats of Android; https://github.com/lief-project/LIEF.

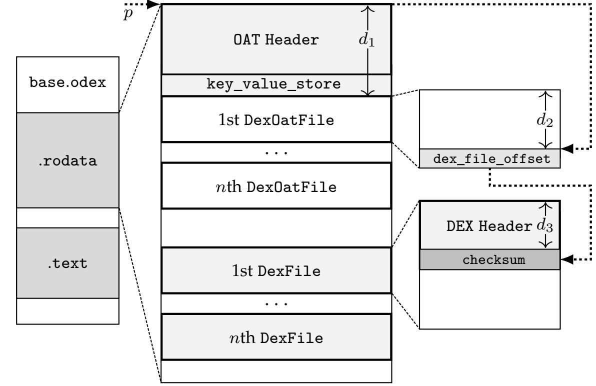


Fig. 2: General idea of retrieving a DEX file's checksum by parsing base.odex at runtime.

general layout, which took place in Android 9.0. As such, using checksum of DEX files for repackage detection shows little risk on the compatibility aspect. To correctly determine which file should be parsed to retrieve DEX checksums as well as the exact layout of the targeted file, AppWarder simply needs to test the existence of base.vdex and check the version of its header if it exists.

The acquisition of OAT/VDEX file pointer (i.e., the pointer p in the above example) inevitably relies on accessing the subject app's memory map at runtime, which is also carried out with diversified implementations. Besides directly reading /proc/self/maps, a virtual file of Android's proc filesystem (procfs) which records memory mapping of the subject app's process, another option adopted by AppWarder is to execute Linux command pipelines to write selected lines of this virtual file into a temporary file, and then obtain the OAT/VDEX file pointer from the latter. The involved command lines can be stored in the form of ciphers and be revealed only before they are sent to the system shell, which also increases the difficulty of resolving semantics of the related code components.

3.2.2 *Integrity of private libraries*

As discussed in Section 3.1, in order to establish a complete cross-verification network AppWarder also needs to check the integrity of the subject app's native partition in addition to its DEX files. Also recall that to retrieve the subject app's DEX file checksums, AppWarder needs to correctly obtain its OAT/VDEX file pointer, and therefore cannot avoid using the file system APIs (e.g., fopen()) and/or shell command execution APIs (e.g., system(), popen(), and execve()). Taking dynamic adversaries into account, these APIs could be unreliable because their call conventions might be intercepted at native level. For example, the adversary could redirect all invocations of fopen() to his hooked functions such that when AppWarder later acquires the app's memory map to locate the OAT/VDEX file (without loss of generality, assume this is done by reading the virtual file /proc/self/maps using fopen()), the adversary can hijack the acquisition process and feed our framework with false information. To perform such hooking, a dynamic adversary typically needs to tamper with *PLT (Procedure Linkage Table) stubs* owned by the subject app's private libraries, which are address look up stubs that handle control transfers towards imported functions in relocatable native executables. As such, in order

TABLE 1: Register holding the callee's entry address in the JNI dlsym lookup stub for different ISAs.

ISA	Designated register
ARM64	x16/w16
ARM	r12
Mips/Mips64	t9
x86/x86_64	eax/rax

to effectively protect a private native library of the subject app, `AppWarder` has to measure the integrity of both

- its code section, for detecting potential logic tampering caused by app repackaging; and
- its PLT section, for defending against API hijacking that might be launched by dynamic adversaries.

And given that `AppWarder` cannot simply trust the subject app's memory mapping before asserting the integrity of the aforementioned key system APIs, it needs another way to locate the library files at runtime.

As in other Linux-based operating systems, Android maps the executable, read-only, and writable segments of a `.so` library maps according to a fixed sequence, making offsets between two specific instructions inside the library a fixed value. Based on this observation, `AppWarder` introduces an obfuscated addressing method to effectively index key sections of the subject app's private libraries for their integrity measuring. Specifically, we looked into the JNI call convention of Android and found a particularly useful feature for fulfilling this purpose: *when invoking a native method from a Java method using JNI, the JNI call convention leaves behind a code pointer referring to the callee's entry point after the invocation*. That is, for all supported Instruction Set Architectures (ISAs), the procedure of a JNI call ends with a *dlsym lookup stub* which loads the callee's entry point into a particular register to be used later by a subsequent tail call. This means: *at runtime, a JNI native function can easily obtain the function pointer of itself*. Table 1 lists the registers used in different ISAs involved in this convention, all of which are general-purpose registers that would not raise suspicion if they are used in a general data processing instruction (such as `mov`). Exploiting this mechanism, `AppWarder` deploys those detection sites assigned to measure a particular private library of the subject app into JNI functions inside that library, and makes sure that these detection sites can access the register listed in Table 1 before the function pointer left in it is overwritten. With this function pointer as a *pivot addresses*, the deployed detection sites could then index any positions within executable/read-only segments of the subject library using offsets determined off-line[6].

One advantage of building `AppWarder`'s native-level integrity measure on top of such an addressing method is that it automatically introduces pointer aliasing that helps impeding efforts of analyzing our protections, because detection sites using different pivot addresses would naturally refer to the same position of a library with different offsets. In addition, `AppWarder` sees code and PLT sections to be checked as binary strings, and divides them arbitrarily into collections of overlapping substrings via an off-line analysis. This allows our framework to infer the integrity of those sections by verifying the integrity of divided substrings instead, as long as each of the divided substrings is covered by at least one detection site. Finally, although the naive implementation of pivot address acquisition requires only a simple inline assembly instruction, to conceal the purpose of involving the particular register used in the aforementioned JNI convention, `AppWarder` introduces scrambled instructions to further increase the difficulty of identifying such behaviors. A simple example is to add random masks on the pivot addresses before copying them out of the register, and having the masks removed later at the related detection sites. These designs together introduce many uncertainties into the detailed implementation of `AppWarder`'s detection sites, and further increases the adversary's difficulty of revealing and/or compromising them.

6. This is viable since a similar idea had been exploited by an existing work to launch code reuse attacks [25]

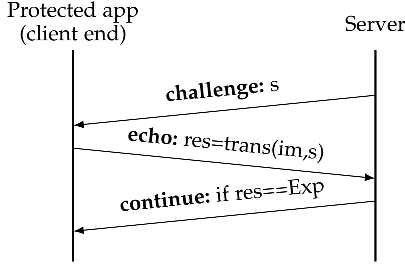
3.3 Repackage Response

Many Android apps communicate with remote servers as part of their functionalities. Taking this into account, `AppWarder` applies two different strategies in responding to app integrity anomalies detected.

- For apps which require substantial interactions with their servers to be functional, `AppWarder` introduces a proprietary handshake protocol to let the server side respond to client-side integrity violations.
- In other cases, `AppWarder` uses a delayed and controlled failure mechanism [26] as its local repackage response strategy.

Figure 3 illustrates the idea of the proprietary handshake protocol. Let im denote an integrity metric collected by a detection site of the subject app (i.e., the client end of the protocol), and $trans(\cdot)$ denote a transformation (e.g., an efficient hash function) performed by a response site (also located at the client side). As the first step of the protocol, `AppWarder` generates a random salt s from the server, and sends it to the app as a challenge. The in-app response site then uses both s and im to compute a response $res = trans(im, s)$, and returns it back to the server. Interactions between the two parties can only continue if the value of res is as expected (represented by Exp in Figure 3); otherwise, the server responds to a potential integrity violation by either disconnecting from the client or starting a patching process to have the repackaged instance repaired. The main advantage of the remote response strategy is that it allows `AppWarder` to disguise the response behaviors into other types of events that appear to be irrelevant to the defense. For example, when having a response that disconnects from the client, the error could be reported as connection time out or poor network performance; similarly, a self-patching response can be done either silently or as an update to a new version.

For the local response strategy, existing works have already provided many valid implementations that delay and control failure responses, e.g., by causing controlled memory corruptions that would later be triggered at undetermined timings [26], [27]. More recently, SSN adopted a stealthy response approach by causing logic malfunctions in the subject app, which is done by modifying the app's integer variables or attributes of its Button, TextView, EditText objects [8]. `AppWarder` adopts the controlled memory



s: a random salt im: an integrity metrictrans: a client end transformation
res: client end response Exp: expected value of the repsonse

Fig. 3: proprietary handshake protocol of `AppWarder` as a remote repackage response strategy.

corruption approach given that memory allocation is tightly related to user actions, making the consequences of controlled memory corruptions harder to predict in actual uses of the subject app. We omit details of this implementation given that these approaches are not our contribution.

3.4 Secure Communication between Detection and Response Sites

As discussed in Section 3.1, for the asynchronous verification process of `AppWarder` to work, integrity metrics need to be passed from corresponding detection sites to response sites in a secure manner; otherwise data-flow analysis by a static or dynamic adversary could potentially trace any exposed communication to other payload of `AppWarder`. To give a motivating example, Algorithm 1 demonstrates a trivial case of detection-response site communication, where im, the integrity metric returned by a detection site $detect(\cdot)$ is directly compared with the corresponding baseline value bl by a response site using a conditional branch, and a $respond()$ behavior is triggered should the verification fails. Such a direct integrity metrics comparison is sufficient to fulfill the goal of repackage-proofing; however, by tainting the code section in this code snippet (represented by "app_code"), an adversary could easily trace the data flow originated from this assumed detection site to the conditional logic where im and bl are compared, and consequently compromise the repackage-proofing routine by making sure that $respond()$ never gets triggered.

Algorithm 1: A trivial (and inadequate) case of detection-response site communication in repackage-proofing

```
...
im = detect(app_code);
if im != bl then
|  respond();
end
...
```

To mitigate this issue, `AppWarder` adopts a customized communication channel which carries out the communication in the form of implicit data flows that are intrinsically difficult to be analyzed. On top of that, `AppWarder` also binds the integrity of related detection sites to the correctness of the subject app's original semantics, making them difficult to be removed without compromising the original

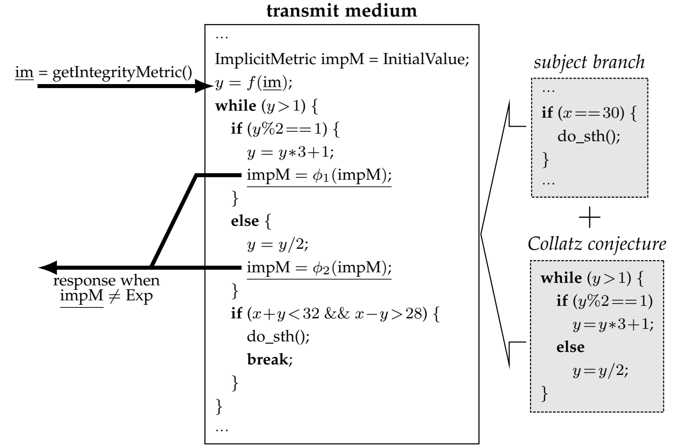


Fig. 4: An intuitive example of the communication channel of `AppWarder`.

app's functionality. As mentioned in Section 2.2, the design of our communication channel is based on the control flow obfuscation technique built on top of Collatz conjecture [15]. Figure 4 illustrates the idea of our communication channels. Again, let im denote an integrity metric obtained from a detection site (as in Figure 3). Our communication channel creates a projection between im and another variable $impM$ with only implicit dependency between the two. This allows `AppWarder` to detect integrity anomalies by retrieving im with its detection sites while verifying $impM$ in its response sites, with the connection between the two portions of the operation unable to be resolved using data-flow analysis. To this end, a conditional branch in one of the subject app's C/C++ functions is obfuscated using a specialized Collatz-conjecture-based obfuscation, creating a *transmit medium*. As an example, assume that the subject branch to be obfuscated guards a `Do_sth()` module with the predicate "$x{=}{=}30$". In our specialized obfuscation, a spurious integer variable $y>0$ is derived using a customized function $f(\cdot)$ of im, and then the subject branch is embedded into a Collatz conjecture loop controlled by y, with the above predicate replaced by the following combinatory logic:

$$\begin{cases} x+y<32 \\ x-y>28 \end{cases}. \tag{3}$$

On top of that, in the two branches of the Collatz function (namely the **if-else** structure within the transmit medium in Figure 3), a pair of distinct non-linear mapping of $impM$ is deployed. That is, let these mappings be denoted respectively by $\phi_1(\cdot)$ and $\phi_2(\cdot)$, then $\phi_1(\cdot)\neq\phi_2(\cdot)$ must hold.

The above transformation turns the subject branch into a dual-purpose construct. On one hand, it still handles the subject app's own control flow: as explained in Section 2.2, Equation 3 is not satisfiable until y yields to 1, at which point it becomes "$29<x<31$" and is functionally equivalent to "$x{=}{=}30$" with x being an integer. On the other hand, when executing this obfuscation construct, $impM$ is computed by a composition of $\phi_1(\cdot)$ and $\phi_2(\cdot)$ determined by the Hailstone sequence of y. As long as the subject app is intact, y takes a fixed sequence of values determined by im. As a result, by fixing the initialization of $impM$, this obfuscation construct becomes a robust projection from im to $impM$.

Thus when execution exits the obfuscation construct, response sites could refer to the value of $impM$ to determine whether they should act on a potential integrity violation. In the meantime, if a detection site which provides im is exposed, an adversary could only leverage this information to trace AppWarder's behavior to the communication channel, but cannot further connect the information flow to $impM$ due to the lack of direct value assignment. Furthermore, being an obfuscation structure, logic of the communication channel itself is also difficult to be revealed using automatic analysis tools.

To make the example given in Figure 4 more motivating, assume $im = 27$ when the subject app is not tampered with, and the $InitialValue$ for $impM$ is 1. In addition, assume

$$\begin{cases} f(im) = 30 - im \\ \phi_1(impM) \ = 2 \cdot impM \\ \phi_2(impM) \ = impM + 5 \end{cases} . \tag{4}$$

This setting gives $y = 30 - 27 = 3$, i.e., the Hailstone sequence of y is $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$. Therefore, the Collatz conjecture loop controlled by y iterates for a total of 7 rounds, with $\phi_1(\cdot)$ executed in the 1st and 3rd round, and $\phi_2(\cdot)$ reached in the other rounds. As a result, the final value of $impM$ is (deterministically) given by

$$impM = \phi_2^4(\phi_1(\phi_2(\phi_1(1)))) = 34. \tag{5}$$

However, assume that im is changed to 23 after a repackaging attack, and $f(\cdot)$ outputs $y = 7$ as a consequence (we omit the detailed Hailstone sequence of 7 for simplicity). The final value of $impM$ in this case would instead be given by

$$impM = \phi_2^4(\phi_1(\phi_2^3(\phi_1(\phi_2^2(\phi_1(\phi_2(\phi_1(\phi_2(\phi_1(1)))))))))) = 242. \tag{6}$$

Therefore, AppWarder is able to monitor the subject app's integrity status by comparing the value of $impM$ with 34 (denoted as Exp in Figure 4). Here, with im not directly used to assign $impM$, but instead controlling the program logic computing its value, an implicit flow from im to $impM$ is therefrom established. We will further show how such intended implicit flows are used as the security basis of AppWarder in Section 5.2.

4 Implementation

We implemented a semi-automatic toolkit of AppWarder (with more than 1,200 lines in Java and 2,000 lines in C/C++), consisting of a Gradle[7] plugin (currently based on Gradle v5.4.1) which manages the payload deployment procedure in general, and a Clang-based code-rewriting module built using LLVM's Libtooling[8]. Given that AppWarder aims to include native code partition of the subject app into its surveillance while having itself deployed in that partition as well, we adopt *partial code-data isolation* in the embedded payload. To be specific,

- offsets for indexing the app's DEX checksums and the baselines for verifying these checksums (henceforth the DEX offsets/baselines for short) are allowed to be used as immediate operands within AppWarder's instructions;
- offsets and baselines for indexing and verifying the app's native code (henceforth the native offsets/baselines), on the other hand, can only be put inside data section of the respective native libraries and referenced using addresses in the code of our framework.

This layout allows AppWarder to complete transforming the subject app's native code partition before determining value of key parameters required for verifying its integrity.

4.1 General Work Flow

Note that the Gradle tool chain of Android Studio builds the APK of a subject app in the general order of *native→Java→class→DEX→APK*. That is, it first compiles the app's C/C++ source into .so libraries, then compile the Java source into .class files, and then uses the above files to generate DEX files and finally makes the APK. To comply with this general procedure, the pipeline of AppWarder is designed as shown in Figure 5.

To begin with, our toolkit modifies the AST (Abstract Syntax Tree) of the app's C/C++ source and deploys a semi-manufactured version of AppWarder payload, and then compiles the modified code into preparatory .so libraries (step ①). This involves

- distributing detection and response sites into different native functions of the app;
- based on the app's call graph, choosing detection sites that serve as sources of to-be-verified integrity metrics for each of the response site; and
- finally, embedding the communication channels to fulfill the aforementioned correspondences.

Note that at this step, content of these libraries are still incomplete. Specifically, the DEX offsets/baselines as well as the native baselines required by AppWarder's detection/response sites have not been computed yet. However, by reserving space for such undetermined parameters using padding, our plugin is able to guarantee that position of code items in the preparatory libraries do not shift due to any subsequent parameter updating. Therefore, the native offsets of these libraries can be computed in this step.

Next, AppWarder compiles the app's DEX files using both its Java source and the preparatory libraries (step ② and ③). With these files generated, our plugin can determine the DEX offsets and baselines and update the corresponding parameters in its payload (step ④). This would at last complete code sections of the previously generated preparatory libraries, allowing AppWarder to further compute and update the native baselines for its payload (also in step ④). In order to launch the above parameter updating at the most appropriate timing, our plugin hooks the Gradle task for generating DEX files to enable itself to take over immediately after these files are built. Finally, after finishing preparing all the code files, our plugin makes the APK of the subject app's repackage-proofed instance (step ⑤).

Figure 6 shows a proof-of-concept demo of AppWarder deployed on Frozen Bubble, a FOSS (Free and Open Source Software) app that can be found from both the F-Droid catalog[9] and Google Play. Specifically, Figure 6.a and .b

7. https://gradle.org/.
8. https://clang.llvm.org/docs/LibTooling.html.
9. https://f-droid.org/en/.

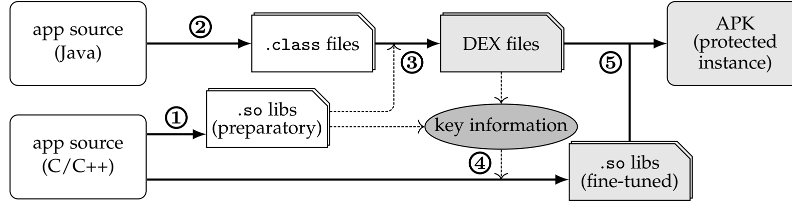
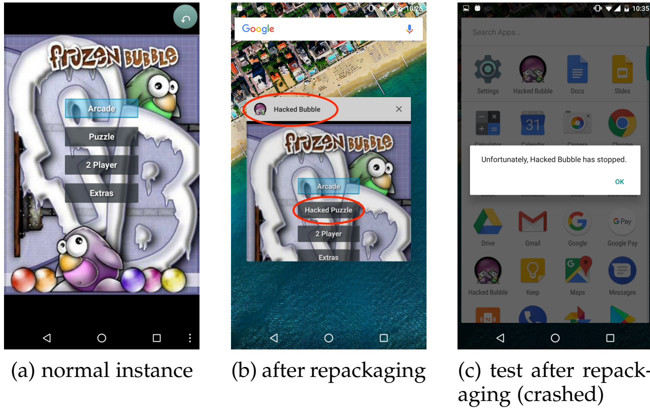


Fig. 5: Implementation pipeline of `AppWarder`.



(a) normal instance (b) after repackaging (c) test after repackaging (crashed)

Fig. 6: A proof-of-concept demo of `AppWarder` responding to a trivial app repackaging attack.

respectively demonstrates the UI of our demo as well as a repackaged instance of it. And as presented in Figure 6.c, when the repackaged instance runs, our repackage-proofing framework causes it to crash in the matter of seconds.

4.2 Deployment of Detection/Response Sites

Thanks to its secure communication mechanism, `AppWarder` is able to adopt a flexible strategy in deploying its detection and response sites. In general, given a subject app, our toolkit analyzes its call graph and embeds detection and response sites of `AppWarder` into randomly selected native methods. Due to the native-level addressing approach mentioned in Section 3.2.2, methods that can directly be invoked via JNI are given priority. Other methods with large code bodies are also preferred because it is less likely for the embedded code pieces to become dominating components. On the other hand, our toolkit rules out

- methods for error/exception handling only, because the chance for them to be executed could be slim;
- methods called iteratively, to avoid causing unnecessary performance burden.

Due to the same reason, our toolkit also analyzes control flow graph of the selected methods to avoid placing detection/response sites of `AppWarder` into loop bodies.

The dependencies between `AppWarder`'s detection/response sites are established after their embedding. Specifically, after a detection site has retrieved an integrity metric reading, address of the metric will be updated into a dynamically maintained lookup table (referenced by a global pointer). Any response site assigned to verify this metric will first check the lookup table, and proceed only if the corresponding entry is not null. Android apps typically contain many functional modules that do not follow a strict execution sequence, thus the behavior of response sites would vary according to specific user behaviors. Consequently, this makes the cross verification mechanism of `AppWarder` probabilistic and more difficult to resolve via static program analysis. This lookup-table-based response site implementation is important because without such a design, `AppWarder`'s detection/response dependencies can only be established in a deterministic manner, which would significantly limit the available locations for deploying response sites (and consequently, the overall complexity of `AppWarder`'s payload). For example, any response site can only be placed at a location where all detection sites it depends on are guaranteed to be executed, and, detection/response dependencies crossing different Android activities/services can no longer be allowed because these functional modules may not follow a strict execution sequence.

To make the above implementation less suspicious to adversaries, `AppWarder` maintains multiple lookup tables for different detection and response sites, which are built in various forms (e.g., arrays, instances of structs/classes, and etc.). The timing of lookup table updating is also flexible. A table entry can be updated either after a detection site finishes metric acquisition, or after the retrieved metric has been transformed by a secure communication channel.

5 Effectiveness

There were generally two theories on which effectiveness of a repackage-proofing scheme should be established. The first was proposed by Droidmarking [9], which suggested abandoning stealth of its payload and relying only on the resilience provided by the self-decrypting code mechanism. Unfortunately, self-decrypting code can be vulnerable over dynamic adversaries. The second and more conservative idea, as used in SSN [8], saw repackage proofing as a special code tamper-proofing defense and accordingly valued both stealth and resilience of its payload. The design of `AppWarder` adopts the second theory, hence in this section, we begin with a discussion on the stealth of its key components, and then test the resilience of its communication channels via simulations. Finally, we provide a brief evaluation on the potential performance overhead of `AppWarder`.

Note that we do not take the timeliness of `AppWarder`'s response activation (namely, the time it takes for a repackaged instance to crash/exit as the repackage-proofing response) as a key component in our evaluation, because this depends almost exclusively on the specific placement of repackage-proofing payload. To give an example, one could easily achieve good timeliness by having a repackage detecting and responding routine embedded in the subject

app's entry method to make it immediately respond to potential integrity violations upon app starting. However, such a strategy is likely unreliable as the responding routine (with relatively unique location as its signature) could be revealed and removed by an adversary. Therefore, we trade this timeliness property for better security resilience with a more randomly distributed detection and response sites.

In addition, it is necessary to point out that software-based protection approaches can typically be cracked as long as a determined adversary is willing to put in sufficient time and resource. The security goal of our framework is more about raising the bar, i.e., to increase the difficulty of pulling off a repackaging attack to an extent that doing so becomes significantly inefficient compared to developing an app of same function from scratch or reverse engineering the critical logic.

5.1 Stealth

The stealth of repackage proofing mainly deals with the difficulty for static analysis techniques to distinguish payload of the repackage-proofing scheme from the original app. Since the definition of this concept makes it hard to be quantified via experiments, we instead provide a high-level analysis on the stealth of `AppWarder`.

As described in Section 3, the payload of `AppWarder` consists mainly the detection and response sites as well as the communication channels for passing key parameters between them. The construction of these components involve 1) data reading referenced by pointers, 2) condition-controlled loops, 3) dynamic memory allocations, 4) network communications, 5) file reading/writing operations, and 6) executing shell commands. All of these are common operations that can be found in normal apps, indicating that the instruction-level signature of `AppWarder`'s payload is not significant. Particularly, the remote response strategy of `AppWarder` works in the form of a proprietary protocol, making response sites adopting this strategy unlikely to be recognized. And in real-world practices, many apps use shell commands (e.g., `ping` and `ls`) for purposes like testing network status and locating required files.

In addition, by encoding parameters of API calls in its payload, specifically the name of files to be operated and the shell commands to be executed, `AppWarder` could make it more difficult for static analysis to determine which instructions serve for its functionality. This effect could be enhanced by modifying the protected app's own file operations and/or shell command execution behaviors by encoding their parameters as well, which would further increase the stealth of `AppWarder`.

5.2 Resilience

Note that in the evaluation of `AppWarder`'s stealth, only those approaches available to static adversaries were considered. This is because for a dynamic adversary, `AppWarder` would inevitably expose part of its payload at runtime. Exploiting this, such an adversary may resort to more sophisticated program analysis techniques, e.g., dynamic binary instrumentation (with tools like `Frida`[10]) or data-flow analysis, to trace the verification procedure of `AppWarder`, hoping that the small amount of detection and/or response sites exposed could lead to more payload discovery. Alternatively, the dynamic adversary could attack `AppWarder` by hijacking key system API calls of Android (see Section 3.2.2), so that the verification process of our framework can be deceived while not being compromised directly.

10. https://github.com/frida.

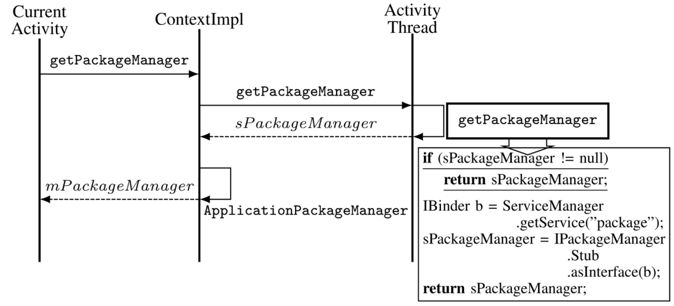


Fig. 7: Android's work flow of obtaining the interface of `PMS`

5.2.1 Security against API hijacking

To begin with, it's necessary to understand that native-level hooking (as mentioned in Section 3.2.2) is not the only type of API hijacking capable of defeating the previously proposed app repackage-proofing schemes. Java APIs provided by services of the Android framework are also vulnerable to such type of malicious manipulations. To give an example, most existing repackage-proofing schemes use the app's signing key as an integrity metric, which is obtained via Android's `PackageManagerService` (`PMS`). Before invoking any `PMS` APIs, an app must obtain an interface of this service by calling `getPackageManager`. According to the work flow shown in Figure 7, this interface is eventually converted from the static field `sPackageManager` within the app's `ActivityThread` object, and then returned by method `ActivityThread.getPackageManager`. Also note that `ActivityThread.getPackageManager` directly returns the current value of `sPackageManager` if it is not NULL (to avoid performance overhead caused by further consulting the Android framework). Therefore, by corrupting `sPackageManager` with a pointer of the hook function before anything else in the victim app tries to obtain the `PMS` interface, a dynamic adversary could fake all `PMS` API calls and hence counterfeit the related integrity verification.

Table 2 lists the integrity verification approaches that have been adopted by `AppWarder` and that can be found in past related literature, as well as the API dependency status of these approaches. We can see that while most of the existing repackage-proofing schemes adopted multiple ways of integrity checking, all the previously proposed approaches either rely on Java/native APIs of Android framework. We have implemented a collection of proof-of-concept demos of the known API hijacking attacks aiming to bypass repackage-proofing protection[11], and have verified that all known sources of integrity metrics that have been adopted in the existing repackage-proofing schemes can be undermined by at least one of these demos.

11. See https://github.com/jnsiw/AWARE for our attack demo and example cases.

TABLE 2: Comparison on different integrity check methods adopted by AppWarder and the existing repackage-proofing schemes (regarding API dependency).

integrity checking method	API dependency	
	rely on Android services?	rely on native API?
singing key verification [8], [9], [12]	√	×
MANIFEST.MF digest verification [12]	√	√
code checksum verification (using memory mapping) [11], [12], [14]	×	√
AppWarder:DEX parsing	×	√
AppWarder: code checksum verification (using residual function pointer)	×	×

On the other hand, while AppWarder's file parsing based DEX checksum verification mechanism may also suffer from attacks using PLT hooking, our native-level integrity measuring process, which locates code sections to be checked using JNI residual function pointers, is (to the best of our knowledge) the only anti-repackaging verification mechanism that cannot be targeted by API hijacking. With this native-level integrity checking as a security root, together with the cross-verification network architecture, AppWarder could therefore effectively fight against potential dynamic attacks based on API hijacking (unlike other schemes proposed in past works).

5.2.2 Security against Dynamic Binary Instrumentation

First of all, AppWarder is an application-level defense not designed to withstand attacks and analysis from the operating system, kernel, or debugger. In fact, the reason why AppWarder had to resort to the complex detection- and response-site communication using Collatz conjecture is to make sure that its repackage-proofing payload cannot be reliably removed by an attacker given successful dynamic analysis from the operating system, kernel, or debugger.

That said, AppWarder is able to withstand dynamic binary instrumentation techniques carried out at user level, such as Frida. In analyses assisted by such dynamic binary instrumentation techniques, an attack inevitably involves tampering with in-memory status of the app's native code files to attach the DBI library and place hooks, which will be picked up by our native-level integrity measuring process.

5.2.3 Security against data flow analysis

Against attacks based on data flow analysis, the resilience of AppWarder mainly relies on its communication channels. Note that a similar idea was adopted in another repackage-proofing scheme, namely SSN [8], which leveraged Android's R class feature (Recall Section 2.1) as the communication medium and relied on reflections to achieve the data transmission. However, such a design was thought to be secure merely because at the time of the proposal, analysis tools targeting Android have not yet supported tracing the aforementioned app features, which is no longer the case today. To demonstrate this, we extended a test case in DroidBench[12], namely FieldSensitivity3, to obtain the SIM serial number of the device and transmit it to an information leakage sink using the R-class-based communication channel[13]. Source code of the extended test case is shown in Figure 8.a, where an R class object test was created (line 30) with its field ic_launcher selected to be the communication medium (line 36). The extended test case was then analyzed using Argus-SAF, a state-of-the-art analysis framework for Android, under its component-based taint analysis mode. The result showed that Argus-SAF successfully traced the embedded information leakage through the R-class-based communication channel. Specifically, as demonstrated in Figure 8.b, the taint path obtained by Argus-SAF precisely located the reflection operations set(·) and get(·) where the R class fields were manipulated (highlighted at line 28 to 30). This proved that the existing communication channel design proposed in SSN is in fact insecure even if it is implemented using reflections.

Contrary to the R-class-based design, the communication channels of AppWarder are constructed on an implicit value assignment mechanism which exists as a side effect of the Collatz-conjecture-based obfuscation, which therefore establish their security basis on a intrinsic limitation of data-flow analysis. Specifically, under-tainting due to implicit flows is a well-known disadvantage of taint analysis [29], hence standard data-flow analysis based on such techniques cannot propagate through AppWarder's communication channels because of the implicit value assignment mechanism. A simple alternative is to alter the taint policy to perform propagation for every branch whose condition was tainted. The tradeoff of doing so, though, is severe over-tainting [30], which will bury dependencies related to payload of AppWarder into numerous irrelevant taint traces. More recent data-flow analysis tools on Android utilize symbolic execution to provide precise control-flow graph (CFG) and support annotation-based analysis. For example, Argus-SAF, or to be exact, its JNI data-flow analysis component JN-SAF [21], relies on symbolic execution backed by angr [31]. However, our communication channels are by all means still Collatz-conjecture-based obfuscation constructs, which are designed specifically to impede symbolic execution by exploiting an intrinsic weakness of the technique, namely the path explosion problem.

To make the evaluation comparable, we have verified the effect of our communication channels using Argus-SAF again. Because AppWarder is deployed in the native partition of the protected apps, we adopted native_leak[14], the official native flow benchmark of the toolkit, as the subject for this test. This app passes the device's IMEI code to a native method send(·) once its request for READ_PHONE_STATE permission is granted, and the latter leaks the IMEI. We modified the send(·) method to pass the IMEI through our communication channel, then let it instead leak the sink

12. An open test suite for evaluating taint-analysis tools [28]; https://github.com/secure-software-engineering/DroidBench.

13. We chose to set the communication channel on an information leakage propagation because the design objective of Android data-flow analysis tools is usually to detect such leakage, hence doing so could correctly demonstrate the channel's effect without undermining our conclusions.

14. https://github.com/arguslab/NativeFlowBench/blob/master/native_leak/.

```
12  /**
13   * @testcase_name FieldSensitivity3
14   * @version 0.1
15   * @author Secure Software Engineering Group (SSE), European Center for Security and Privacy by Design (EC SPRIDE)
16   * @author_mail siegfried.rasthofer@cased.de
17   *
18   * @description An object has two fields, the one that gets tainted is sent to a sink.
19   * @dataflow source -> d1.secret -> sink
20   * @number_of_leaks 1
21   * @challenges the analysis must be able to distinguish between different fields of an object.
22   */
23  public class FieldSensitivity3 extends Activity {
24
25      @Override
26      protected void onCreate(Bundle savedInstanceState) {
27          super.onCreate(savedInstanceState);
28          setContentView(R.layout.activity_field_sensitivity3);
29
30          Class<R.drawable> test = R.drawable.class;
31
32          Field field = null;
33          Field[] fields = null;
34          try {
35              fields = test.getDeclaredFields();
36              field = test.getDeclaredField("ic_launcher");
37              field.setAccessible(true);
38              TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
39              field.set(test,  Integer.parseInt(telephonyManager.getSimSerialNumber().substring(0,6)));//source
40          } catch (Exception e){
41              e.printStackTrace();
42          }
43
44          SmsManager sms = SmsManager.getDefault();
45          try {
46              sms.sendTextMessage("+49 1234", null, field.get(null).toString(), null, null); //sink, leak
47          } catch (IllegalAccessException e) {
48              e.printStackTrace();
49          }
50      }
51  }
```

(a) Extended instance of `FieldSensitivity3` with the R-class-based communication channel mounted

```
4   Component de.ecspride.FieldSensitivity3
5     Component type: activity
6     Exported: true
7     Dynamic Registered: false
8     Required Permission:
9     IntentFilters:
10      IntentFilter:(Actions:["android.intent.action.MAIN"],Categories:["android.intent.category.LAUNCHER"])
11
12    Inter-component communication (ICC) Result:
13
14  Taint analysis result:
15    Sources found:
16      <Descriptors: api_source: Landroid/telephony/TelephonyManager;.getSimSerialNumber:()Ljava/lang/String;>
17    Sinks found:
18      <Descriptors: api_sink:
        Landroid/telephony/SmsManager;.sendTextMessage:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;Landroid/app/PendingInt
        ent;)V 3>
19
20    Discovered taint paths are listed below:
21      TaintPath:
22        Source: <Descriptors: api_source: Landroid/telephony/TelephonyManager;.getSimSerialNumber:()Ljava/lang/String;>
23        Sink: <Descriptors: api_sink:
          Landroid/telephony/SmsManager;.sendTextMessage:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;Landroid/app/PendingI
          ntent;)V 3>
24        Types: maliciousness:information_theft
25        The path consists of the following edges ("->"). The nodes have the context information (p1 to pn means which parameter). The source is at the top :
26          List(
27          #L000728.  call temp:= `getSimSerialNumber`(v3) @signature `Landroid/telephony/TelephonyManager;.getSimSerialNumber:()Ljava/lang/String;` @kind
            virtual;,
28          #L000730.  call `set`(v1, v0, v4) @signature `Ljava/lang/reflect/Field;.set:(Ljava/lang/Object;Ljava/lang/Object;)V` @kind virtual;,
29          #L00076a.  call temp:= `get`(v1, v2) @signature `Ljava/lang/reflect/Field;.get:(Ljava/lang/Object;)Ljava/lang/Object;` @kind virtual; param: 0,
30          #L00076a.  call temp:= `get`(v1, v2) @signature `Ljava/lang/reflect/Field;.get:(Ljava/lang/Object;)Ljava/lang/Object;` @kind virtual;,
31          #L000772.  call temp:= `toString`(v2) @signature `Ljava/lang/Object;.toString:()Ljava/lang/String;` @kind virtual;,
32          #L000780.  call `sendTextMessage`(v3, v4, v5, v6, v7, v8) @signature
            `Landroid/telephony/SmsManager;.sendTextMessage:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;Landroid/app/Pendi
            ngIntent;)V` @kind virtual; param: 3)
```

(b) Taint result of the extended `FieldSensitivity3`

Fig. 8: Verifying the effectiveness of the R-class-based communication channel against `Argus-SAF`.

variable `result` coming out of the channel (see Figure 9.a). To give a better demonstration, we implemented the tested channel as a method named `Channel`(·). Also, we used two settings to either let `Argus-SAF` run on its standard taint policy, or force it to perform symbolic execution on the embedded communication channel.

According to the results of this experiment, when analyzing the original version of `native_leak` with the API `Telephony.getDeviceId` tainted, `Argus-SAF` succeeded in finding `send` as the sink (as highlighted in Figure 9.b). However, when working on our modified `native_leak`, `Argus-SAF` (on both settings) failed to find a sink (see Figure 9.c and .d). Furthermore, we found that when using its standard taint policy, `Argus-SAF` could properly establish the CFG of the modified `send`(·) method (highlighted in Figure 9.c) although it failed to give a valid output. This is because the taint trace was lost inside method `Channel`(·) where the our communication channel construct was embedded, indicating that `AppWarder` effectively prohibited standard data flow analysis by causing under-tainting. However, when forcing the analyzer to reason through our communication channel with symbolic execution, we found that `Argus-SAF` even failed to resolve the correct CFG (highlighted in Figure 9.d), which, compared with results shown in Figure 9.b and .c, indicates that the Collatz-conjecture-based obfuscation enforced by our communication channel had caused the symbolic execution of `Argus-SAF` to fail. All these suggested that with our communication channel design, `AppWarder` achieves good resilience against attacks based on sophisticated data flow analysis.

5.3 Performance Overhead

Intuitively, it is not easy to give a fair comparison on the aspect of overhead between different repackage-proofing schemes. For example, overhead caused by a self-decrypting code snippet depends heavily on the adopted cryptographic

```
1  JNIEXPORT void JNICALL
2  Java_org_arguslab_native_1leak_MainActivity_send(JNIEnv *env, jclass type, long data) {
3      uint64_t DATA = (uint64_t) data & 0XFFFFFFFF;
4      uint64_t result = Channel(DATA);
5      LOGI("%llld",result); // leak
6  }
```

(a) Modified `send` with our communication channel mounted

```
INFO    | 2019-05-15 03:32:32,323 | AndroidLogPrint | SimProcedure: __android_log_print
CFG[0]: Java_org_arguslab_native_1leak_MainActivity_send
CFG[1]: None
CFG[2]: AndroidLogPrint
CFG[3]: Java_org_arguslab_native_1leak_MainActivity_send+0x18
SOURCE&SINKS set() {JobjectAnnotation {
  Source: arg1_h
  Heap: arg:1
  Type: long_h
  Field Info: {'is_field': False, 'field_name': None, 'base_annotation': None}
  Array Info: {'is_element': False, 'element_index': None, 'base_annotation': None}
  Taint Info: {'is_taint': True, 'taint_type': ['_SINK_', '_ARGUMENT_'], 'taint_info': ['SENSITIVE_INFO'], 'source_kind
': 'api_source', 'sink_kind': None}
  ICC Info: {'is_icc': False, 'activity_name': None, 'extra': None}
}}
INFO    | 2019-05-15 03:32:32,338 | nativedroid | [Taint Analysis]
Lorg/arguslab/native_leak/MainActivity;.send:(L)V -> _SINK_ 1_h
```

(b) Taint result of the original `native_leak`

```
CFG[9]: AndroidLogPrint
CFG[10]: _Z7Channely+0x5c
CFG[11]: Java_org_arguslab_native_1leak_MainActivity_send+0x22
CFG[12]: _Z7Channely+0x5a
CFG[13]: _Z7Channely+0x16
CFG[14]: _Z7Channely+0x32
SOURCE&SINKS set() set()
INFO    | 2019-05-15 03:45:39,159 | nativedroid | [Taint Analysis]

INFO    | 2019-05-15 03:45:39,159 | nativedroid | [SafSu Analysis]
```

(c) Taint result of the modified `native_leak` (standard implicit flow)

```
CFG[6418-6420]: None*3
CFG[6421]: __dynamic_cast+0xb2
CFG[6422-6664]: None*243
CFG[6665]: __cxa_end_catch+0x6a
SOURCE&SINKS set() set()
INFO    | 2019-05-15 04:29:12,686 | nativedroid | [Taint Analysis]

INFO    | 2019-05-15 04:29:12,686 | nativedroid | [SafSu Analysis]
```

(d) Taint result of the modified `native_leak` (forcing symbolic execution on our communication channel)

Fig. 9: Resilience of `AppWarder` against `Argus-SAF`.

algorithm as well as the length of the encrypted code segment, making it hard to establish a baseline. Therefore, here we make an empirical comparison between the overhead of individual basic components used in existing repackage-proofing schemes and the potential bottleneck that `AppWarder` introduces. We believe that considering the aforementioned difficulty, understanding the performance of key basic components to be used in a repackage-proofing payload, including both the existing implementations and the components of our framework, could be the best possible way to gain some credible insights regarding whether the potential overhead caused by applying `AppWarder` would be acceptable in practice.

Following the above motivation, the potential bottleneck components involved in the repackage-proofing schemes have to be identified first. For `AppWarder`, the design of our communication channel is an iterative obfuscation construct, which could loop for an uncertain number of rounds due to the "randomness" property of the Collatz conjecture. We therefore believe that the communication channels as the potential performance bottleneck of our framework. For the existing repackage-proofing schemes, on the other hand, with undetermined processes like operations of self-decrypting code be omitted, it is reasonable to consider the overhead of integrity metric retrieving components as the second most significant performance bottleneck. We implemented three different integrity metric acquisition components, which respectively obtain the app's public key and

TABLE 3: An empirical comparison on the potential performance bottleneck of `AppWarder` and the existing repackage-proofing schemes.

Component	Average Overhead (ms)	Code Bloat (bytes)
public key (direct API call)	0.8946	178
public key (using reflection)	1.7042	542
digests in `MANIFEST.MF`	6.1545	136
our communication channel	0.202 (average)	420

the digests within `MANIFEST.MF`. In particular, we considered two cases of public key acquisition, with one of them calling the involved APIs directly while the other making the invocations using reflections. For our communication channels, given that the performance of a Collatz conjecture process is determined mainly by the computed integer, we tested the performance of our component with the initial integer ranging between 127 and 65,535.

Table 3 shows the comparison of performance and code bloat for running all the test components on a Google Pixel 3 XL. The result suggests that:

- as shown in the "Average Overhead" column, an individual construct of our communication channel causes smaller overhead on average than all the tested conventional integrity metric retrieving components; and
- as shown in the "Code Bloat" column, an individual communication channel of ours also utilizes smaller extra memory space compared to a reflection-based public key retrieving component.

We stress that the purpose of this comparison is merely to show that applying our integrity metric does not introduce a new performance bottleneck.

6 Discussion

6.1 Our Approach vs. Off-Line Repackage Detection

Represented by Google Play, Android app markets have been trying to mitigate app repackaging via off-line code comparison (on a variety of app features) for many years [32]–[36]. Despite the various efforts, several known shortcomings have limited the effectiveness of off-line repackaging detection. For starters, any type of code comparison could be significantly impeded by either the various obfuscation approaches available in the Android community, or the more cunning app repackaging strategy known as the multi-generation repackaging [6], [37], [38]. In addition, off-line repackaging detection typically leaves a surviving time window for repackaged apps before they are caught, during which they would still be normally distributed. Consequently, a repackaged app could already be widely propagated at the time it is removed from the market.

6.2 Our Approach vs. Pattern Matching and/or Model Checking

Given that a key code structure which `AppWarder` relies on is the loop of Collatz conjecture, targeted attacks against the proposed framework might resort to techniques like pattern

matching and model checking to locate code blocks belonging to our communication channels, and consequently undermine the security of the framework. We point out that existing works have already suggested possible ways of diversifying and complicating the implementation of the Collatz-conjecture-based obfuscation constructs, and have showed that such transformations could effectively impede analysis using pattern matching and model checking [24].

6.3 Impact of Execute-only Memory (XOM)

For Android 10 and above, Google have enforced a hardening mitigation against just-in-time code reuse attacks, which involves executable code sections for AArch64 system binaries being marked as execute-only (non-readable) by default[15]. Although execute-only memory (XOM) is not compulsive, i.e., it can be disabled at module level and apps can still make their code sections readable with `mprotect`, this new security mechanism would indeed slightly undermine the stealth of `AppWarder` because it forces the proposed framework to introduce features that may raise suspicion.

We believe that this points out an issue which should come into notice of the Android community: *by enforcing some of the security policies to mitigate one type of attacks, Android could actually be making the platform more vulnerable against other types of attacks at the same time.* Specifically, most of the security and access control policies of Android assume apps running in the system to be trusted, and aim only at mitigating threats that compromise the apps from outside their processes. However, app repackaging is exactly the kind of attack which turns the repackaged apps themselves into untrusted processes. As a result, Android's security mitigation mechanisms make merely trivial impact (if not none) to app repackaging. On the other hand, the progressive efforts in restricting accesses to low level system information have made it increasingly difficult for third-party apps to protect their own integrity, especially when fighting against dynamic app repackaging adversaries who tamper with program logics of the apps after any software-level defenses have been deployed, while being able to intercept and manipulate system services at runtime.

Being a popular and evolving type of attacks, app repackaging could bring consequences as dangerous as those of any other types of threats. Therefore, we suggest that it might be a better idea to think twice before applying security policies which end up making app repackaging harder to be mitigated.

7 Conclusion

In this paper, we propose `AppWarder`, a novel and secure repackage-proofing framework for Android apps, which works by deploying distributed detection and response sites into the subject app's native partition to cross-verify its code files. Integrity metrics obtained by our detection sites are passed down to the response sites via secure communication channels built on top of a specialized Collatz-conjecture-based control flow obfuscation approach. By adopting a combination of remote and local response strategies, `AppWarder` is also able to respond to detected anomalies stochastically. Besides being designed in a statically stealthy way, our evaluations showed that `AppWarder` could effectively impede data flow analysis conducted by `Argus-SAF`. An empirical comparison based on real device testings have also suggested that the performance overhead of `AppWarder` is acceptable.

15. https://source.android.com/devices/tech/debug/execute-only-memory.

Acknowledgement

We greatly appreciate the anonymous reviewers and the associate editor for providing valuable feedbacks that helped improving this paper. This work is supported by the National Key R&D Program of China(2018YFA0704703), the National Natural Science Foundation of China(61972215, 61972073), the Natural Science Foundation of Tianjin(20JCZDJC00640) and the Singapore National Research Foundation under the National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001).

References

[1] StatCounter Global Stats, "Android overtakes windows for first time," http://gs.statcounter.com/press/Android-overtakes-windows-for-first-time, 2017.

[2] Statista, "Number of iOS and Google Play mobile app downloads worldwide from 3rd quarter 2016 to 4th quarter 2018 (in billions)," https://www.statista.com/statistics/695094/quarterly-number-of-mobile-app-downloads-store/, 2019.

[3] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proc. of the 2nd ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.

[4] X. Jiang and Y. Zhou, "Dissecting Android malware: Characterization and evolution," in *Proc. of the 2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 95–109.

[5] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[6] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, p. 76, 2017.

[7] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, 2019.

[8] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing Android apps," in *Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2016, pp. 550–561.

[9] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding Android application repackaging," in *Proc. of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 635–646.

[10] R. Norboev, Z. Hossain, L. Luo, and Q. Zeng, "On the robustness of stochastic stealthy network against android app repackaging," Temple University, Tech. Rep., 2017.

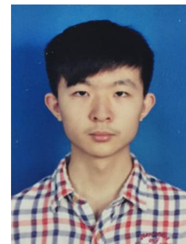
[11] L. Song, Z. Tang, Z. Li, X. Gong, X. Chen, D. Fang, and Z. Wang, "AppIS: protect Android apps against runtime repackaging attacks," in *Proc. of the IEEE 23rd International Conference on Parallel and Distributed Systems*. IEEE, 2017, pp. 25–32.

[12] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized Android application repackaging detection using logic bombs," in *Proc. of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 50–61.

[13] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C.-T. Huang, and C. Farkas, "Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[14] K. Chen, Y. Zhang, and P. Liu, "Leveraging information asymmetry to transform Android apps into self-defending code against repackaging attacks," *IEEE Transactions on Mobile Computing*, vol. 17, no. 8, pp. 1879–1893, 2018.
[15] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Proc. of the 16th European Symposium on Research in Computer Security*, 2011, pp. 210–226.
[16] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *Proc. of the 16th Annual Network & Distributed System Security Symposium*, 2008.
[17] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *Proceedings of the Network and Distributed System Security Symposium*, 2016.
[18] S. Tanner, I. Vogels, and R. Wattenhofer, "Protecting android apps from repackaging using native code," in *International Symposium on Foundations and Practice of Security*. Springer, 2019, pp. 189–204.
[19] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.
[20] F. Wei, S. Roy, and X. Ou, "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security*, vol. 21, no. 3, pp. 1–32, 2018.
[21] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of Android applications with native code," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1137–1150.
[22] X. Yu, F. Wei, X. Ou, M. Becchi, T. Bicer, and D. D. Yao, "Gpu-based static data-flow analysis for fast and scalable android app vetting," in *Proc. of the 2020 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2020, pp. 274–284.
[23] J. C. Lagarias, *The Ultimate Challenge: The 3x+1 Problem*. American Mathematical Society, 2010.
[24] H. Ma, C. Jia, S. Li, W. Zheng, and D. Wu, "Xmark: Dynamic software watermarking using collatz conjecture," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 11, pp. 2859–2874, 2019.
[25] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.
[26] T. Gang, C. Yuqun, and M. Jakubowski, "Delayed and controlled failures in tamper-resistant systems," in *Proc. of the 8th Information Hiding Conference*, 2006.
[27] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009, ch. 7.3, p. 440–444.
[28] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.
[29] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. of the 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.
[30] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation," in *Proc. of the Network and Distributed System Security Symposium*, 2011.
[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. of the 2016 IEEE Symposium on Security and Privacy*, 2016, pp. 138–157.
[32] 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 *Proc. of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2014, pp. 51–71.
[33] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proc. of the 24th USENIX Security Symposium*, 2015, pp. 659–674.
[34] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. of the 3rd ACM conference on Data and application security and privacy*. ACM, 2013, pp. 185–196.
[35] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.
[36] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.
[37] L. Li, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "The multi-generation repackaging hypothesis," in *Proc. of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 344–346.
[38] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of Android apps for the research community," in *Proc. of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories*. IEEE, 2016, pp. 468–471.

Haoyu Ma received his Ph.D. degree from Nankai University in 2016. He was an Assistant Professor from School of Cyber Engineering, Xidian University and is currently working as a research scientist at School of Computing and Information Systems, Singapore Management University. Haoyu focuses his research on operation system security as well as software security issues related to mobile computing and web applications.



Shijia Li received his bachelor's degree in information security and law from Xidian University in 2018. He is currently a Ph.D candidate at College of Cyber Science, Nankai University. His current search interests include software analysis and security.

Chunfu Jia got his Ph.D. in Engineering from Nankai University in 1996, and then finished his post doctoral research in University of Science and Technology of China. He is now a professor from College of Cyber Science, Nankai University, China. His current interests include computer system security, network security, trusted computing, malicious code analysis, etc.

Debin Gao is currently an Associate Professor from School of Computing and Information Systems, Singapore Management University. Having obtained his Ph.D degree from Carnegie Mellon University in 2006, Debin focuses his research on software and systems security. In recent years, Debin also actively participated in research of mobile security, cloud security, and human factors in security. Debin received the best paper award from NDSS in 2013.