Computer Science Review 39 (2021) 100365

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Review article

A survey of android application and malware hardening

Vikas Sihag [a,b,*], Manu Vardhan [b], Pradeep Singh [b]

[a] Sardar Patel University of Police, Security and Criminal Justice, Jodhpur, India
[b] National Institute of Technology, Raipur, India

ARTICLE INFO

Article history:
Received 22 June 2020
Received in revised form 25 December 2020
Accepted 13 January 2021
Available online 22 January 2021

Keywords:
Android
Malware analysis
Code obfuscation
Evasion techniques

ABSTRACT

In the age of increasing mobile and smart connectivity, malware poses an ever evolving threat to individuals, societies and nations. Anti-malware companies are often the first and only line of defense for mobile users. Driven by economic benefits, quantity and complexity of Android malware are increasing, thus making them difficult to detect. Malware authors employ multiple techniques (e.g. code obfuscation, packaging and encryption) to evade static analysis (signature based) and dynamic analysis (behavior based) detection methods. In this article, we present an overview of Android and its state of the art security services. We then present an exhaustive and analytic taxonomy of Android malware hardening techniques available in the literature. Furthermore, we review and analyze the code obfuscation and preventive techniques used by malware to evade detection. Hardening mechanisms are also popular amongst application developers to fortify against reverse engineering. Based on our in-depth survey, we highlight the issues related to them and manifest future directions. We believe the need to examine the effectiveness and efficiency of hardening techniques and their combination.

Contents

* Corresponding author at: Sardar Patel University of Police, Security and Criminal Justice, Jodhpur, India.
E-mail addresses: vikas.sihag@policeuniversity.ac.in (V. Sihag), mvardhan.cs@nitrr.ac.in (M. Vardhan), psingh.cs@nitrr.ac.in (P. Singh).

https://doi.org/10.1016/j.cosrev.2021.100365
1574-0137/

1. Introduction

Smartphone's pervasive presence has offered new possibilities to life experiences, with its power to compute, sense, connect and to be mobile. Android OS since its release in 2008, has grown as the most preferred choice in the market with over 2.5 billion active devices worldwide and 74.13% share in December 2019 [1]. Android's success can be ascribed to its free open source code, which provides smartphone manufacturers with the liberty to transform their devices with pre-installed applications (aka apps) and customized user-interface for enriched customer experience. Google Play Store, Android's official application hosting service has over 2.57 million apps generating about 140 million USD [2]. The popularity of Android, its open environment and well established universal app distribution model accord to the creation of dangerous attack surfaces for threat actors targeting user's security and privacy [3].

Spotted in 2010, the first Android Malware was DroidSMS, which targeted users by subscribing premium SMS services. Since then multiple genres of malware have targeted Android ranging from downloaders to clickers, spyware to banking trojans and adware to ransomware. Recently CamScanner a popular document scanning app with more than 100 million downloads on Google Play store was identified to be infected with `AndroidOS.Necro.n` dropper, which once installed attempts to install a payload [4]. Recently, 983 cases of known vulnerabilities and 655 zero-days were found among the top 5000 free apps (each with 1M to 500M downloads).

The growing popularity of Android has brought the attention of developers adopting state of the art application hardening techniques like obfuscation and protection mechanisms as reflected in Fig. 1. By application hardening, we refer to enhancements of an application to deter tampering or reverse engineering. Malware researchers are propagating obfuscated and encrypted banking trojans, evading anti-malware scanners. They employ code obfuscation, encryption, dynamic loading and native code execution to circumvent Google Play protection [5–14]. App developers, on the other hand, are using them to prevent their source code and intellectual property from misuse. Both extremes, benign and malicious are fighting against app reversal. In computer science vocabulary, reverse engineering also known as back engineering is the process by which an object or executable file (`APK` archive for Android) is deconstructed to reveal its designs and architecture.

A typical malware scanner, extract features and characteristics of a target application, which are then used to identify its behavior and thus understand its internal working. Application features present information about *"What an application looks like"* and *"How an application behaves?"*. Features extracted using static analysis gives insight into the former and features extracted using dynamic analysis answers the latter. *Static analysis* investigates malware without the real code or instructions being executed. It provide basic information about app functionality and collect technical indicators, which may include file name, MD5 hashes, file type, file size, API calls, libraries, etc. *Dynamic analysis* executes and monitors an application, to track its behavior, understand features and identify indicators that can be used as detection signatures. Dynamic analysis technical indicators can include the location of files, registry keys, domain names, IP addresses and dynamic libraries [15].

The goal of this survey is to review and classify the existing Android application hardening techniques. They are categorized based on their target and impact. In particular, we examine mechanisms targeting application `APK` file, application code or application execution environment. This survey would be beneficial to developers and researchers, to understand the current state of hardening techniques and their effectiveness.

Unlike previous reviews, this survey does not do a general overview of Android malware evolution and detection techniques [11,16,17] but in detail focuses on Android application hardening methods systematically. It differs from previous surveys as depicted in Table 1. Previous works either reviewed code obfuscation methods or evasion/preventive techniques. The works on obfuscation except a few were limited to popular techniques. This paper assesses the effectiveness of obfuscation methods and compares the state of the art tools. The existing literature on preventive or evasion techniques is constrained to a few methods (anti-debugging, anti-emulator). This article provides a comprehensive view of all preventive measures to the best of our knowledge. This study fills the gap by presenting an eagle-eye view of application hardening used by malware authors and developers.

Organization of the paper

The layout of the survey is illustrated in Fig. 2. We describe Android architecture (Section 2), application format, compilation and Android security framework. We elaborate application hardening techniques (Section 3) employed at various levels by both developers and malware authors. Furthermore, we assess effectiveness of obfuscation techniques (Section 4) and contrast state of the art hardening tools (Section 5). Subsequently we propose future research directions (Section 7) and conclude (Section 8).

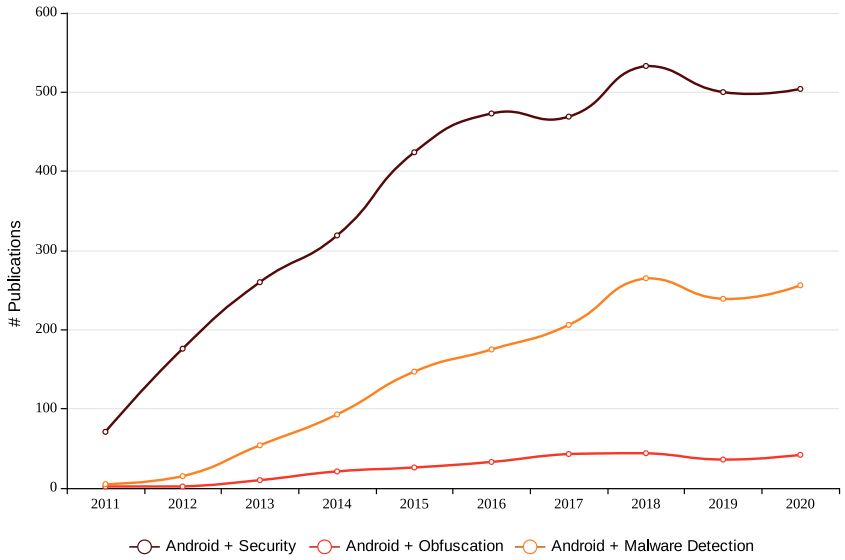

Fig. 1. The number of publications over the last decade related to keywords Android, security, obfuscation and malware detection. Note that the graph only accounts for publications having the desired keyword(s) in its title or abstract and belonging to the related field of research.

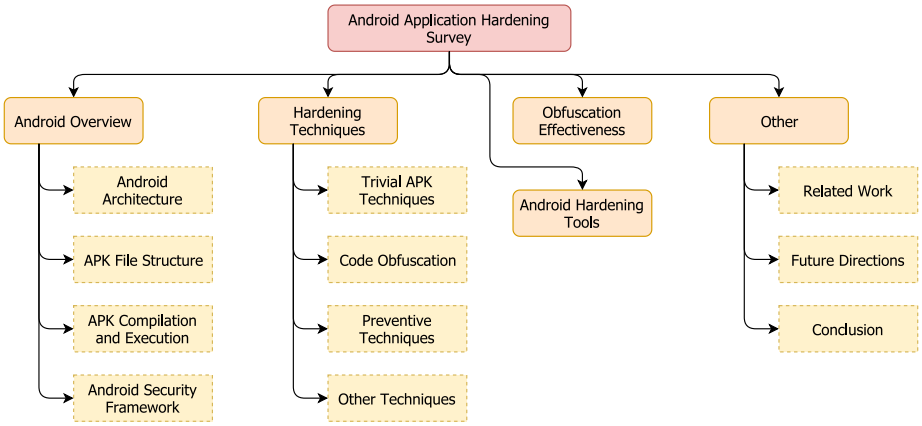

Fig. 2. Layout of the survey.

Table 1
Timeline comparison of surveys and coverage of topics in them. [✗ = little or no coverage]

Year	Surveys	Background	Obfuscation techniques	Preventive techniques	Assessment	Tools
2020	This article	Android ecosystem + security framework	Comprehensive coverage	Comprehensive coverage	Obfuscation + preventive	Comparative study
2019	Afianian et al. [16]	Android debugging	✗	Anti-debugging + sandbox evasion only	Preventive (two) only	✗
2018	Haupert et al. [11]	Threat model	Mention studies, ✗ details	Mention studies, ✗ details	Preventive only	✗
2017	Bulazel et al. [17]	PC, mobile + web platforms	✗	Comprehensive coverage	Preventive only	✗
2017	Tam et al. [18]	Android architecture + malware analysis	Mention studies, ✗ details	Anti-emulator + VM-aware overview	✗	✗
2016	Xu et al. [19]	Android architecture + Security framework	Repackaging only	✗	✗	✗
2016	Faruki et al. [7]	Android architecture	Comprehensive coverage	✗	Obfuscation only	Comparative analysis
2015	Maiorca et al. [20]	Android architecture + Dex file structure	Comprehensive coverage	✗	Obfuscation only	✗
2014	Apvrille et al. [21]	Malware threat + evolution	Real world analysis	✗	✗	Comparative analysis
2014	Freiling et al. [22]	Obfuscation + Metric Zoo	Limited methods	✗	Obfuscation only	✗

2. Android overview

Android was designed by the Open Handset Alliance(OHA), which is a consortium led by Google of companies such as Samsung, Sony, Intel and more to give services and deploy handsets with Android platform. With the release of the very first model of Android on Nov 5, 2007, versions are released beneath a codename predicated on desserts, such as Apple Pie, Gingerbread, Marshmallow, etc.

In this section, we discuss the Android architecture, application file structure, application compilation, execution, and security framework.

2.1. Android architecture

Android is an open-source software stack of interfaces, with each layer, and the corresponding elements within each layer, tightly integrated and carefully tuned to provide the optimal application execution and development environment. The interfaces as depicted in Fig. 3 includes a Linux Kernel, set of libraries, runtime environment, API framework and applications.

Linux kernel

The foundation of the Android platform is the Linux kernel customized for smart devices with power, memory and computational constraints. For instance, the Android Runtime (ART) relies on the Linux kernel for inherent functionalities like threading and low-level memory control. Using a Linux kernel allows Android to take advantage of its key security features and allows device manufacturers to develop hardware drivers for a well-known kernel.

Hardware Abstraction Layer (HAL)

On top of Linux kernel, HAL provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. It consists of library modules, each of which implements an interface for a specific hardware element, such as Bluetooth or camera module.

Android Run Time (ART)

For Android version 5.0 or higher, each app runs its process with its instance of the Android Runtime (ART). ART converts the application's DEX bytecode into native code at install time. DEX is a Dalvik EXecutable bytecode format to execute on dalvik virtual machine for memory-constrained Android devices. ART runs multiple virtual machines (VMs) each with a DEX file executing on it. Ahead-Of-Time (AOT), Just-In-Time (JIT) compilation, Optimized garbage collection (GC), compact machine code generation and better debugging support are some of the considerable features of ART. Dalvik runtime was a virtual environment used before Android version 5.0. This change has negatively affected analysis frameworks.

Android core libraries

Android contains several Core Runtime Libraries which offer Java programming language functionality, such as database access, interface construction and graphics rendering. Some of the leading Android cored libraries open to developers are: `android.app` for access to application model, `android.database` for content providers data access, `android.opengl` for graphics rendering, `android.os` for access to OS services like inter-process communication, `android.text` to manage text display, `android.view` for building user interface, `android.widget` to access prebuilt user interface widgets and `android.webkit` to access web browsing capabilities.

Native C/C++ libraries

Most key components and utilities of Android system like ART and HAL are created from native codes that include C and C++ compatible libraries. Android provides access to some of these Native C/C++ Libraries using Java APIs. Native platform libraries can directly be accessed from native code using Android NDK, which allows implementing parts of an app in native code, using languages such as C and C++.

Java API framework

Features of Android are available for developers to write apps easily and quickly. It includes APIs to design UI, work with databases, handle user interaction, etc. APIs are grouped into modular system services. (*a*) *Content Providers* to access or share data to or from other apps; (*b*) *View System* for building UI including lists, grids and even internet browsers; and (*c*) Various *Managers* for accessing location, network status, resources like graphics, displaying notification and managing app activities.

System apps

Android contains a set of preinstalled System Apps to ensure minimum functionalities of SMS, internet browsing, contact management, calendar, music and more. These system apps provide vital capabilities that developers can access from their app, for instance sharing a message by system messaging app.

2.2. APK file structure

Android app is a zip archive with `.apk` file extension. It generally contains files and folders required for the application as depicted in Fig. 4. The purpose of them is listed below.

lib

It contains `.so` libraries as the code compiled for platforms. The code for each platform (like `armeabi, x86, x86 64`) is stored in a subdirectory. While not mandatory, programs usually have a `lib` directory

META-INF

It contains metadata information, which also includes signature and certificate information used for integrity and identity validation.

– *CERT.RSA*

It is the certificate of the app. An APK file must be digitally signed with a certificate whose private key is owned by the creator of the request in order to be accepted for download. As a trustworthy certificate authority is not required to sign the certificate, it is usually not done.

– *CERT.SF*

It lists application resources with their SHA-1 hashes.

– *MANIFEST.MF*

It is the application manifest file.

res

directory is responsible for storing raw resource files (such as images and audio files), which are later mapped to `.R` files.

Assets

directory is res like and used in the APK to store external resources (e.g., audio, images, and even executable exploits). Developers can build arbitrary folder hierarchy, unlike the res directory.

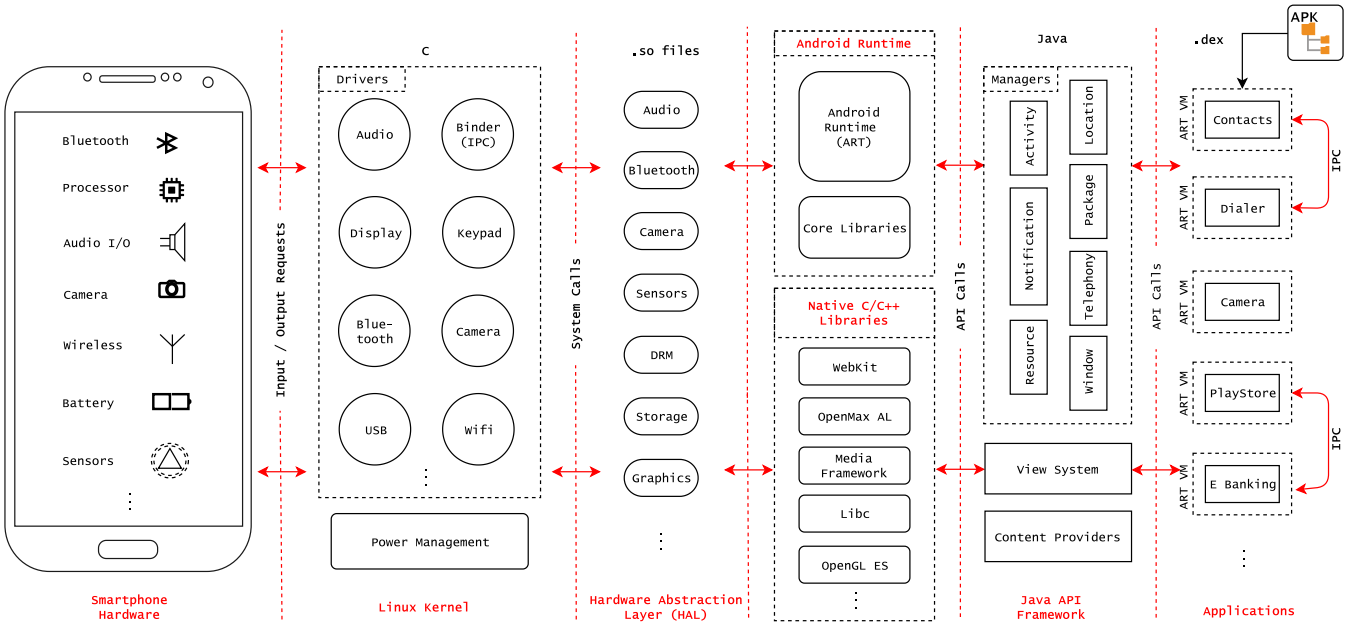

Fig. 3. Android architecture.

AndroidManifest.xml

stores the configuration information like name, version, required permissions, and components. It is responsible for protecting the system by specifying permissions to access any protected sections.

classes.dex

contains all the information about the classes in Dalvik Executable bytecode format. The data is organized in a way the Dalvik virtual machine can understand and execute. It contains vital information for app reversal and static analysis.

resources.arsc

provides precompiled application resources and is used to record the relationship between the resource files and related resource ID and can be leveraged to locate specific resources.

2.3. APK compilation and execution

Applications in Android are developed in Java. They are built into the corresponding `.class` files with the `javac` compiler. The `.class` files contain Java Bytecode, which is not directly executable on an Android device. Rather, Android has a distinct machine code format called Dalvik Bytecode. Fig. 5 gives an overview of application compilation process. Java `.class` files along with other `.jar` library files are forwarded to dex converter to convert into a single `classes.dex` file. Code listings 1 and 2 illustrates a sample java source code and its corresponding dalvik bytecode respectively.

```
1 public MainActivity() {
2   super();
3   currentPosition = 0;
4 }
```

Listing 1: Java source code

```
1 0x0000: iput-object v1, v0, Lcom/abc/myapp/
    MainActivity;com.abc.myapp.MainActivity$2.this$0
2 0x0002: invoke-direct {v0}, void java.lang.Object.<
    init>()
3 0x0005: return-void
```

Listing 2: Corresponding Dalvik bytecode

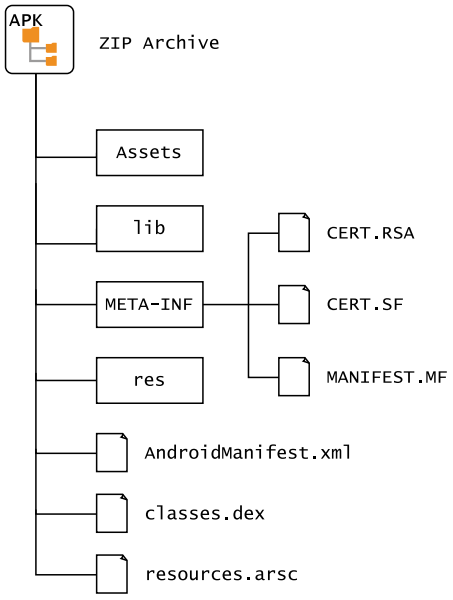

Fig. 4. APK file structure.

After it has been created, classes.dex file with compiled resources and shared object (`.so`) files containing native code are then compressed by the ApkBuilder tool into an Android PacKage (APK) file. For authenticating an APK for distribution, it is signed using `jarsigner` tool followed by `zipalign`.

Application execution in Android is a bit different from execution in a regular OS. Each application runs as a separate process in its own Dalvik/ART VM. Before executing an app, its UID, package names, entry point classes, required permissions and app components are extracted from the `AndroidManifest.xml`. Android at its core runs a process called `zygote` at startup after init. The zygote is a half started process with memory space and required core libraries, but without any of the code. As loading a new process is memory is system intensive, a copy of zygote is created using a fork system call to launch the desired app.

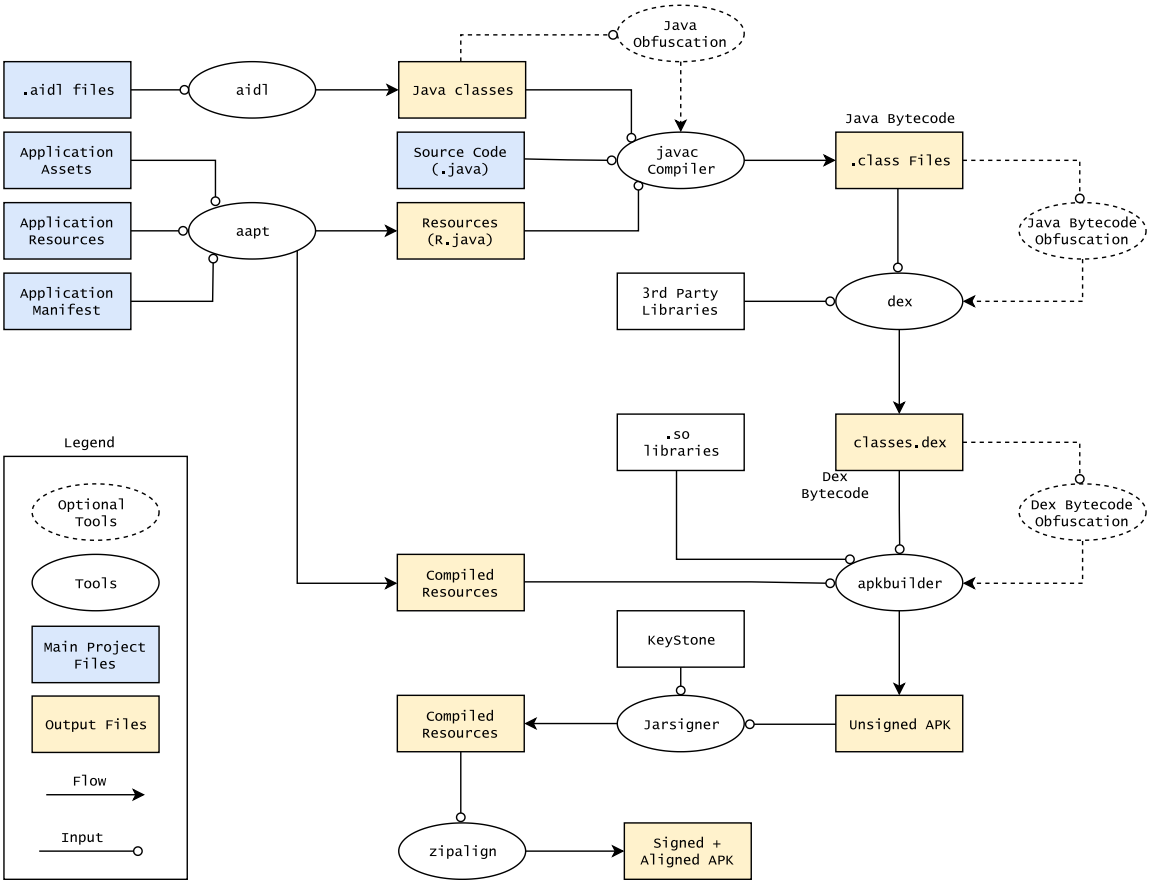

Fig. 5. APK compilation flowchart.

In Android Runtime (ART) during installation, the application byte code in `classes.dex` zipped inside APK is extracted and converted using the `dex2oat` tool to Executable and Linkable Files (ELF) shared objects which contain both DEX and native code. Generation of native code during installation is known as *Ahead Of Time (AOT)* compilation. The converted file is stored in `/data/dalvik-cache/...` path, which contains package name in the path to prevent overwriting. ELF (also known as OAT format) OAT format is a machine code that is specific to the CPU of the Android device, later mapped to the process memory. In Dalvik environment, which is the precursor of ART, an optimized version of DEX file called Optimized DEX (ODEX) file is generated. *Just In Time (JIT)* compilation is used to execute ODEX file. Fig. 6 illustrates comparative outline of APK execution in Dalvik and ART architecture in Android. Apps in ART runtime executes faster and requires less execution memory as compared to Dalvik runtime due to pregenerated machine code during installation. However, it takes longer for application installation in ART runtime.

2.4. Android security framework

Android platform has been designed with multiple security mechanisms. Android system has a hierarchical structure, and each layer has its own security mechanism, namely, traditional access control mechanism, a mechanism based on inspection of permission, sandbox mechanism, digital signature mechanism and encryption mechanism.

2.4.1. Linux security

Android's Linux kernel incorporates the access control mechanism of traditional Linux OS. Users access to resources and services is restricted based on user authentication and authorization. Android has a Mandatory Access Control (MAC) over traditional Discretionary Access Control(DAC). MAC manages access control decisions on all access attempts as part of the Linux Security Module (LSM) framework. However, in DAC owner of a particular resource controls access permissions associated with it. Android's access control policy greatly limits the potential damage to compromised machines and accounts. It ensures apps are running at the minimum privilege level [23]. This approach protects resource confidentiality and integrity.

2.4.2. Digital signature mechanism

Digital signature is a prerequisite for an app to be hosted on Google Play Store. It is generated by private key certificate allocated by a certifying authority thus ensuring the integrity of the app and authentication of its developer. Developers use verified certificates to validate app updation and other sibling apps developed by the same developer. If an APK is modified by an attacker, it needs to be re-signed for validation, which is only possible if the private key of the original publisher is known to the attacker.

2.4.3. Sandboxing

Android runs each app in an insulated kernel level sandbox environment with its memory and resources. This approach protects developer apps and system apps from malicious ones. As the sandbox is at the kernel, all above software (OS libraries, applications, framework) runs within the sandbox. Apps are restricted from interactions. If an app X tries to read application Y's resources, it is prevented because of the lack of user privileges. If an app has the permission of a resource (e.g., Contacts), the

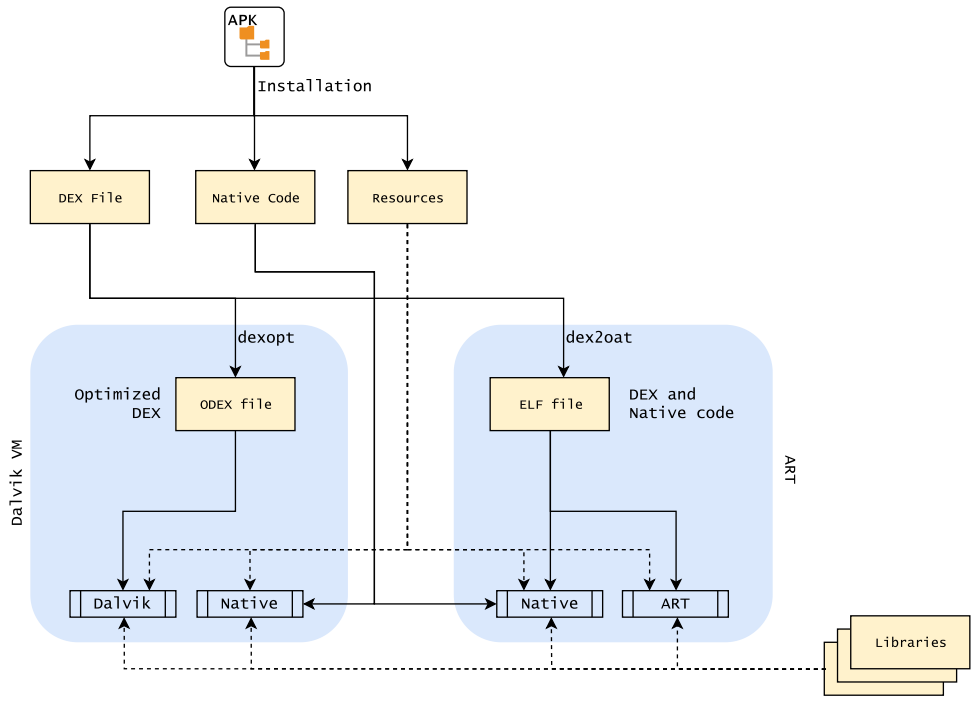

Fig. 6. APK execution process.

app process is assigned to the corresponding resource access id. As apps are digitally signed with the developer's private key, apps with same developer's certificate are assigned the same UID (i.e. sandbox) for resource and permission sharing. Thus malware authors with developer's key can design an app with the same certificate to access private resources of other sibling apps developed by the same developer.

2.4.4. Encryption

User data in Android device is encrypted using symmetric keys to provide confidentiality and authorized access. All disk write operations on user data follow after encryption and corresponding read operations precedes decryption. An unauthorized access to data, expose the real file content. Android employs two types of encryption methods to ensure confidentiality [24].

- *File Based Encryption (FBE):* Files in FBE are encrypted using different keys, which can be accessed independently. Using Direct Boot feature FBE enabled devices can boot straight to the lock screen without the requirement of user credentials.
- *Full Disk Encryption (FDE)*: In FDE user data partition /data is encrypted on block level using a single key generated by user credential. During boot time the encrypted device is detected and prompted for the password, which is then used to decrypt the user partition. FDE used 128 Advanced Encryption Standard (AES) with cipher-block chaining (CBC) for encryption.

2.4.5. App manifest

The Android application contains a mandatory manifest configuration file (AndroidManifest.xml). It specifies essential attributes about the application to Android OS. Some of these essential attributes are App's package name, its components, permissions requested and required set of hardware & software features. A sample manifest file is shown in listing 3. A manifest file values are configured at compile time and cannot be differed at execution. Following are the important characteristics specified in a manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android=
      "http://schemas.android.com/apk/res/android"
   package="com.example.helloworld"
   android:versionCode="1"
   android:versionName="1.0" >
   <uses-permission android:name=
      "android.permission.SEND_SMS" .../>
   <uses-feature android:name=
      "android.hardware.sensor.compass"
      android:required="true" .../>
   <application ... >
      <activity android:name=
        "com.example.helloworld.MainActivity" ... >
      </activity>
      <service android:name=".TestService" .../>
   </application>
   ...
</manifest>
```

Listing 3: A sample manifest file.

Package name. Once an APK is compiled, the package attribute represents an app's universally unique application ID, as it is used to identify an app in the system and Play Store. Listing 3 presents a sample package element name where `com.example.helloworld` is the action string.

App components. An app is composed of multiple components declared as corresponding XML elements in the manifest file are discussed below:

- *Activity*: An activity is a user inference component to interact with the user. Multiple activity components can be declared in the manifest file. Each activity is allocated a window on the screen for its UI, which can be of variable size and floating on other windows. One activity is specified as "main" activity displayed during app launching,

then may be followed by other activities based on user interaction. A sample activity "com.example.helloworld.MainActivity" declaration is shown in listing 3.

- *Service*: A Service component performs operations in the background without the user interface, even when the user switches to a different application. Application components also use services to interact with the app and perform inter-process communication (IPC).
- *Broadcast Receiver*: It is a component which allows apps to register for application or system generated events. Once registered, the receiver for an event is notified by Android runtime upon its occurrence. For example, application registered for BOOT_COMPLETED system event will be notified after completion boot process.
- *Content Provider*: Content provider is a standard interface to access structured data from within or outside an app. It is primarily intended to be used by other apps using provider client object for inter process communication and secure data access.

String and meta-data information in Android manifest file have also been used for malware detection [25,26].

2.4.6. Inter Component Communication (ICC)

ICC, a key feature of Android is an analogue of Inter Process Communication (IPC). It allows a component of an application to access data from another component within the same application, other application within the same device or an external service. Applications can depend upon others for third party services by borrowing services. For instance, a package delivery application can depend upon Google Map's for user's geolocation thus aiding developers.

Intents and intent filters. Activities, services and broadcast receivers, the core components of an application are invoked through an asynchronous messaging system called intents. An Intent object principally is a bundle containing information about action to be taken, data to act on and event that has been announced. Separate mechanisms exist for intent delivery to each type of component. Fig. 7 depicts starting of an activity using intent.

Intents can be divided into two types:

- *Explicit intents* are designed for the fixed target component. Only the component name in intent is considered to identify the target component.
- *Implicit intents* are not for a fixed target and are often used to actuate components in other applications.

Listing 4 declares an activity with an intent filter to receive an ACTION_SEND intent for text datatype.

```
1 <activity android:name="ShareActivity">
2   <intent-filter>
3     <action android:name="android.intent.action.SEND"
        />
4     <category android:name="android.intent.category.
        DEFAULT"/>
5     <data android:mimeType="text/plain"/>
6   </intent-filter>
7 </activity>
```

Listing 4: Sample activity with intent filter in manifest file to receive ACTION_SEND intent

The intent filtering mechanism cannot be relied on for security. While implicit intents are tested against intent filters for targeting components, the explicit intents can name the components as the target. An explicit intent should be used for starting a service and developers should prevent from declaring intent filter for service components. Using an implicit intent to start a service is a security hazard [27]. Intent filters have been used for malware detection. Refs. [28–30] have evaluated effectiveness of Android intents (explicit and implicit) alone or in combination with other features such as permission for identifying malicious applications.

2.4.7. Permission model

Android uses permission-based security model to restrict application's access using APIs to system resources. Requested permissions to access resources are specified using <uses-permission> tags in the manifest file as depicted in listing 3. Android permission model defines four access levels for permissions.

- *Normal permissions* are required by an app to access data or resources outside its sandbox to isolated application level features, but with negligible privacy or security risk. For example, ACCESS_NETWORK_STATE is normal permission.
- *Dangerous permissions* are higher risk permission that request exposure to user's private data or device control. Explicit consent from user during installation is required for them. For example, ability to access calendar and phone book are dangerous permissions.
- *Signature permissions* are used by developer to share resources among its sibling apps. It is used to access resources between apps signed by the same developer certificate. For example, INJECT_EVENTS allows an application to forcibly stop other applications.
- *SignatureOrSystem permissions* are required to change system setting and installation privilege. These are generally given to apps signed by the same developer certificate as of system image. For example, WRITE_SETTINGS allows an application to alter system settings.

3. Android application hardening

Application developers employ various hardening techniques to prevent analysis of their code. With the term *stealth*, we refer to modifications or enhancements employed by an Android application to make its structure and behavior inconspicuous, i.e. to deter tampering or reverse engineering.

Hardening technique such as obfuscation is a double-edged sword as it protects legit developers against code cloning as well the malware authors against a range of analysis engines [31]. For this survey, we used search engines and databases to identify high quality refereed articles from journals and conference papers. We employed keywords (such as Android obfuscation, hiding, hardening, protection etc.) and regular expression based search for literature identification. The research was focused to identify articles in the domain of Android application hardening. Correspondingly, the application hardening is classified into Trivial APK techniques, code obfuscation, preventive and other techniques. Categories and sub-categories are outlined in Fig. 8. *Trivial APK techniques* contains enhancements that modify the structure or packaging of an APK file. *Code obfuscation* in agreement with current literature contains obfuscation techniques that target source code or byte code. *Preventive techniques* focus on detecting and preventing application execution in a test(i.e. virtual, emulator, etc.) environment. We have also considered *Other techniques*, which consists of strengthening network interaction and resource centric obfuscations, where focus is on securing the application communication over the network to prevent eavesdropping and traffic analysis. And resource centric targets resource files an application requires for execution.

In the following subsections, existing Android application hardening approaches are detailed.

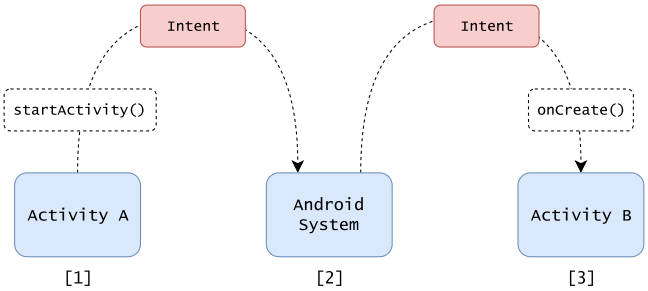

Fig. 7. Inter Component Communication using intent to start activity: [1] Activity A creates an Intent with an action description and passes it to startActivity(). [2] The Android System searches all apps for an intent filter that matches the intent. When a match is found, [3] the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

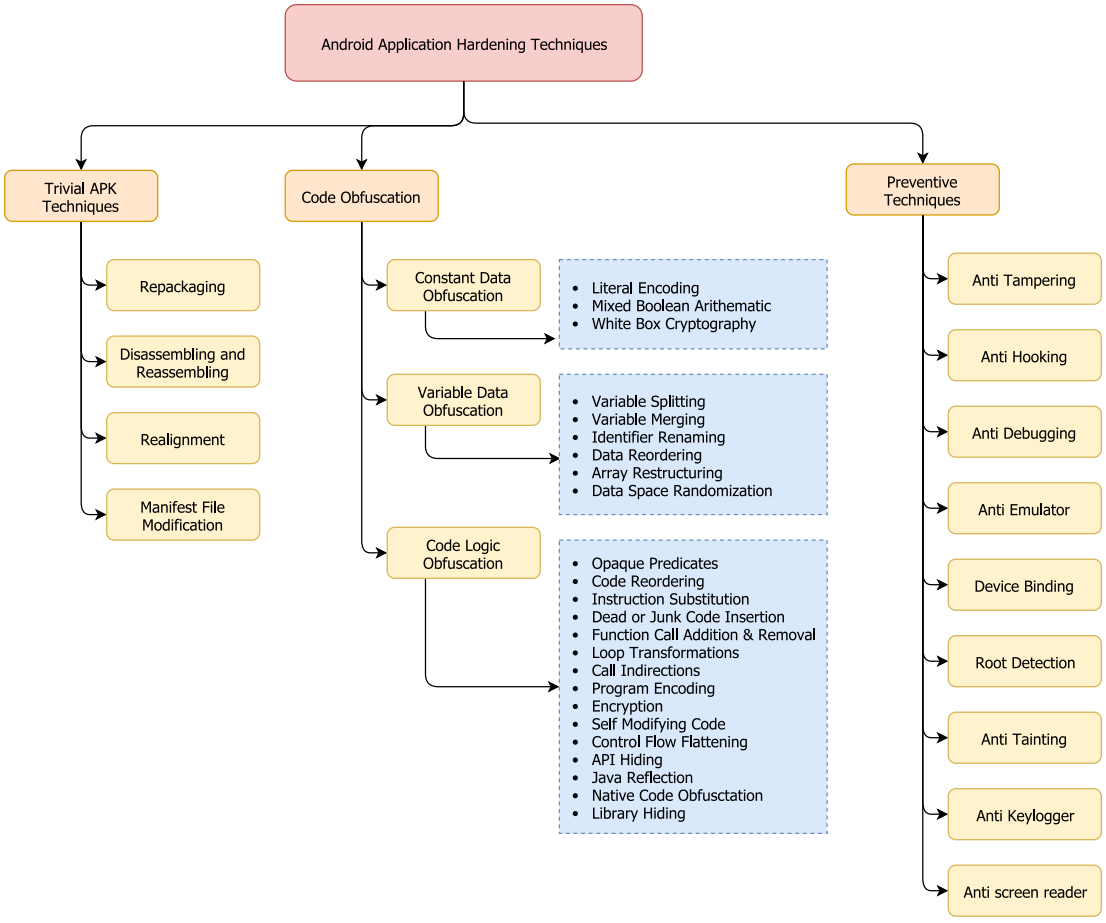

Fig. 8. A hierarchical view of existing malware hardening techniques for Android OS [8,11].

3.1. Trivial APK techniques

Trivial APK techniques are called so because they require less technical skills and are easier to apply. The purpose of these techniques is to evade application detection engines based on signatures of complete or specific segments of an application. They comprise of transformations which do not require code-level modifications but simple operations, like signing the APK file with a new signature.

Such transformations include unpackaging and repackaging the APK file, renaming application packages, reassembling the byte code [5,20].

3.1.1. Repackaging

Anti-malware scanners often rely on APK digests for malware identification. Repackaging includes unzipping the APK, adding junk code or resources and rezipping to a functionally identical APK. Malware authors often repackage popular applications to receive its purchase and advertisement profit [32–35]. Previous studies have shown that 86% of over 1200 malware families were repackaged to induce malicious payloads [36]. Popular applications are repackaged with malicious payloads in order to steal user information, make purchases, or send premium SMS [32,33,37].

As depicted in Fig. 9 repackaging includes decompilation of DEX file extracted from the APK using tool baksmali [38]. The decompiled smali code is then modified, recompiled and repacked. It is then self signed by the attacker to generated forged APK to be hosted on market places [39].

Recall from Fig. 5 that Android applications are signed by developer's certificate that will be lost after disassembling and

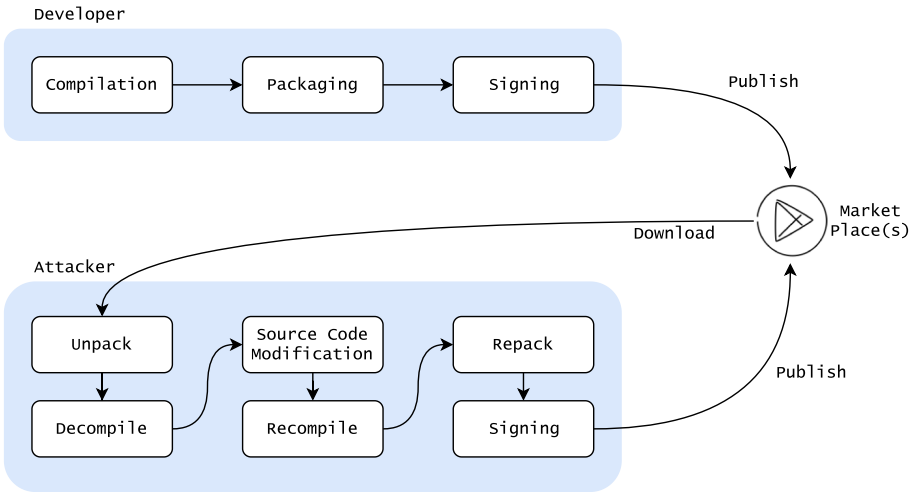

Fig. 9. Attackers uses repackaging to publish forged applications on market places.

reassembling. A repackaged APK can be distinguished by comparing its checksum to the original one [5,13]. Repackaging is a popular technique employed by malware authors to generate different instances of the same malicious APK [40,41].

3.1.2. Disassembling and reassembling

Disassembling and reassembling of the DEX code (i.e., `classes.dex`) residing in APK archive also creates a functionally equivalent application yet with a different digest like repackaging. Dalvik byte code is disassembled and reassembled with `apktool` to generate reordered byte code [14]. Application identification methods relying on code orders are likely to be ineffective against it [13]. Malware authors use this technique to generate malware instances with different file based and code order based signatures to circumvent detection engines.

3.1.3. Realignment

Recall from Fig. 5, `zipalign` tool is used to align signed applications. Zipalign is an alignment tool for APK file optimization. It particularly aligns uncompressed data such as media and raw files w.r.t. start of the file to be matched to 4-byte limits [13]. Realignment reduces RAM consumption during application execution [27]. Aligning an APK creates a functional equivalent but slightly different file. Digest of a realigned file is different from its original APK.

3.1.4. Manifest file modification

As explained in 2.4.5, each application contains AndroidManifest.xml file specifying application components, features and permissions [42]. During APK compilation the manifest is compiled into a binary XML file. This transformation modifies the XML file by altering permissions, intent filters and components. Code listing in 5 illustrates modification in manifest file of Plankton malware to evade detection as demonstrated in [5]. Package, activity and service are renamed to non-readable text. For it to work, components in byte code are required to be renamed.

A popular technique available in literature is *package renaming*, which is a subset of manifest file modification [13,14,43]. Android uses the package name to determine if an application has been installed or not. Usually, packages are defined by using hierarchy separated by dots. General package nomenclature followed is: `<com.companyname.applicationname>`. Listing 5 illustrates package renaming used by Plankton malware. It modifies package name to a predefined random string format.

```
1 <manifest ... package= "com.hDEWJu.oYlCvk.hFYkwc.
    FgDOHA.UPkmVF" ... >
2   <application android:label="@string/app_name"
      android:icon="@drawable/icon">
3     <activity android:label="@string/app_name"
        android:name= ".LncHMH" ... >
4     :
5     </activity>
6     <service android:name= "com.rawJbA.DKPTQc.aaMYse.
        QUivSk" ... />
7   </application>
```

Listing 5: Manifest file with renamed packages.

3.2. Code obfuscation

Code obfuscation is a popular and effective technology to make reverse engineering harder. Code obfuscation was introduced to prevent intellectual property violations of the source code [44]. But since then it also has been extensively employed by malware authors for evasion [45,46]. Malware authors use obfuscation techniques to conceal working and protect information such as strings, domain names and web addresses, which may be used as artifacts for detection.

The objective of code obfuscation is to make it harder to interpret the code. It transforms the given code into a functionally identical one of the original but difficult to follow [13,47]. The theoretical perspective of code obfuscation is well studied in [48–50] to define its limits. To make their apps more difficult for analysis, Android application developers use different obfuscation strategies. First defined by Barak et al. [48], an obfuscator is a program that transforms a given code or application into a new one satisfying functionality property (must be functionally equivalent), slowdown property (be polynomially slower than the original) and virtual black-box property (be as hard to analyze and reverse as a black-box version of the application) [51]. Code obfuscation methods accounted are applicable on java source code or dalvik byte code. We have also included existing methods which have been applied in other OS environments but can be applied on Android application source code. The prevailing code obfuscation methods [8] as outlined in Fig. 8 are discussed below.

3.2.1. Constant data obfuscation

As the name suggests, this technique targets constants data types. It includes methods whose objective is to obscure the constant values by different arithmetic, logical or cryptographic transformations.

Literal encoding. An easy way to hide a constant is to transform it into a function that generates the constant during runtime [52,53]. The literal encoding includes an invertible function f, a constant d as an input to f and storage for the output. The inverse function f^{-1} is required during runtime to generate the constant value d from the stored output. Basic literal encoding function induces low computation overheads [8]. Opaque expressions which always have a certain fixed value during program execution can also be used to encode constant value. For example $cos^2(x) + sin^2(x)$ is always equal to 1, regardless of the value of x. Therefore, the literal `int flag = 1;` can be encoded as shown in listing 6.

```
1 double x = Math.random()
2 double s = Math.sin(x);
3 double c = Math.cos(x);
4 int flag = s*s + c*c;
```

Listing 6: Literal encoding

Mixed boolean arithmetic. Mixed Boolean Arithmetic (MBA) technique for code obfuscation is suggested in [54]. MBA encodes data using arithmetic (addition, multiplication, etc.) and boolean operations (XOR, AND, OR). The resulting encoding relies on external inputs so that compiler optimization strategies do not deobfuscate it.

For example, a linear MBA expression as $A = (x \oplus y) + 2 \times (x \wedge y)$ which simplifies to $A = x + y$ can be used for encoding.

White box cryptography. An application processing encrypted data needs to use and manage keys, which are often used to secure a user's private and transactional information. If an application is reversed with keys exposed, the sensitive data managed by the application may be bared. An extreme form of code obfuscation is white-box cryptography, which applies obfuscation, mathematical transformations and encryption to securely store the secret keys in applications without hardware keys or third party entities [55,56]. The fundamental idea of white-box cryptography is to fuse the key with cipher logic (s-boxes, for example) such that the key no longer is found in the application code [53,57]. It aims to provide one-way cryptographic functions, which are easy to encrypt or decrypt but hard to reverse the key [58]. The first Data Encryption Standard (DES) and Advanced Encryption Standard (AES) based white box transformations were implemented by Chow et al. [55,59] along with Link and Neumann 2005 [60]; Bringer et al. 2006 [61]. The secret key can be embedded within S-boxes and T-boxes for DES and AES based white box transformation respectively. Dual-ciphers, linear encoding and perturbation to the cipher equations are techniques used to strengthen white-box cryptography [61–63].

3.2.2. Variable data obfuscation

Variable splitting. In order to make it difficult for analyst to identify variables, they can be split into multiple variables. During obfuscation a variable can be split into two or more variables and then be reconstructed from its split parts during execution [53]. A boolean variable for instance can be replaced by a boolean expression. A variable V can be split into k parts $v_1, v_2, \ldots, v_k$. As illustrated in code snippet 8, boolean variable x is concealed as XOR operation on x_1 and x_2 [64]. If $x = 1$, it can be split into $x_1 = 1$ & $x_2 = 0$ and reconstructed using XOR operation.

```
1 //before
2 boolean x=true;
```

Listing 7: Before variable spliting

```
1 boolean x1 = false;
2 boolean x2 = true;
3 boolean x = x1 ^ x2;
```

Listing 8: After variable splitting

Variable merging. It involves merging multiple variables into a single variable. Variables $v_1, v_2, \ldots, v_k$ can be merged into a single variable V. While merging the selected variables must be of same type [52]. For example two 8-bit variables $x_1 = 10101010$ and $x_2 = 11110000$ are merged into a 16-bit integer $x = 1010101011110000$.

Identifier renaming. The names of identifiers contain meaningful expressive information about the object they are used for. To avoid scanning engines relying on nomenclature for detection, it renames original identifiers such as classes, fields, methods and packages names by random or predefined names [12,42,65–69]. Renaming with meaningless names removes semantic information and induces more time to analyze an application code [31,53]. APK obfuscation tools reuse short meaningless names for identifier renaming [31,70,71]. Code listing 10 illustrates identifier renaming by ProGuard [72].

```
1 const-string v10, "profile"
2 const-string v11, "mount -o remount rw system\nexit\n
    "
3 invoke-static {v10, v11}, Lcom/android/root/Setting
    ;->runRootCommand(Ljava/lang/String;Ljava/lang/
    String;)Ljava/lang/String;
4 move-result-object v7
```

Listing 9: A byte code A code fragment from DroidDream malware

```
1 const-string v10, "profile"
2 const-string v11, "mount -o remount rw system\nexit\n
    "
3 invoke-static {v10, v11}, Lcom/hxbvgH/IWNcZs/jFAbKo
    ;->axDnBL(Ljava/lang/String;Ljava/lang/String;)
    Ljava/lang/String;
4 move-result-object v7
```

Listing 10: Listing 9 after identifier renaming

Data reordering. Data and code locality plays a significant role during code analysis. Thus data reordering is often employed for data delocalization. It alters the order of data units within a code segment. It is applied to: (a) Reorder instance variable: randomizes the variable declarations, (b) Reorder array: Data reordering is also applicable on array [73,74]. An example of which is reordering the elements in an array, storing the ith element in a new position determined by a function $f(i)$, and (c) Reorder method: Methods within a code segment are reordered to harden the code reversing. This syntactic change is helpful in evading scanning engines based on instruction or dalvik opcode order [13,75].

Array restructuring. Arrays as variable are transformed based on their structural property into subarrays or merged into one. Array flattening decreases the dimensions, whereas array folding increases the array dimensions as illustrated in Fig. 10. Array reordering as introduced in 3.2.2 decreases the data locality by permuting the variables in array [76].

Data Space Randomization (DSR). DSR was introduced by Bhatkar and Sekar in [77]. DSR performs XOR operation on data variables stored in application memory with masks. Masks are randomly generated at runtime to mask the data variable. To unmask a data variable, the same mask value is required. DSR gives protection against targeting process memory during application execution. Code listings 11 and 12 illustrates obfuscating variables using DSR .

```
1 int x = 3;
2 int y = 5;
3 z = x + y;
```

Listing 11: Addition operation

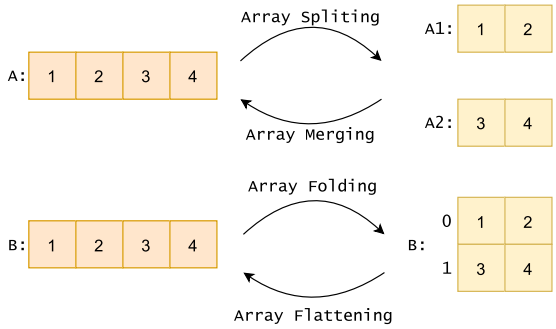

Fig. 10. Array restructuring transformations.

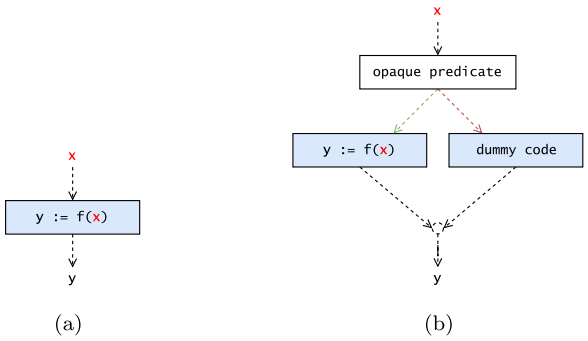

Fig. 11. Example of opaque predicates: In (a) the code block produces an output y depending on the input x. (b) presents an obfuscated opaque predicate (that is always true) and a code block ("dummy code") that is never executed.

```
1 int x_mask = random();
2 int y_mask = random();
3 int x = 3 ^ x_mask;
4 int y = 5 ^ y_mask;
5 int mask_z = random();
6 int sum = ((x ^ mask_x) + (y ^ mask_y)) ^ mask_z
```

Listing 12: DSR obfuscated addition operation

3.2.3. Code logic obfuscation

Code logic obfuscation includes transformations targeting the logic or semantics of the code, thus adding complexity [44,78].

Opaque predicates. A predicate is a function which is called opaque if the outcome of the function is difficult to predict even after statically analyzing it [44]. An opaque predicate is a condition that will make it more complex and difficult to interpret the code until the condition is executed. This condition is followed by a conditional jump [53]. Unable to predict opaque predicate, both branches of the jump are seen possible even if only one is executed at each run time [13,79]. As depicted in Fig. 11 it is even referred to as a branch that always executes in one direction, which is known to the author but is unknown to the analyst. Static properties of opaque predicates prevent the code interpretation from an analyst. The notion of dynamic opaque predicates was introduced by Palsberg et al. in [80]. It implements a group of interconnected opaque predicates, all computing the same output in a single execution, but to the different output in other executions. Majumdar and Thomborsom in [81] introduced the opaque predicates that were temporarily unstable.

Code reordering. Code reordering modifying the byte code instructions order in smali methods of an application [82]. It performs instruction reordering along with goto instructions to preserve the execution order. Code reordering is employed by malware authors as a defense against pattern matching based scanning engines [83]. Dalvik byte code in listing 14 have code reordered from its original in listing 13 [5]. Code reordering is applied to the methods that contain independent instructions and do not have control jumps.

```
1 const-string v10, "profile"
2 const-string v11, "mount -o remount rw system\nexit\n
    "
3 invoke-static {v10, v11}, Lcom/android/root/Setting
    ;->runRootCommand(Ljava/lang/String;Ljava/lang/
    String;)Ljava/lang/String;
4 move-result-object v7
```

Listing 13: A byte code A code fragment from DroidDream malware

```
1 goto :i_1
2 :i_3
3 invoke-static {v10, v11}, Lcom/android/root/Setting
    ;->runRootCommand(Ljava/lang/String;Ljava/lang/
    String;)Ljava/lang/String;
4 move-result-object v7
5 goto :i_4 # next instruction
6 :i_2
7 const-string v11, "mount -o remount rw system\nexit\n
    "
8 goto :i_3
9 :i_1
10 const-string v10, "profile"
11 goto :i_2
```

Listing 14: Reversed byte code of listing 13

Instruction substitution. Instruction substitution was first introduced in [82]. It is based on the fact that in the instruction set architecture, instructions can be replaced by other existing equivalent instruction. Registers can also be substituted. For example, instruction `const/4 v0, #int 0` in listing 13 can be replaced by `const/16 v0, #int 0`.

Dead or junk code insertion. As the name suggests, the term dead or junk code refers to the code which does not execute or cannot be reached during execution [52]. Its insertion induces additional complexity for analysis and alters static file features such as application digest or n-gram based signatures. As depicted in Fig. 11, dead code is often supplemented by opaque predicates (either true predicate or false predicate) to evade its execution [53]. Addition of no-operation, well known as NOP instruction (opcode 0x00 in Dalvik) is a simple dead code insertion without any effect to application functionality [13]. Application scanning engines are usually able to identify code using dead code insertion alone [14,84], thus encouraging malware authors to combine it with other code obfuscation methods. Android however, uses code-shrinking to remove unused methods, fields, and classes from the application and its libraries. Android also employs code optimization to remove unused conditional branches [85].

Function call addition and removal. Proposed by Cohen [82] addition of function call implies creating a function of the sequence of instructions and substituting the original sequence with call to the function. Removal of function call implies substituting all calls to a function with the body of the function and removing the function. Function call addition and removal is also popularly known as *code inlining and outlining* [7].

Loop transformations. Loop transformations were originally designed for performance optimization, but some of them are useful code obfuscation techniques as they increase the structural complexity of the code [86]. Popular loop transformations are: (a) Loop unrolling, which replicates the code inside the loop and decreasing the loop counter; (b) Loop tiling originally designed for code optimization creates an inner loop to optimize loop cache behavior; and (c) Loop fission bifurcates a loop into multiple loops [84].

Call indirections. Scanning engines often rely on advanced signatures based on application call graph to identify or classify a malware. Call indirection aims to evade it by redirecting a method invocation in the smali code to proxy methods that then calls the original method [14]. Syntax (argument type, return type, registers and invocation type) of these proxies are generally kept same as that of the original method. Listing 13 and 14 are the original smali code and smali code with call indirection respectively. A proxy method is inserted at line 5 in listing 14 with a random identifier.

```
1 .class public final L<ClassName>;
2   .super Ljava/lang/Object;
3   .source "<ClassName>.java"
4
5   .method public static FogLow(Ljava/lang/String;
      Ljava/lang/String;)V
6     .registers 2
7     .prologue
8    invoke-static {p0, p1}, Landroid/util/Log;->d(
       Ljava/lang/String;Ljava/lang/String;)I
9     return-void
10  .end method
```

Listing 15: Original byte code showing a method.

```
1 .class public final L<ClassName>;
2   .super Ljava/lang/Object;
3   .source "<ClassName>.java"
4   # direct methods
5   .method public static <Identifier>(Ljava/lang/
      String;Ljava/lang/String;)I
6     .registers 3
7     .prologue
8     invoke-static {p0, p1}, Landroid/util/Log;->d(
        Ljava/lang/String;Ljava/lang/String;)I
9     move-result v0
10    return v0
11  .end method
12  .method public static FogLow(Ljava/lang/String;
      Ljava/lang/String;)V
13    .registers 2
14    .prologue
15    invoke-static {p0, p1}, L<ClassName>;->
        Identifier>(Ljava/lang/String;Ljava/lang/String
        ;)I
16    return-void
17  .end method
```

Listing 16: Call indirection: Byte code with proxy method inserted.

Program encoding. Strings and data structures used in an application are stored in the `.dex` files. Strings and byte code patterns are used to create detection signatures for malware identifications [14,53]. Encoding converts a data or string representation into a different one with encoding function. The encoded sequence is deobfuscated during execution [82]. Developers use program encoding techniques such as encryption [87,88] and compression [89]. Zhou et al. in [54] introduced a variant of encoding via mixed-mode computation over boolean-arithmetic algebras. Program encoding induces additional computational cost, which is relatively high depending upon the type of encoding. A resilient encoding method is computationally expensive.

```
1 String option = "@^@#\x ‘1 m*7 %**9 _!v";
2 this.execute(decrypt(options));
```

Listing 17: String encryption in java

```
1 const-string v1, "\t\u001b\u0002\u0019\u0019\u0001\
    u0014EX"
2 const/16 v2, 0x79
3 invoke-static {v1, v2}, Lcom/software/app/Activator$2
    ;->getChars(Ljava/lang/String;I)Ljava/lang/String;
4 move-result-object v1
```

Listing 18: String encryption by DashO.

Encryption: It protects the application against static analysis. Encrypted code is decrypted on-the-fly during runtime. Scanning engines based on bytecode sequences and string based fingerprinting are futile against it [13]. Popular encryption techniques employed to strengthen Android applications include string encryption and class encryption.

1. *String Encryption:* Strings are significant data structures often encrypted to prevent application identification based on string-based features such as package names and permissions [90–92]. As a result, string encryption could effectively hinder hard-coded static scanning by rendering strings unreadable [20,31,93]. The original string is stored in an encrypted form and requires an additional decryption function [65–68,90]. Listing 17 illustrates application of string encryption and `decrypt()` subroutine at source code level in java. Listing illustrates an instance of string encryption by DashO, a commercial obfuscator developed by PreEmptive Solutions. DashO replaces strings with function calls to a decryption function. The same decryption function, `Activator$2;->getChars`, is called multiple times in the application code [71].
2. *Class Encryption:* Class encryption is an advanced code obfuscation technique which encrypts a class. The encrypted class is decrypted and loaded at runtime by a separate function. Maiorca et al. in [20] have created a class obfuscation scheme which encrypts and compresses (by GZIP algorithm) each class. During runtime encrypted class is decrypted, unzipped and loaded in memory. The computational overhead of class encryption is high along with its resilience against static analysis based reverse engineering [42,94].

Self modifying code. Malware authors use all tricks to evade scanning engines, and one of those is called polymorphism or self modifying code. It allows application code to change itself without changing its functionality. Polymorphism, which has successively been exploited over Windows OS, is being used against Android too. Malware `Android.Opfake` [95] employs server-side polymorphism to modify itself every time it is updated to evade detection. The polymorphic code is able to modify itself by variable data changes, file reordering and dummy file insertion. Polymorphic malwares often use bytecode encryption, which encrypts pieces of an application to be decrypted only at runtime [18].

Control Flow Flattening (CFF). Control flow flattening, which hides the control flow graph of the application was introduced by Wang et al. [96] and Chow et al. [97]. It is an effective and deobfuscation resilient method [98,99]. CFF uses switch constructs instead of using easily identifiable loops and conditional jumps to divert

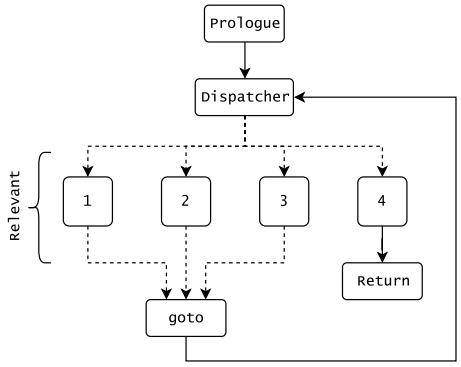

Fig. 12. Control Flow Flattening: Flattened sequence of the original control flow sequence 1 → 2 → 3 → 4 → Return.

control flow. According to Wang et al. in [98] structure of CFF obfuscated code contains (a) Prologue: entrance block of the CFF (b) Dispatcher: a conditional jumping block such as switch; (c) Return: basic code block which returns; and (d) Relevant: maintains operation of the original function. Fig. 12 illustrates structure of control flow after flattening.

API hiding. Applications are often characterized as sequences of Application Programming Interface (API) methods. API calls help in identifying application interactions with the operating system applications. Smartphone is a host of privacy sensitive information such as phone numbers, call details, SMSs, calendar and location information which are accessed by API calls. Scanning engines often rely on API calls for malware detection [37,100–103]. For example, malware sample using getDeviceId() followed by sendTextMessage() is highly probable to leak device's identifier data. API hiding based code obfuscation is used by applications to prevent discovery of API usage pattern. API hiding uses Java reflection mechanism to hide invocations of sensitive APIs, such as cryptographic functions [6,42,104]. Listing 20 show an instance of API hiding by DexProtector on code listing 19 [42]. API calls `getText()` and `toString()` are obfuscated using API hiding.

```
1 const-string v1, "Hello World"
2 invoke-virtual {v0}, Landroid/widget/TextView;->
    getText()Ljava/lang/CharSequence;
3 move-result-object v2
4 invoke-interface  {v2}, Ljava/lang/CharSequence;->
    toString()Ljava/lang/String;
```

Listing 19: Smali code with a string variable and API calls.

```
1 const-string v1, "\ub6e8\ud801\ub2bb"
2 invoke-virtual {v0}, Lcom/dexprotector/ha;->gbeab(
    Ljava/lang/Object;)Ljava/lang/Object;
3 invoke-static {v2}, Lcom/dexprotector/ha;->oodab(
    Ljava/lang/Object;)Ljava/lang/Object;
```

Listing 20: Smali code with encrypted string variable and API hiding.

Java reflection. Reflection is a popular feature in Java programming language to evade static analysis, as it allows object interaction at runtime. Reflection is popular among developers to obfuscate sensitive library and API calls [21,31]. It transfers execution flow to the desired code segment implicitly, thus evading static analysis techniques. Code listing 21 and 22 illustrate usage of reflection for code obfuscation.

```
1 Lock myLock = new Lock();
2 key = myLock.getKey();
```

Listing 21: Before reflection.

```
1 Class c = Class.forName("me.locklink.RefletiveClass")
    ;
2 Object o = c.newInstance();
3 Method m = c.getMethod("getKey");
4 Object r = m.invoke(c);
```

Listing 22: Reflection based code obfuscation.

Native code obfuscation. Android comprises the Android Native Development Kit (NDK), which allows developers to execute their C and C++ code (called as native code). Native code is popular amongst developer [105–107]. Analysis methods available for byte code are ineffective against native code, thus making them favorite for malicious payloads [105,108–110]. To make it more difficult for analysts, native code obfuscation exists. Obfuscator-LLVM, is a popular native code obfuscator for ARM and x86 architecture [111]. It offers (a)Instruction substitution; (b) Control Flow Flattening and (c) Junk code addition at native code level.

Library hiding. Applications rely on system and third party libraries for features and interactions. Libraries are cornerstone of Android ecosystem, as they provide functionality to developers such as advertisement, authentication and social networking. An application thus can be classified as malicious based on its library usage. Various approaches for library detection exists [112–120] in literature. These detection methods are challenged by code obfuscation methods as they mostly rely on nomenclature based matching of package or classes [121]. Library class and package renaming approaches are used for library hiding. Another approach followed by some developers is to recreate custom versions of library by merging, splitting or changing existing libraries [7].

3.3. Preventive techniques

Preventive techniques comprise of methods which prevent or make difficult to analyze an application. Scanning engines, application hosting platforms employ dynamic analysis methods for malware analysis and detection. Below we discuss a broad range of prevention techniques used both by malwares and benignwares to detect and evade analysis environment. Benignwares such as banking applications use them for primary objective to keep user's authentication and transaction information confidential and intact. Prevention techniques prime objective to detect analysis environment is based on identifying static properties of environment, runtime sensor information of Android and other emulator related properties.

3.3.1. Anti tampering

Attackers are always trying to reverse popular applications to inject malicious code. An attacker using the tampered application can get inside smartphone to access, manipulate and exchange user data [11,42,122]. Anti tampering popular as *Integrity Checking* uses methods to ensure that application code and resources are not altered by a third party. Basic anti tampering is to check for integrity or digest at startup. A digest mismatch with the developer's certificate suggests a tampered application [123]. A distributed approach of code integrity checking using checkers is introduced in [124]. Fig. 13 illustrates multiple checkers responsible for integrity check of code segments are planted at different locations of an application.

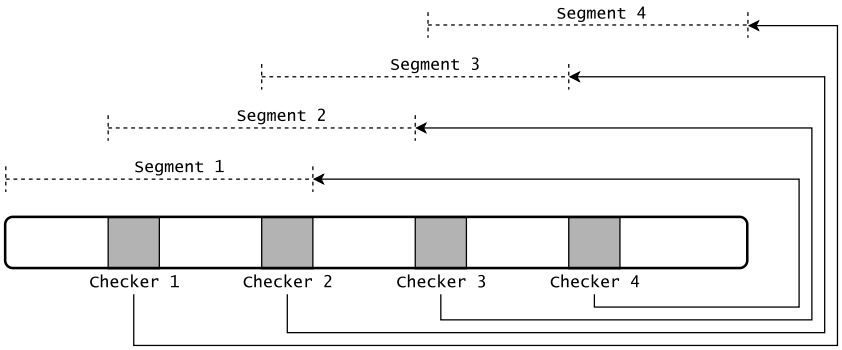

Fig. 13. Control Flow Flattening: The original control flow in (a) is flattened to (b).

```
1 PackageManager pacman = context.getPackageManager();
2 List appInfoList = pacman.getInstalledApplications(
    PackageManager.GET_META_DATA);
3
4 for(ApplicationInfo appInfo : appInfoLst) {
5   if(appInfo.packageName.equals("de.robv.android.
      xposed.installer"))
6
7   if(appInfo.packageName.equals("com.saurik.substrate
      "))
8 }
```

Listing 23: Anti Hooking: Code snippet to identify presence of hooking frameworks

3.3.2. Anti hooking

Application interaction with Android OS and other application is a probable attack surface. With application code intact, API calls can be hooked to insert functionality at runtime [11,125–127]. Hooking allows an attacker or analyst to alter method arguments, method return or even a complete method. Hooking detection is considered difficult as hooks are generally placed from higher privilege levels. Anti hooking comprises of the mechanisms for detection of hooking framework (e.g. Cydia Substrate [128], Frida and Xposed [129] framework) in memory. Current anti hooking solutions rely on fingerprints of hooking frameworks for their detection [11]. Code listing 23 illustrates the identification of Xpose and Cydia Substrate frameworks using package name artifact. Jose Lopes in [130] uses file path and stack trace (strace) artifacts to detect popular Xposed and Cydia Substrate frameworks.

3.3.3. Anti debugging

Debuggers were primarily designed for developers to find bugs in their applications, and are also used by reverse engineers to perform dynamic analysis of an application. Debuggers operate by setting up interrupts at a particular instruction in an application. Obfuscation methods successful against static analysis approaches can be countered by debugging. Two kinds of debuggers can be attached to Android: *Native code debugger* and *JDWP(Java Debug Wire Protocol) debugger* [11,131]. An application developer or malware author detect and evade debugger by anti debugging techniques. A debugger if detected, the anti debugger can invoke crashes, custom action, data corruption or warning signal to a remote server. Anti debugging checks for unique artifacts of the environment to detect and identify the presence of a debugger [124].

Artifacts here include debugger flag and kernel system calls related to debugging. Listing 24 is code snippet to detect presence of debugging environment using `FLAG_DEBUGGABLE` attribute in AndroidManifest.xml.

```
1 public static boolean checkDebuggable(Context context
    ){
2 return (context.getApplicationInfo().flags &
    ApplicationInfo.FLAG_DEBUGGABLE) != 0;
3 }
```

Listing 24: Anti Debugging: Code snippet to identify presence of debugger packages

Anti debugging methods [131–133] targeting Native code debugger are: (a) Execution timing mismatch if code is executed in debugger; (b) Interrupt signal for an application will be diverted to debugger thus enabling detection; and (c) Application code integrity check as insertion of breakpoint shall alter the hash value of the code.

Anti debugging methods [131–133] targeting JDWP debugger are: (a) `debuggerConnected` and `debuggerActive` member variables in `DVMGlobals` datastructure is 1 in Dalvik VM instance of application for an attached debugger; and (b) The `count` member of `BreakpointSet` structure in Dalvik VM instance represents number of breakpoints set can detect presence of breakpoints.

3.3.4. Anti emulation

Due to large scale development and evolution of malware, security researchers are using automated dynamic analysis techniques for detection [127,134–139]. Analysts performing dynamic analysis, executes an application in a virtual environment, an emulator or a sandbox as it allows them to monitor and inspect application's state. Several emulators and sandbox detection mechanisms are employed by applications to exhibit different behavior on detecting an emulation or virtual environment [11,126,127,140–144]. Anti emulation is based on the principle that creating a complete and real system emulator is very difficult. Anti emulation methods are developed by observing and detecting differences between application interaction with a real and virtual system. Anti emulation artifacts or fingerprints thus may be differences due to system hardware or software state values or application execution differences [11].

They are categorized into static heuristics and dynamic heuristics [140]. Static heuristic techniques are based on the static properties which are initialized to default values in an emulation environment. Whereas dynamic heuristic techniques are based on the information provided during runtime. Table 2 illustrates a compiled list of artifacts employed by applications to detect the presence of emulation environment.

3.3.5. Device binding

Banking applications requires to work only on the installed device with which the account has been bound. This requirement of application services binded to the device is known as device

Table 2
Artifacts to detect presence of an emulator or sandbox.

	Artifact types	Artifacts
Static heuristics	Device identifier	Android device ID; IMEI; IMSI
	Current build	Build.ABI; Build.ABI2; Build.BOARD; Build.BRAND; Build.DEVICE; Build.FINGERPRINT; Build.HOST; Build.ID; Build.MANUFACTURER; Build.MODEL; Build.PRODUCT; Build.RADIO; Build.SERIAL; Build.TAGS; Build.USER
	Telephony manager	TelephonyManager.getDeviceId(); TelephonyManager.getLine1Number(); TelephonyManager.getNetworkCountryIso(); TelephonyManager.getNetworkType(); TelephonyManager.getNetworkOperator(); TelephonyManager.getPhoneType(); TelephonyManager.getSimCountryIso(); TelephonyManager.getSimSerial Number(); TelephonyManager.getSubscriberId(); TelephonyManager.getVoiceMailNumber()
	Hardware components	/proc/cpuinfo; /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq; /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
Dynamic	Sensor information	Accelerometer; Light; Magnetic field; Orientation; Proximity sensor; Rotation vector; Gravity; Gyroscope sensor; Temperature
	CPU Performance	Differential CPU performance analysis
	Graphical performance	Emulators exhibit lower Frames Per Second (FPS)
	Presence of other apps	GoogleLoginService.apk; GoogleServicesFramework.apk; Phonesky.apk; Vending.apk; VZWBackupAssistant.apk

binding. Device binding act as a physical second factor authentication [11,145]. Device binding fingerprints the device which are matched during app initialization. Device fingerprinting requires to be unique such as device IMEI number and Android Device ID [146].

3.3.6. Anti rooting

Android device manufacturers limit user control over the operating system and applications. Every application in Android executes with its context inside a sandbox. Rooting allows user to attain full privilege over hardware, OS, applications and data stored among them [124]. Rooting is achieved by obtaining persistent root access of the OS [147], which is achieved by installing a customized superuser (su) binary. `su` binary is then used to perform operations as root such as remove application restrictions and security constraints [148,149]. Sun et al. in [150], detailed five rooting methods namely: (a) Fastboot or Download mode; (b) Custom recovery; (c) Bootable SD card; (d) Rooting apps; and (e) Privileged ADB. A rooted device is a risk for applications transacting sensitive information (such as financial applications) or applications that restrict user interaction (such as a company's internal mailing application). High valued applications perform rooting detection to prevent exposure to a rooted device. Techniques and artifacts to detect presence of a rooted device are popularly categorized into Anti rooting. Depending on how a device is rooted, artifacts for its detection vary. Table 3 presents a comprehensive view compiled from existing root detection methods [11,151,152].

3.3.7. Anti tainting

A recent popular analysis technique known as taint analysis is based on the data flow from source to sink using tags [9,153]. For instance, in the case of sending DeviceID information over the network, method returning DeviceID is the source and socket's send method is a sink. Movement of tagged data in taint analysis is logged, thus enabling the detection of information leakage. Taint analysis for Android was introduced in [154]. Evasion of information flow detection taint analysis is known as Anti Tainting. Hoffmann et al. [9] and Sarwar et al. [155] experimented anti tainting by monitoring sensitive sources (such as contact list, location data) and instead of original data sent the duplicated data or a copy to the sink.

3.3.8. Anti keylogger

Android offers the feature to install third-party keyboards, which may be designed to extract user's sensitive information or credential data during typing [11]. Few application hardening solutions such as Promon Shield uses anti key-logging by providing its own keyboard whenever an application is calling `SecureEditText` and `SecureKeyboard` classes. Another approach followed is to maintain a whitelist of trusted keyboard applications. An application may quit on finding the presence of a keyboard not in the whitelist.

```
1 URL url = new URL("https://www.policeuniversity.ac.in
    ");
2 HttpsURLConnection con = (HttpsURLConnection)url.
    openConnection();
3 con.connect();
```

Listing 25: Sample code in java to open HTTPS connection

3.3.9. Anti-screen reader

Malicious applications can target for screen captures to eavesdrop user activity and credentials using Android accessibility services [156]. Similar to Anti Key-logging, Anti screen reader technique try to detect presence of a screen reader when a critical application such as banking is running. An application can restrict screen captures by activating `FLAG_SECURE`, which is a display flag to treat window content of the application as secure [157].

3.4. Other techniques

In this section we consider methods which cannot be into above categories.

3.4.1. Network communication hardening

A stealthy network communication lowers the possibility of eavesdropping over network connection or interfaces. This subsection presents techniques to strengthen network communications, which may serve as preliminary network security by developers [11,158].

- HTTPS: As most applications use HTTP to transceive data, Android supports HTTPS client connection using `HttpsURLConnection`. HTTPS encrypts traffic to prevent eavesdropping. It provides TLS based confidentiality and authentication. Listing 25 a sample code to open a client connection using HTTPS [159].
- DNS: Android 10 and above supports DNS request over TLS using `DNS-over-TLS` mode. The API `DnsResolver` can be used for name resolutions.
- Certificate and Public Key Pinning: Whenever an HTTPS connection is made to server, authentication of server certificate is requested by underlying TLS connection. Certificates

Table 3
Artifact list to detect a rooted device.

Categories of artifacts	Artifacts
Directories	/cache; /data; /dev/data/app; /data/data; /data/dalvik-cache; /data/local; /data/local/bin; /data/local/xbin; /magisk/.core/bin; /sbin; /su/xbin; /su/bin; /system/app; /system/bin; /system/bin/.ext; /system/bin/failsafe; /system/sd/xbin; /system/usr; /system/usr/we-need-root; /system/xbin
Rooted device	busybox; daemonsu; kingroot; kinguser; supersu; superuser
Root privilege	adfree; greenify; kerneladiutor; setcpu; shootme; stericson; titanium
Evading root detection	chainfire; rootcloak; rootcloakplus; xposed
Package names	com.devadvance.rootcloak; com.devadvance.rootcloakplus; com.grarak.kerneladiutor; com.jrummy.apps.build.prop.editor; com.jrummy.root.browserfree; com.jumobile.manager.systemapp; com.koushikdutta.superuser; com.oasisfeng.greenifiy; com.thirdparty.superuser; de.robv.adnroid.xposed.installer; org.namelessrom.devicecontrol; stericson.busybox

```
1 private static boolean isValidJPEG(String path)
    throws IOException
2 {
3   RandomAccessFile file = null;
4   try{
5     file = new RandomAccessFile(path, "r");
6     long length = file.length();
7     if (length < 10L){
8       return false; }
9     byte[] head = new byte[2];
10    file.readFully(head);
11    file.seek(length - 2);
12    byte[] tail = new byte[2];
13    file.readFully(tail);
14  return head[0] == -1 && head[1] == -40 && tail[0]
      == -1 && tail[1] == -39;}
15  finally{
16    if (file != null){
17      file.close();} } }
```

Listing 26: Code to validate JPEG file based on header (FF D8) and footer (FF D9) values

presented by a server are issued and verified by Certificate Authority (CA). A valid certificate based https connection greatly reduces the chances of man-in-the-middle (MITM) attacks. Certificate pinning checks the authenticity of server certificate. Only when the server is authenticated, the connection is established.

- Sanitization and Validation: Validation is the process of ensuring the data submitted to server is sensible i.e. it fulfills the constraints of validity. For example, if server is expecting a 10 digit number then making sure that the submitted value is 10 digit. It reduces the likelihood of memory corruption and injection attacks. Listing 26 illustrates validating a submitted file to be a JPEG.
- End to End encryption: Limitation with https protection level is that it provides a confidential communication channel between client and the server. But there exist scenarios in which an end to end communication between two peers going through multiple servers need to be encrypted (for instance, Whatsapp). It means the intermediary servers or nodes shall have no information about data content but metadata. End to end encryption solution is specifically designed as per application functionalities.

3.4.2. Resource centric obfuscation

Assets and res directories of an APK are responsible to store resource files such as audio, images. They are even used by developers to store database files. Below we discuss different hardening techniques used by developers to protect or evade.

Resource renaming. List of resources and their names are defined in the XML files in the APK. Scanning engines often identify malicious applications based on resource files with specific names. This class of obfuscators aims at evading signatures based on string matching. For example, Android Rootnik malware can be identified by the presence of `secData0.jar` file in the assets folder in APK [108]. In resource renaming protection technique, user defined resource identifiers are renamed with random or predefined naming pattern and the same is updated with respective files. Preda et al. in [13] replaced resource names by first 8 characters from MD5sum of the file name.

Resource encryption. Resource encryption technique encrypts the file content of resources and assets. For it to work, subsequent changes are required in byte code for resource decryption during execution. For example, `secData0.jar` in Rootnik malware is an encrypted resource file [108]. Developers may use existing popular encryption methods (for e.g. DES, AES and XOR based encoding) or custom methods for it.

Code and package hiding. Malware authors often use resource files to hide malicious elements such as code segments, `DEX` file and `APK` inside resource files [21]. It makes the application look benign upon inspection. For example, GingerMaster malware hides it malicious script in resources with the extension of `.png` [160]. Another malware, `Gamex.A!tr` contains a ZIP file in Assets under the name `logos.png` [21]. Malware even employ steganography to hide malicious packages and JAR files in valid resource files. The objective is to embed an executable piece of code or payload to evade detection, static analysis in particular. Advantage with steganography based obfuscation approach is that the malicious content is difficult to recognize during the static phase as it is revealed at execution time only. For example, `SmsZombie.A!tr` uses jpeg file to hide package using steganography. Code and package hiding in combination with encryption increase the obfuscation level [21,161].

4. Effectiveness of obfuscation methods

Faruki et al. in [7] discussed obfuscation methods, application protection and deobfuscation methods specific to Android. Dong et al. in [31] provided an understanding into Android code obfuscation and carried out a large scale investigation on 114,560 samples for its usage.

Obfuscation methods are new normal for both developers and malware authors. Malware authors leverage above discussed techniques such as code obfuscation, anti-debugging, anti-hooking to bypass detection. Developers furthermore engage them to defend against evasive systems. Never ending cycle of detection, anti-detection and anti-anti-detection propel innovation in both detection and evasion. While the effectiveness of detection methods against ever improving anti-detection is a question to contemplate, limited research in available literature focuses on it. Evolution of these techniques in the form of a timeline is illustrated in Fig. 14. It has been created based on the introduction of techniques in the literature over the decade to the best of our knowledge.

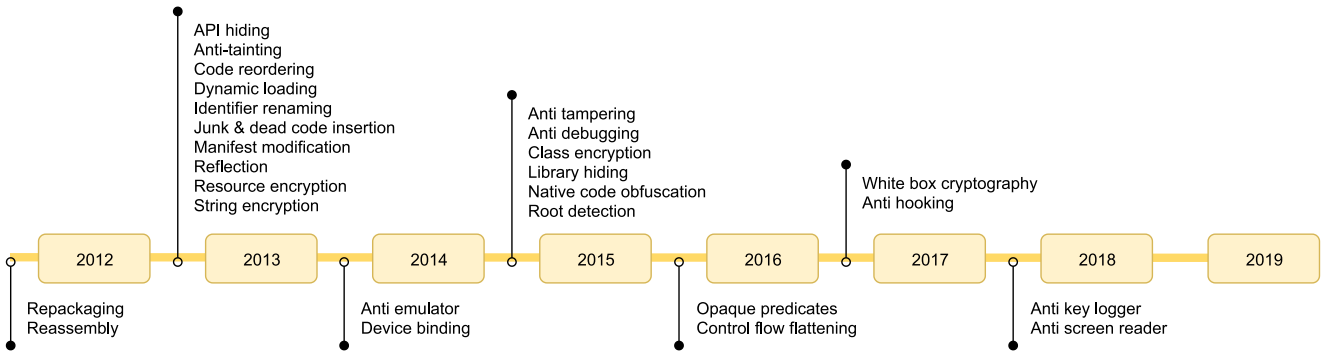

Fig. 14. Evolution of obfuscation and preventive techniques.

Tam et al. [18], Nigam [162] and Suarez-Tangil [161] have extensively discussed the evolution of Android malware over the last decade. Apvrille and Nigam in [21] explore the practical usage of stealth techniques by Android malware. Christodorescu and Jha in their seminal paper [163] tests the effectiveness of different code obfuscation methods on desktop malware to evade popular scanning engines. Moreover, it reasons their low resilience against code transformations. Mairco et al. in [20] did a comparative evaluation of trivial obfuscation, string encryption, reflection and class encryption techniques. For trivial encryption, it considered class, field, filename, method and package renaming followed by repackaging. Furthermore, they assess the cumulative effects of obfuscation techniques. Preda and Maggi in [13] assess the effectiveness of mobile anti-virus against particular classes of code transformations. It evaluates techniques categorized into five classes namely, Trivial techniques(Repackaging, Reassembly and Realignment), Simple Control Flow modifications (Junk code insertion, Debug symbol stripping, Defunct code insertion and Unconditional jump insertion), Advanced Control Flow modifications (Call indirection, Code Reordering, Reflection and Opaque predicate insertion), Renaming (Resource renaming, Identifier renaming and Package renaming) and Encryption (Resource encryption, Native code encryption and string encryption). Like Mairco et al. in [20], it also concludes encryption to be effective in evading all anti-virus products. It used DES based encryption, whereas [20] employed XOR for encryption.

Badhani and Muttoo in [165] calculates control graph based similarity measure between original and obfuscated apps. It divides code obfuscation into five levels based on changes it performs. It uses techniques from [5,13,164] for creating obfuscated apps. It concludes its resilience towards detecting single level obfuscations, but during multiple obfuscations, similarity was found low. Cho et al. in [94] propose an obfuscation assessment scheme to evaluate the obfuscation level. It tries to quantify by calculating obfuscation score based on sensitive API usage. Strength of API hiding, Repackaging and Class encryption obfuscation methods are computed. Table 4 briefs work on detection of obfuscation types. There exist multiple works on obfuscation detection but very limited methods focus on the detection of specifics. It also compares the obfuscation strength of each method based on detection results available in the literature. It is worth highlighting, that Resource Encryption, String Encryption and Class Encryption are hardest to detect and most evasive. While they induce code complexity, they also cause size overhead [20].

5. Android obfuscators and hardening tools

Above discussed techniques are widely used to mitigate automatic analysis, reverse engineering and securing the intellectual property from prying analysts [31,98]. Multiple frameworks are available in the market providing range and combination of code obfuscation and application hardening methods [5,20,65–68,72,93,111,169,177–180]. These open source and commercial frameworks differ in scope and obfuscation depth [13]. Code obfuscators target different code levels namely: Java code, bytecode and native code.

We have performed a comprehensive analysis of popular Android application hardening tools with reference to the features and techniques. Table 5 illustrates the compiled list.

6. Related works

Android is a market mover and popular target amongst malware authors, various studies on Android application protection and code obfuscation are available in the literature. A comprehensive Android threat taxonomy is detailed by Shabtai et al. in [181]. Current literature studies focus on evolving hardening techniques; its challenges and improvements [182]; detection [5,183] and evasion work. Tam et al. [18], Nigam [162] and Suarez-Tangil [184] discussed evolution of Android malware over the last decade. Apvrille and Nigam in [21] focussed on practical usage of hardening techniques by Android malwares. In [53] and [8] authors survey existing obfuscation techniques employed in various systems. Faruki et al. discussed obfuscation methods, application protection and deobfuscation methods specific to Android in [7].

Dong et al. in [31] provided an understanding into Android code obfuscation and carried out a large scale investigation on 114,560 samples for its usage. Various static and dynamic code obfuscation approaches are presented in [5,6,22,31,70,164] such as renaming, string encryption, control flow obfuscation and reflections. Furthermore, they also analyze obfuscation at bytecode. Effectiveness of these hardening measures is also available in literature [9–13,20,52,94,126,185]. Park et al. in [12], empirically analyzed application similarity between original software and the one transformed by code obfuscation. Furthermore, it tried to detect obfuscation and check its authenticity. Code obfuscation detection methods are available in [6,14,42,161,186–188]. Cho et al. in [42] introduced DexMonitor a Dalvik byte code analysis framework. State of art deobfuscation methods is discussed in [71,98,99,183].

Haupert et al. presented a comprehensive survey of Runtime Application Self-Protection (RASP) methods for application hardening and analysis environment detection [11]. They presented an in-depth study of Promon Shield, a market leader in RASP tool. Bulazel in [17] experimented a fingerprint based method for multiple platforms for evasion detection and evasion mitigation. Furthermore, it presents case studies of defensive and offensive evasion approaches. Other anti-debugging, anti-rooting, anti-emulator and anti-tampering approaches are evaluated in [47,127,131,140,150,151].

Different methods on packaging exist in [70,189] and the approaches targeting packed applications are evaluated in [39–41].

Table 4
Comparative analysis of effectiveness of obfuscation techniques.

Obfuscation	[164]	[20]	[13]	[94]	[14]	OS	SO
Repackaging	✓	✓	✓	✓		○	○
Reassembly	✓	✓	✓	✓		○	○
Dead code insertion	✓		✓		✓	○	◔
Call indirection	✓		✓		✓	◔	◔
Code reordering	✓		✓		✓	◔	◔
Opaque predicate			✓			◔	◔
Resource renaming	✓	✓	✓			◔	◔
Method & field renaming	✓	✓	✓		✓	◔	◔
Package renaming	✓	✓	✓		✓	◔	◔
Resource encryption		✓	✓			●	◔
String encryption	✓	✓	✓		✓	●	◔
Class encryption		✓		✓		●	●
API Hiding				✓		◔	◔
Reflection	✓	✓	✓			◔	◔
Combinations		✓	✓	✓			

OS: Obfuscation Strength; SO: Size Overhead; Notations from low to high: ○ ◔ ◔ ● .

Table 5
Comparative analysis of android application hardening tools.

Tools	CO	AD	AE	AH	AR	AT	RP	DB	AK	AS	SN
Allatori [65]	✓										
APK Protect [166]	✓	✓									
Arxan [167]	✓	✓		✓	✓	✓					
Baidu [167]	✓	✓									
Bangcle [168]	✓	✓									
CrackProof		✓	✓		✓	✓					
DexGuard [66]	✓	✓	✓	✓	✓	✓					✓
DashO [169]	✓	✓	✓	✓	✓	✓	✓	✓			
DexProtector [67]	✓	✓	✓	✓	✓	✓	✓				✓
Entersekt Transakt [170]						✓		✓	✓		✓
Ijiami [171]	✓	✓									
InsideSecure [172]	✓	✓			✓	✓					
MobileProtector [173]	✓	✓		✓	✓			✓	✓		✓
ProGuard [72]	✓										
PromonShield [174]	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
SecNeo AppShield [175]	✓	✓									
Stringer [68]	✓										
WhiteCryption [176]	✓	✓			✓	✓					
YGuard [177]	✓										

CO: Code Obfuscation, AD: Anti Debugging, AE: Anti Emulator, AH: Anti Hooking, AR: Anti Rooting, AT: Anti Tampering, RP: Resource Protection, DB:Device Binding, AK: Anti Keylogger, AS: Anti Screen reader, SN: Secure Network.

Techniques on Assets and resources are presented and evaluated in [21,190]. Suarez-Tangil in [190] introduced *Stegomalware*, an approach to hide executable application components within its resources such as audio files. Comparative analysis of state of the art application hardening approaches in recent years is illustrated in Table 6

7. Discussion and directions for future works

In summary, we feel Android application hardening methods are effective against reverse engineering. Malware detection studies have shown that code obfuscation hinders the analysis process. Multiple studies about the effectiveness of commercial anti-malware engines have been tested against obfuscation and evasion based methods. As mentioned in Section 1, our investigation has yielded an enumeration of challenges to address. Below we provide some direction for future work in the field of Android application hardening and their detection.

1. Trivial APK techniques being non-complex and incurring low overhead are popular among developers. Machine learning based approaches are likeable in future to ensure scalable detection of them. However, feature extraction and selection for learning must be carefully crafted to be truly representative.
2. Combination of code obfuscation techniques results in added space and code complexity. Thus finding the optimal obfuscation level and combinations which are efficient against overhead requires further consideration.
3. A challenge for researchers is the lack of benchmark or standard methodology for evaluating the effectiveness and efficiency of code obfuscation transformations. We conceive the existence of publicly available test datasets for researchers to benchmark their methods. And present datasets should expand to include new samples on regular basis.
4. Third party libraries often used for location, networking, advertising and other services are pervasively integrated with applications. A scalable third party library search can be modeled to identify application semantics in future.
5. More recent malware have introduced runtime based obfuscation (using reflection and native code) to subvert Android Runtime. With the history of native code obfuscation effectiveness on x86, we foresee runtime obfuscation to be more relevant for Android code obfuscation.
6. Polymorphic and metamorphic code obfuscations are successful in x86 architecture but are not explored by researchers for Android. Dynamic analysis approaches being used against obfuscation methods can be mitigated by

Table 6
Survey on the state-of-the-art android malware hardening approaches.

Year	Author	RP	CDO	VDO	CLO	AT	AH	AD	RD	AE	DB	AK	AS	NW	Objective
2017	T. Cho et al. [94]			✓	✓										A
2017	A. Arora and S. Peddoju [188]													✓	D
2017	M. Li et al. [120]				✓										D
2017	M. Preda and F. Maggi [13]	✓	✓	✓	✓										E
2017	G. Tangil et al. [161]				✓										D
2017	A. Continella et al. [191]			✓	✓									✓	D
2017	L. Vu et al. [151]								✓						D,E
2018	S. Dong et al. [31]		✓	✓	✓										A
2018	V. Haupert et al. [11]		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		D,E
2018	M. Wong and D. Lie [192]			✓	✓										E
2018	Y. Wang et al. [121]			✓	✓										D
2018	J. Garcia et al. [187]		✓	✓	✓										D
2018	Y. Moses [71]			✓	✓										D
2018	H. Cho et al. [42]		✓	✓	✓	✓									D
2018	A. Bacci et al. [14]		✓	✓	✓										D
2018	N. Totosis et al. [125]						✓								D
2019	O. Mirzaei et al. [104]			✓	✓										D
2019	Z. Kan et al. [98]				✓										D
2019	M. Ikram et al. [193]			✓	✓										D
2019	L. Li et al. [40]	✓													D
2019	J. Zhang et al. [194]				✓										D
2019	A. Ali and F. Faghih [195]			✓	✓										D
2019	M. Park et al. [196]			✓	✓										D
2019	B. Kim et al. [197]	✓													D
2019	Z. Li et al. [198]			✓	✓										D
2019	X. Yang et al. [199]				✓										E
2020	S. Aonzo et al. [200]	✓	✓	✓	✓										E
2020	L.Glanz et al. [201]			✓	✓										A

R: Repackaging, CDO: Constant Data Obfuscation, VDO: Variable Data Obfuscation, CLO: Control Logic Obfuscation, AT: Anti Tampering, AH: Anti Hooking, AD: Anti Debugging, RD: Root Detection, AE: Anti Emulator, DB: Device Binding, AS: Anti Screen reader, AK: Anti Key, NW: Network, A: Assessment, D: Detection and E: Evasion.

polymorphic code which differs on each execution. For a self modifying code, it needs to have a runtime code transformer.

7. Work on formal analysis of Android hardening is rarely available in the literature, which may be exploited for presenting mathematical solutions in future.
8. Researchers emulating Android environment faces challenges implementing numerous sensors embedded in real devices (such as camera, GPS, thermometer, WiFi, cellular, bluetooth, accelerometer, proximity sensors are a few). Generating real-like sensor patterns for emulating sensor readings is a challenge due to their complex interactions also.
9. Game theory based formalization of malware and analysis system model can be helpful in strengthening both detection and evasion aspects. Malware can be formulated to follow evasion, while the analysis system to detect it.
10. Device binding on a few occasions can be evaded by using a registered cloned app. Aggressive signature based on multiple host values is an approach to mitigate evasion. Another approach can be an asymmetric key based solution generated in a trusted environment for client identity binding.
11. Hardening of network traffic analysis (e.g., encryption) is a common approach amongst malware. It is imperative to improve encrypted network traffic analysis based malware identification using methods like deep packet inspection.
12. Presence and identification of obfuscation or analysis environment are increasingly getting popular amongst researchers. Side-channel based attacks against hardening techniques are still to be explored for Android, as has been the case with PCs.

8. Conclusion

Android being the most popular smartphone OS is available on various smart devices. Apps installed on these smart terminal are on the rise. Thus, attracting malware authors and researchers alike with increased research in reversing techniques. Through surveying the collected literature this paper follows a literature review process, first to conduct a survey of the existing obfuscation and preventive techniques used in the state of the art literature. Second, to illustrate current security services and framework in Android OS. Third, to assess the effectiveness of obfuscation techniques and hardening tools. Fourth, based on the survey, we highlight the issues of existing approaches related to them. Finally, we summarize the trends in application hardening and provide research gap for future works to present a complete picture. We conceive this work as a complement to existing works by filling research gaps and presenting future directions. We trust this survey will cater researchers to identify desirable hardening techniques and their respective analysis approaches.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Mobile operating systems' market share worldwide from january 2012 to december 2019, 2020, [Accessed: 09-Apr-2020]. URL https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.
[2] Number of apps available in leading app stores as of 4th quarter 2019, 2020, [Accessed: 09-Apr-2020]. URL https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.
[3] N. Grover, J. Saxena, V. Sihag, Security analysis of onlinecabbooking android application, in: Proceedings of the International Conference on Data Engineering and Communication Technology, Springer, 2017, pp. 603–611.
[4] Kaspersky Team, Malicious android app had more than 100 million downloads in google play, 2020, [Accessed: 09-Apr-2020]. URL https://www.kaspersky.com/blog/camscanner-malicious-android-app/28156.

[5] V. Rastogi, Y. Chen, X. Jiang, Catch me if you can: Evaluating android anti-malware against transformation attacks, IEEE Trans. Inf. Forensics Secur. 9 (1) (2013) 99–108.
[6] A. Kovacheva, Efficient code obfuscation for android, in: International Conference on Advances in Information Technology, Springer, 2013, pp. 104–119.
[7] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, M. Gaur, Android code protection via obfuscation techniques: past, present and future directions, 2016, CoRR abs/1611.10231.
[8] S. Banescu, A. Pretschner, A tutorial on software obfuscation, in: Advances in Computers, Vol. 108, Elsevier, 2018, pp. 283–353.
[9] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, T. Holz, Evaluating analysis tools for android apps: Status quo and robustness against obfuscation, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 139–141.
[10] P. Faruki, A. Bharmal, V. Laxmi, M.S. Gaur, M. Conti, M. Rajarajan, Evaluation of android anti-malware techniques against dalvik bytecode obfuscation, in: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2014, pp. 414–421.
[11] V. Haupert, D. Maier, N. Schneider, J. Kirsch, T. Müller, Honey, i shrunk your app security: The state of android app hardening, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2018, pp. 69–91.
[12] J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, M. Park, Effects of code obfuscation on android app similarity analysis, JoWUA 6 (4) (2015) 86–98.
[13] M. Dalla Preda, F. Maggi, Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology, J. Comput. Virol. Hacking Tech. 13 (3) (2017) 209–232.
[14] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, Detection of obfuscation techniques in Android applications, in: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018, pp. 1–9.
[15] F. Wei, Y. Li, S. Roy, X. Ou, W. Zhou, Deep ground truth analysis of current android malware, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 252–276.
[16] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, Malware dynamic analysis evasion techniques: A survey, ACM Comput. Surv. 52 (6) (2019) http://dx.doi.org/10.1145/3365001.
[17] A. Bulazel, B. Yener, A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web, in: Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium, 2017, pp. 1–21.
[18] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Comput. Surv. 49 (4) (2017) 1–41.
[19] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, et al., Toward engineering a secure android ecosystem: A survey of existing techniques, ACM Comput. Surv. 49 (2) (2016) 1–47.
[20] D. Maiorca, D. Ariu, I. Corona, M. Aresu, G. Giacinto, Stealth attacks: An extended insight into the obfuscation effects on android malware, Comput. Secur. 51 (2015) 16–31.
[21] A. Apvrille, R. Nigam, Obfuscation in android malware, and how to fight back, Virus Bull. (2014) 1–10.
[22] F.C. Freiling, M. Protsenko, Y. Zhuang, An empirical evaluation of software obfuscation techniques applied to android apks, in: International Conference on Security and Privacy in Communication Networks, Springer, 2014, pp. 315–328.
[23] Android, Selinux concepts, 2018, [Accessed: 09-Apr-2020]. URL https://source.android.com/security/selinux/concepts.
[24] Android security features, 2020, [Accessed: 09-Apr-2020]. URL https://source.android.com/security/features.
[25] S. Feldman, D. Stadther, B. Wang, Manilyzer: automated android malware detection through manifest analysis, in: 2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems, IEEE, 2014, pp. 767–772.
[26] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, K.-P. Wu, Droidmat: Android malware detection through manifest and api calls tracing, in: 2012 Seventh Asia Joint Conference on Information Security, IEEE, 2012, pp. 62–69.
[27] Android, Android developer, 2020, [Accessed: 09-Apr-2020]. URL https://developer.android.com.
[28] J. Song, C. Han, K. Wang, J. Zhao, R. Ranjan, L. Wang, An integrated static detection and analysis framework for android, Pervasive Mob. Comput. 32 (2016) 15–25.
[29] K. Xu, Y. Li, R.H. Deng, Iccdetector: Icc-based malware detection on android, IEEE Trans. Inf. Forensics Secur. 11 (6) (2016) 1252–1264.
[30] A. Feizollah, N.B. Anuar, R. Salleh, G. Suarez-Tangil, S. Furnell, Androdialysis: Analysis of android intent effectiveness in malware detection, Comput. Secur. 65 (2017) 121–134.
[31] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, K. Zhang, Understanding android obfuscation techniques: A large-scale investigation in the wild, in: International Conference on Security and Privacy in Communication Systems, Springer, 2018, pp. 172–192.
[32] J. Crussell, C. Gibler, H. Chen, Attack of the clones: Detecting cloned applications on android markets, in: European Symposium on Research in Computer Security, Springer, 2012, pp. 37–54.
[33] H. Huang, S. Zhu, P. Liu, D. Wu, A framework for evaluating mobile app repackaging detection algorithms, in: International Conference on Trust and Trustworthy Computing, Springer, 2013, pp. 169–186.
[34] H. Chang, M.J. Atallah, Protecting software code by guards, in: ACM Workshop on Digital Rights Management, Springer, 2001, pp. 160–175.
[35] J. Crussell, C. Gibler, H. Chen, Scalable semantics-based detection of similar android applications, in: Proc. of ESORICS, Vol. 13, Citeseer, 2013.
[36] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: 2012 IEEE Symposium on Security and Privacy, IEEE, 2012, pp. 95–109.
[37] R. Xu, H. Saïdi, R. Anderson, Aurasium: Practical policy enforcement for android applications, in: Presented As Part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012, pp. 539–552.
[38] B. Gruver, An assembler(smali) and disassembler(baksmali) for androids dex format, 2020, [Accessed: 09-Apr-2020]. URL https://github.com/JesusFreke/smali.
[39] J.-H. Jung, J.Y. Kim, H.-C. Lee, J.H. Yi, Repackaging attack on android banking applications and its countermeasures, Wirel. Pers. Commun. 73 (4) (2013) 1421–1437.
[40] L. Li, T.F. Bissyandé, J. Klein, Rebooting research on detecting repackaged android apps: Literature review and benchmark, IEEE Trans. Softw. Eng. (2019).
[41] L. Luo, Y. Fu, D. Wu, S. Zhu, P. Liu, Repackage-proofing android apps, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2016, pp. 550–561.
[42] H. Cho, J.H. Yi, G.-J. Ahn, Dexmonitor: Dynamically analyzing and monitoring obfuscated android applications, IEEE Access 6 (2018) 71229–71240.
[43] P. Schulz, F. Matenaar, Android Reverse Engineering and Defenses, Bluebox Labs, 2013.
[44] C. Collberg, C. Thomborson, D. Low, Manufacturing cheap, resilient, and stealthy opaque constructs, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, pp. 184–196.
[45] M. Musale, T.H. Austin, M. Stamp, Hunting for metamorphic javascript malware, J. Comput. Virol. Hacking Tech. 11 (2) (2015) 89–102.
[46] S.M. Sridhara, M. Stamp, Metamorphic worm that carries its own morphing engine, J. Comput. Virol. Hacking Tech. 9 (2) (2013) 49–58.
[47] Y. Piao, J.-H. Jung, J.H. Yi, Server-based code obfuscation scheme for apk tamper detection, Secur. Commun. Netw. 9 (6) (2016) 457–467.
[48] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, On the (im) possibility of obfuscating programs, in: Annual International Cryptology Conference, Springer, 2001, pp. 1–18.
[49] M. Dalla Preda, R. Giacobazzi, Semantics-based code obfuscation by abstract interpretation, J. Comput. Secur. 17 (6) (2009) 855–908.
[50] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, B. Waters, Candidate indistinguishability obfuscation and functional encryption for all circuits, SIAM J. Comput. 45 (3) (2016) 882–929.
[51] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, A. Pretschner, Code obfuscation against symbolic execution attacks, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 189–200.
[52] C. Collberg, C. Thomborson, D. Low, A Taxonomy of Obfuscating Transformations, Technical Report 148, Department of Computer Science, University of Auckland, 1997.
[53] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, E. Weippl, Protecting software through obfuscation: Can it keep pace with progress in code analysis?, ACM Comput. Surv. 49 (1) (2016) 1–37.
[54] Y. Zhou, A. Main, Y.X. Gu, H. Johnson, Information hiding in software with mixed boolean-arithmetic transforms, in: International Workshop on Information Security Applications, Springer, 2007, pp. 61–75.
[55] S. Chow, P. Eisen, H. Johnson, P.C. Van Oorschot, White-box cryptography and an aes implementation, in: International Workshop on Selected Areas in Cryptography, Springer, 2002, pp. 250–270.
[56] A. Anand, Securing android code using white box cryptography and obfuscation technique, Int. J. Comput. Sci. Mob. Comput. 4 (4) (2015) 347–352.
[57] V. Sánchez Ballabriga, Automation of White-Box Cryptography Attacks in Android Applications, Universitat Oberta de Catalunya (UOC), 2018.
[58] B. Wyseur, W. Michiels, P. Gorissen, B. Preneel, Cryptanalysis of white-box des implementations with arbitrary external encodings, in: International Workshop on Selected Areas in Cryptography, Springer, 2007, pp. 264–277.

[59] S. Chow, P. Eisen, H. Johnson, P.C. Van Oorschot, A white-box des implementation for drm applications, in: ACM Workshop on Digital Rights Management, Springer, 2002, pp. 1–15.
[60] H.E. Link, W.D. Neumann, Clarifying obfuscation: improving the security of white-box des, in: International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II, Vol. 1, IEEE, 2005, pp. 679–684.
[61] J. Bringer, H. Chabanne, E. Dottax, White box cryptography: Another attempt, IACR Cryptol. ePrint Arch. 2006 (2006) (2006) 468.
[62] Y. Xiao, X. Lai, A secure implementation of white-box aes, in: 2009 2nd International Conference on Computer Science and Its Applications, IEEE, 2009, pp. 1–6.
[63] M. Karroumi, Protecting white-box aes with dual ciphers, in: International Conference on Information Security and Cryptology, Springer, 2010, pp. 278–291.
[64] Paladion, Code obfuscation - part 2: Obfuscating data structures, 2015, [Accessed: 09-Apr-2020]. URL https://www.paladion.net/blogs/code-obfuscation-part-2-obfuscating-data-structures.
[65] B. Saikoa, Allatori java obfuscator, 2020, [Accessed: 09-Apr-2020]. URL http://www.allatori.com/.
[66] B. Saikoa, DexGuard.
[67] L. Licel, Dexprotector–cutting edge obfuscator for android apps, 2020, [Accessed: 09-Apr-2020]. URL https://dexprotector.com/.
[68] L. Licel, Stringer JAVA obfuscator, 2020, [Accessed: 09-Apr-2020]. URL https://jfxstore.com/stringer/.
[69] Z. Tang, X. Chen, D. Fang, F. Chen, Research on java software protection with the obfuscation in identifier renaming, in: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), IEEE, 2009, pp. 1067–1071.
[70] M. Kühnel, M. Smieschek, U. Meyer, Fast identification of obfuscation and mobile advertising in mobile malware, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 1, IEEE, 2015, pp. 214–221.
[71] Y. Moses, Y. Mordekhay, Android app deobfuscation using static-dynamic cooperation, 2018, VB2018.
[72] Guard Square, Proguard, 2020, [Accessed: 09-Apr-2020]. URL https://www.guardsquare.com/en/products/proguard.
[73] M. Dalla Preda, Code Obfuscation and Malware Detection by Abstract Interpretation, (PhD diss.), Citeseer, 2007, http://profs.sci.univr.it/dallapre/MilaDallaPreda_PhD.pdf.
[74] D. Low, Java Control Flow Obfuscation, (Ph.D. thesis), Citeseer, 1998.
[75] V. Sihag, A. Mitharwal, M. Vardhan, P. Singh, Opcode n-gram based malware classification in android, in: 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), IEEE, 2020, pp. 645–650.
[76] C. Collberg, C. Thomborson, D. Low, Breaking abstractions and unstructuring data structures, in: Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225), IEEE, 1998, pp. 28–38.
[77] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, B. Waters, Candidate indistinguishability obfuscation and functional encryption for all circuits, SIAM J. Comput. 45 (3) (2016) 882–929.
[78] T.-W. Hou, H.-Y. Chen, M.-H. Tsai, Three control flow obfuscation methods for java software, IEE Proc.-Softw. 153 (2) (2006) 80–86.
[79] M. Dalla Preda, R. Giacobazzi, Control code obfuscation by abstract interpretation, in: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05), IEEE, 2005, pp. 301–310.
[80] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, Y. Zhang, Experience with software watermarking, in: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00), IEEE, 2000, pp. 308–316.
[81] A. Majumdar, C. Thomborson, Manufacturing opaque predicates in distributed systems for code obfuscation, in: Proceedings of the 29th Australasian Computer Science Conference-Volume 48, Australian Computer Society, Inc., 2006, pp. 187–196.
[82] F.B. Cohen, Operating system protection through program evolution., Comput. Secur. 12 (6) (1993) 565–584.
[83] I. You, K. Yim, Malware obfuscation techniques: A brief survey, in: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, IEEE, 2010, pp. 297–300.
[84] C.S. Collberg, C.D. Thomborson, D.W.K. Low, Obfuscation Techniques for Enhancing Software Security, Google Patents, 2003, US Patent 6, 668, 325.
[85] Android Studio, Shrink, obfuscate, and optimize your app, 2020, [Accessed: 09-Apr-2020]. URL https://developer.android.com/studio/build/shrink-code.
[86] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Comput. Surv. 26 (4) (1994) 345–420.
[87] Z. Vrba, Cryptexec: Next-generation runtime binary encryption using on-demand function extraction, 2003, Phrak 0x0b (0x3f).# 0x0d of 0x14.[online] http://www.phrack.org/archives/63/p63-0x0d_Next_Generation_Runtime_Binary_Encryption.txt.
[88] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, K. De Bosschere, Towards tamper resistant code encryption: Practice and experience, in: International Conference on Information Security Practice and Experience, Springer, 2008, pp. 86–100.
[89] M.F. Oberhumer, Upx the ultimate packer for executables, 2004, http://upx.sourceforge.net/.
[90] Y. Aafer, W. Du, H. Yin, Droidapiminer: Mining api-level features for robust malware detection in android, in: International Conference on Security and Privacy in Communication Systems, Springer, 2013, pp. 86–103.
[91] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, C. Siemens, Drebin: Effective and explainable detection of android malware in your pocket, in: Ndss, Vol. 14, 2014, pp. 23–26.
[92] M. Lindorfer, M. Neugschwandtner, C. Platzer, Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis, in: 2015 IEEE 39th Annual Computer Software and Applications Conference, Vol. 2, IEEE, 2015, pp. 422–433.
[93] Z. Cai, R.H. Yap, Inferring the detection logic and evaluating the effectiveness of android anti-virus apps, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 172–182.
[94] T. Cho, H. Kim, J.H. Yi, Security assessment of code obfuscation based on dynamic monitoring in android things, IEEE Access 5 (2017) 6361–6371.
[95] Symantec Security Response, Server-side polymorphic android applications, 2020, [Accessed: 09-Apr-2020]. URL https://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.
[96] C. Wang, J. Davidson, J. Hill, J. Knight, Protection of software-based survivability mechanisms, in: 2001 International Conference on Dependable Systems and Networks, IEEE, 2001, pp. 193–202.
[97] S. Chow, Y. Gu, H. Johnson, V.A. Zakharov, An approach to the obfuscation of control-flow of sequential computer programs, in: International Conference on Information Security, Springer, 2001, pp. 144–155.
[98] Z. Kan, H. Wang, L. Wu, Y. Guo, G. Xu, Deobfuscating android native binary code, in: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, in: ICSE '19, IEEE Press, 2019, pp. 322–323, http://dx.doi.org/10.1109/ICSE-Companion.2019.00135.
[99] S.K. Udupa, S.K. Debray, M. Madou, Deobfuscation: Reverse engineering obfuscated code, in: 12th Working Conference on Reverse Engineering (WCRE'05), IEEE, 2005, pp. 10–pp.
[100] K.Z. Chen, N.M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T.R. Magrino, E.X. Wu, M. Rinard, D.X. Song, Contextual policy enforcement in android applications with permission event graphs, in: NDSS, 2013, p. 234.
[101] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, W. Zou, Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications, in: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 2012, pp. 93–104.
[102] L.K. Yan, H. Yin, Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis, in: Presented As Part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012, pp. 569–584.
[103] A. Amamra, C. Talhi, J.-M. Robert, Smartphone malware detection: From a survey towards taxonomy, in: 2012 7th International Conference on Malicious and Unwanted Software, IEEE, 2012, pp. 79–86.
[104] O. Mirzaei, J.M. de Fuentes, J. Tapiador, L. Gonzalez-Manzano, Androdet: An adaptive android obfuscation detector, Future Gener. Comput. Syst. 90 (2019) 240–261.
[105] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, G. Vigna, Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy, in: The Network and Distributed System Security Symposium, 2016, pp. 1–15.
[106] S. Alam, Z. Qu, R. Riley, Y. Chen, V. Rastogi, Droidnative, Comput. Secur. 65 (C) (2017) 230–246, http://dx.doi.org/10.1016/j.cose.2016.11.011.
[107] M. Sun, G. Tan, Nativeguard: Protecting android applications from third-party native libraries, in: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, 2014, pp. 165–176.
[108] K. Lu, Deep analysis of android rootnik malware using advanced anti-debug and anti-hook, 2017, [Accessed: 09-Apr-2020]. URL https://www.fortinet.com/blog/threat-research/deep-analysis-of-android-rootnik-malware-using-advanced-anti-debug-and-anti-hook-part-i-debugging-in-the-scope-of-native-layer.html.
[109] Mobile Threat Response Team, Zniu: First android malware to exploit dirty cow vulnerability, 2017, [Accessed: 09-Apr-2020]. URL https://blog.trendmicro.com/trendlabs-security-intelligence/zniu-first-android-malware-exploit-dirty-cow-vulnerability/.
[110] M. Stone, Unpacking the packed unpacker: Reversing an android anti-analysis native library.
[111] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin, Obfuscator-llvm–software protection for the masses, in: 2015 IEEE/ACM 1st International Workshop on Software Protection, IEEE, 2015, pp. 3–9.

[112] M. Backes, S. Bugiel, E. Derr, Reliable third-party library detection in android and its security applications, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 356–367.
[113] K. Chen, P. Liu, Y. Zhang, Achieving accuracy and scalability simultaneously in detecting application clones on android markets, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 175–186.
[114] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, W. Zou, Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios, in: 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 357–376.
[115] J. Crussell, C. Gibler, H. Chen, Andarwin: Scalable detection of android application clones based on semantics, IEEE Trans. Mob. Comput. 14 (10) (2014) 2007–2019.
[116] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, M. Mezini, CodeMatch: obfuscation won't conceal your repackaged app, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 638–648.
[117] M.C. Grace, W. Zhou, X. Jiang, A.-R. Sadeghi, Unsafe exposure analysis of mobile in-app advertisements, in: Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, 2012, pp. 101–112.
[118] Z. Ma, H. Wang, Y. Guo, X. Chen, LibRadar: fast and accurate detection of third-party libraries in Android apps, in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp.653–656.
[119] H. Wang, Y. Guo, Z. Ma, X. Chen, Wukong: A scalable and accurate two-phase approach to android app clone detection, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 71–82.
[120] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo, Libd: scalable and precise third-party library detection in android markets, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 335–346.
[121] Y. Wang, H. Wu, H. Zhang, A. Rountev, Orlis: Obfuscation-resilient library detection for android, in: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2018, pp. 13–23.
[122] C. Ren, K. Chen, P. Liu, Droidmarking: resilient software watermarking for impeding android application repackaging, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014, pp. 635–646.
[123] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, X. Su, Identifying and understanding self-checksumming defenses in software, in: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, 2015, pp. 207–218.
[124] Intertrust Secure Systems, Application shielding with intertrust's white-cryption code protection, 2018, [Accessed: 09-Apr-2020]. URL https://www.infosecurityeurope.com/__novadocuments/594809.
[125] N. Totosis, C. Patsakis, Android hooking revisited, in: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), IEEE, 2018, pp. 552–559.
[126] D. Maier, T. Müller, M. Protsenko, Divide-and-conquer: Why android malware cannot be stopped, in: 2014 Ninth International Conference on Availability, Reliability and Security, IEEE, 2014, pp. 30–39.
[127] T. Vidas, N. Christin, Evading android runtime analysis via sandbox detection, in: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, 2014, pp. 447–458.
[128] L. SaurikIT, Cydia substrate, the powerful code modification platform behind cydia, 2016.
[129] X. Framework, The xposed framework source code, 2017.
[130] J. Lopes, Who Owns Your Runtime?, Nettitude Labs, 2017, URL https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/.
[131] H. Cho, J. Lim, H. Kim, J.H. Yi, Anti-debugging scheme for protecting mobile apps on android platform, J. Supercomput. 72 (1) (2016) 232–246.
[132] S. Cesare, Linux anti-debugging techniques, in: Security Focus, 1999.
[133] M.N. Gagnon, S. Taylor, A.K. Ghosh, Software protection through anti-debugging, IEEE Secur. Priv. 5 (3) (2007) 82–84.
[134] Amat: Android malware analysis toolkit, 2020, [Accessed: 09-Apr-2020]. URL http://sourceforge.net/projects/amatlinux/.
[135] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, C. Platzer, Andrubis: Android Malware Under the Magnifying Glass, Tech. Rep. TR-ISECLAB-0414-001, Vienna University of Technology, 2014, pp. 1–10.
[136] K. Tam, S.J. Khan, A. Fattori, L. Cavallaro, Copperdroid: Automatic reconstruction of android malware behaviors, in: Ndss, 2015.
[137] A. Desnos, P. Lantz, Droidbox: an Android Application Sandbox for Dynamic Analysis, Tech. Rep, Lund Univ., 2011.
[138] Botnet Research Team and others, SandDroid: An Apk Analysis Sandbox, Xi'an jiaotong university, 2014.
[139] T. Bläsing, L. Batyuk, A.-D. Schmidt, S.A. Camtepe, S. Albayrak, An android application sandbox system for suspicious software detection, in: 2010 5th International Conference on Malicious and Unwanted Software, IEEE, 2010, pp. 55–62.
[140] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, S. Ioannidis, Rage against the virtual machine: hindering dynamic analysis of android malware, in: Proceedings of the Seventh European Workshop on System Security, 2014, pp. 1–6.
[141] P. Ferrie, Attacks on more virtual machine emulators. Symantec technology exchange 2007, 2017.
[142] R. Paleari, L. Martignoni, G.F. Roglia, D. Bruschi, A fistful of red-pills: How to automatically generate procedures to detect CPU emulators, in: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT), Vol. 41, 2009, pp. 86.
[143] J. Rutkowska, Redpill: Detect vmm using (almost) one cpu instruction, 2004, http://invisiblethings.org/papers/redpill.html.
[144] Y. Jing, Z. Zhao, G.-J. Ahn, H. Hu, Morpheus: automatically generating heuristics to detect android emulators, in: Proceedings of the 30th Annual Computer Security Applications Conference, 2014, pp. 216–225.
[145] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, G. Vigna, Exploitation and mitigation of authentication schemes based on device-public information, in: Proceedings of the 33rd Annual Computer Security Applications Conference, 2017, pp. 16–27.
[146] V. Haupert, T. Müller, On app-based matrix code authentication in online banking, in: ICISSP, 2018, pp. 149–160.
[147] Cecilia, 80% China's mobile users rooted smartphones in 2014, 2015, [Accessed: 09-Apr-2020]. URL https://www.chinainternetwatch.com/12926/80-china-smartphone-users-rooted/.
[148] A.P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, in: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 2011, pp. 3–14.
[149] D.R. Thomas, A.R. Beresford, A. Rice, Security metrics for the android ecosystem, in: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, 2015, pp. 87–98.
[150] S.-T. Sun, A. Cuadros, K. Beznosov, Android rooting: Methods, detection, and evasion, in: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, 2015, pp. 3–14.
[151] L. Nguyen-Vu, N.-T. Chau, S. Kang, S. Jung, Android rooting: An arms race between evasion and detection, Secur. Commun. Netw. 2017 (2017).
[152] V. Sihag, A. Swami, M. Vardhan, P. Singh, Signature based malicious behavior detection in android, in: International Conference on Computing Science, Communication and Security, Springer, 2020, pp. 251–262.
[153] E.J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: 2010 IEEE Symposium on Security and Privacy, IEEE, 2010, pp. 317–331.
[154] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N. Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, ACM Trans. Comput. Syst. (TOCS) 32 (2) (2014) 1–29.
[155] G. Sarwar, O. Mehani, R. Boreli, M.A. Kaafar, On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices, in: SECRYPT, Vol. 96435, 2013.
[156] Y. Fratantonio, C. Qian, S.P. Chung, W. Lee, Cloak and dagger: from two permissions to complete control of the ui feedback loop, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 1041–1057.
[157] Android Developer, Display, 2020, [Accessed: 09-Apr-2020]. URL https://developer.android.com/reference/android/view/Display.html.
[158] Kolin Sturt, Securing communications on android, 2018, [Accessed: 09-Apr-2020]. URL https://code.tutsplus.com/tutorials/securing-communications-on-android--cms-31596.
[159] P. Sharma, V.K. Sihag, Hybrid single sign-on protocol for lightweight devices, in: 2016 IEEE 6th International Conference on Advanced Computing (IACC), IEEE, 2016, pp. 679–684.
[160] G. Suarez-Tangil, J.E. Tapiador, F. Lombardi, R. Di Pietro, Alterdroid: differential fault analysis of obfuscated smartphone malware, IEEE Trans. Mob. Comput. 15 (4) (2015) 789–802.
[161] G. Suarez-Tangil, S.K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, L. Cavallaro, Droidsieve: Fast and accurate classification of obfuscated android malware, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017, pp. 309–320.
[162] R. Nigam, A timeline of mobile botnets, Virus Bull. (2015).
[163] M. Christodorescu, S. Jha, Testing malware detectors, ACM SIGSOFT Softw. Eng. Notes 29 (4) (2004) 34–44.

[164] M. Zheng, P.P. Lee, J.C. Lui, Adam: an automatic and extensible platform to stress test android anti-virus systems, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2012, pp. 82–101.
[165] S. Badhani, S.K. Muttoo, Analyzing android code graphs against code obfuscation and app hiding techniques, J. Appl. Secur. Res. 14 (4) (2019) 489–510.
[166] Apk protect: Android apk security protection, 2013, URL https://sourceforge.net/projects/apkprotect/.
[167] Baidu Inc., URL http://app.baidu.com.
[168] Bangcle Inc., URL http://www.bangcle.com.
[169] P. Solutions, Dasho: Java & android obfuscator & runtime protection.
[170] Transakt, Entersekt Mobile App Security, URL https://www.entersekt.com/.
[171] Ijiami Inc., URL http://www.ijiami.cn.
[172] Inside Secure Code & Application Protection, URL https://www.insidesecure.com/us/Products/Mobile-and-IoT-Security/Applications-Protection/Code-Protection.
[173] Mobile Protector by Gemalto, a Thales company, URL https://thales-protector-oath-sdk.docs.stoplight.io/releases/5.2.0/general/overview.
[174] Promon Shield | In-App Protection & Application Shielding, URL https://promon.co.
[175] SecNeo, The Professional service provider for the mobile application security, URL https://www.secneo.com/.
[176] Source Code Protection — whiteCryption. URL https://www.intertrust.com/products/application-shielding/code-protection/.
[177] yworks, Yguard - java bytecode obfuskator and shrinker, 2020, [Accessed: 09-Apr-2020]. URL https://www.yworks.com/products/yguard.
[178] H. Ohuchi, Jarg-java archiver grinder, 2003, http://jarg.sourceforge.net/index.en.
[179] J. Hoenicke, Jode, 2004, URL http://jode.sourceforge.net.
[180] W. Zhou, Z. Wang, Y. Zhou, X. Jiang, Divilar: Diversifying intermediate language for anti-repackaging on android platform, in: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, 2014, pp. 199–210.
[181] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, Google android: A comprehensive security assessment, IEEE Secur. Priv. 8 (2) (2010) 35–44.
[182] J. Shu, J. Li, Y. Zhang, D. Gu, Android app protection via interpretation obfuscation, in: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, IEEE, 2014, pp. 63–68.
[183] B. Bichsel, V. Raychev, P. Tsankov, M. Vechev, Statistical deobfuscation of android applications, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 343–355.
[184] G. Suarez-Tangil, G. Stringhini, Eight years of rider measurement in the android malware ecosystem, IEEE Trans. Dependable Secure Comput. (2020) 1.
[185] V. Balachandran, D.J. Tan, V.L. Thing, et al., Control flow obfuscation for android applications, Comput. Secur. 61 (2016) 72–93.
[186] Y. Wang, A. Rountev, Who changed you? Obfuscator identification for android, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2017, pp. 154–164.
[187] J. Garcia, M. Hammad, S. Malek, Lightweight, obfuscation-resilient detection and family identification of android malware, ACM Trans. Softw. Eng. Methodol. (TOSEM) 26 (3) (2018) 1–29.
[188] A. Arora, S.K. Peddoju, Minimizing network traffic features for android mobile malware detection, in: Proceedings of the 18th International Conference on Distributed Computing and Networking, 2017, pp. 1–10.
[189] F. Zhang, H. Huang, S. Zhu, D. Wu, P. Liu, ViewDroid: Towards obfuscation-resilient mobile application repackaging detection, in: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, 2014, pp. 25–36.
[190] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, Stegomalware: Playing hide and seek with malicious components in smartphone apps, in: International Conference on Information Security and Cryptology, Springer, 2014, pp. 496–515.
[191] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, G. Vigna, Obfuscation-resilient privacy leak detection for mobile apps through differential analysis, in: NDSS, 2017.
[192] M.Y. Wong, D. Lie, Tackling runtime-based obfuscation in android with {TIRO}, in: 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 1247–1262.
[193] M. Ikram., P. Beaume., M.A. Kaafar., Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling, in: Proceedings of the 16th International Joint Conference on E-Business and Telecommunications - Volume 2: SECRYPT, SciTePress, 2019, pp. 211–219, http://dx.doi.org/10.5220/0007834602110219.
[194] J. Zhang, A.R. Beresford, S.A. Kollmann, LibID: reliable identification of obfuscated third-party Android libraries, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 55–65.
[195] A. Aghamohammadi, F. Faghih, Lightweight versus obfuscation-resilient malware detection in android applications, J. Comput. Virol. Hacking Tech. (2019) 1–15.
[196] M. Park, G. You, S.-j. Cho, M. Park, S. Han, A framework for identifying obfuscation techniques applied to android apps using machine learning., J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 10 (4) (2019) 22–30.
[197] B. Kim, J. Jung, S. Han, S. Jeon, S.-j. Cho, J. Choi, A new technique for detecting android app clones using implicit intent and method information, in: 2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2019, pp. 478–483.
[198] Z. Li, J. Sun, Q. Yan, W. Srisa-an, Y. Tsutano, Obfusifier: Obfuscation-resistant android malware detection system, in: International Conference on Security and Privacy in Communication Systems, Springer, 2019, pp. 214–234.
[199] X. Yang, L. Zhang, C. Ma, Z. Liu, P. Peng, Android control flow obfuscation based on dynamic entry points modification, in: 2019 22nd International Conference on Control Systems and Computer Science (CSCS), IEEE, 2019, pp. 296–303.
[200] S. Aonzo, G.C. Georgiu, L. Verderame, A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for android apps, SoftwareX 11 (2020) 100403.
[201] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, M. Mezini, Hidden in plain sight: Obfuscated strings threatening your privacy, in: ASIA CCS '20, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3320269.3384745.