Hindawi
Security and Communication Networks
Volume 2018, Article ID 6280768, 12 pages
https://doi.org/10.1155/2018/6280768

Research Article

Static and Dynamic Analysis of Android Malware and Goodware Written with Unity Framework

Jaewoo Shim,[1] **Kyeonghwan Lim**,[1] **Seong-je Cho**,[1] **Sangchul Han**,[2] **and Minkyu Park**[2]

[1]*Department of Computer Science and Engineering, Dankook University, Yongin 16890, Republic of Korea*
[2]*Department of Computer Engineering, Konkuk University, Chungju 27478, Republic of Korea*

Correspondence should be addressed to Minkyu Park; minkyup@kku.ac.kr

Received 23 February 2018; Accepted 15 May 2018; Published 20 June 2018

Academic Editor: Karl Andersson

Unity is the most popular cross-platform development framework to develop games for multiple platforms such as Android, iOS, and Windows Mobile. While Unity developers can easily develop mobile apps for multiple platforms, adversaries can also easily build malicious apps based on the "*write once, run anywhere*" (WORA) feature. Even though malicious apps were discovered among *Android apps written with Unity framework (Unity apps)*, little research has been done on analysing the malicious apps. We propose static and dynamic reverse engineering techniques for malicious Unity apps. We first inspect the executable file format of a Unity app and present an effective static analysis technique of the Unity app. Then, we also propose a systematic technique to analyse dynamically the Unity app. Using the proposed techniques, the malware analyst can statically and dynamically analyse Java code, native code in C or C ++, and the Mono runtime layer where the C# code is running.

1. Introduction

The smartphone market is currently dominated by various mobile platforms such as Android, iOS, Windows Phone, and BlackBerry OS. Because these mobile platforms use different development frameworks, app developers who want to cover many users need to develop their native apps for each platform. A *native app* is an application built to run only on a certain mobile platform. Developing an app for each platform has disadvantages. Developers need to know in-depth knowledge of each platform, dedicated software development kits (SDKs), and different programming language. Developers cannot reuse the source codes of one platform for another platform and must write codes multiple times. The development and maintenance cost increases.

Several *cross-platform development frameworks* [1–4] have emerged to tackle these challenges. These development frameworks employ the "*write code once, run on multiple platforms*" feature: the frameworks translate codes for one platform to another platform and provide rum-time library supporting app execution. The frameworks reduce development and maintenance costs. Such frameworks include *Unity*, *Xamarin*, *Appcelerator Titanium*, *PhoneGap*, *Sencha*, and *Cocos2d*.

Among the frameworks, Unity (commonly known as Unity3D) is a very popular cross-platform framework for mobile app development [5–8]. According to the report from Unity Technologies [9], Unity currently supports 28 different platforms and touches 770 million gamers all over the world through games made using the engine. In Q1 2016, 34% of top 1000 free mobile games are developed and around 2.4 billion unique mobile devices have been running Unity-made games.

Adversaries also utilise the frameworks to make their malware spreadable easily while minimizing malware development cost and time. Lee et al. of ShophosLabs [10] reported that more and more malicious apps are being developed with the frameworks. They collected Android malware statistics for a couple of months in 2015 and observed that the number of malware and potentially unwanted app (PUA) samples written with Unity was 32 and 1351, respectively. PUA may pose high risk or have unwanted impact on user security and privacy. Also, according to the study performed by Kaspersky Lab, malware writers are eager to use Microsoft's .NET and

high-level programming languages such as C# to create and modify their malware more rapidly [11]. Adopting such convenient frameworks or tools, malware writers can gain the additional benefit of leveraging already available source code from the underground scene.

Each framework uses its own executable format and loads many libraries necessary for app execution. These are utilised to hide malicious payloads in the package. The malware developers can hide malicious codes in components (e.g., libraries) loaded by the framework as well as the platform's native codes (e.g., Java or Swift). Therefore, we need to analyse framework's components as well as native codes to detect malware [10].

We propose an effective technique to statically and dynamically analyse suspicious Android apps written in C# on Unity framework, which allows for (1) detection of instances of known malware and (2) early detection of known buggy or vulnerable code. We define an Android app written in C# on Unity framework as a Unity app or a Unity APK. We focus on analysing Unity apps which do not have debugging symbol and cannot be easily debugged. We first generate debugging symbols for the target app; repackage the app with the generated symbols; and run and debug it by attaching SDB (a client for the Mono soft debugger).

The rest of this paper is organised as follows. Section 2 explains background and related research work. Section 3 proposes an effective technique for statically analysing Unity apps and presents some results of the static analysis of a Unity app focusing on its initialisation to show the structure and interaction between object codes. Section 4 describes a dynamic analysis technique of a Unity app built in release mode which does not originally have debugging symbols. Section 5 discusses the proposed technique including the differences between native apps and Unity apps in terms of programming languages, library, and debugging tools and protocol. Conclusions are presented in Section 6.

2. Background and Related Work

2.1. Background. The Unity framework is built on Mono runtime which enables developers to use C# or UnityScript [7, 10–12]. The Mono project is an open source implementation of Microsoft's .NET Framework. The .NET Framework consists of three components: a set of supported programming languages, a base class library which implements all basic operations involved in software development, and the *Common Language Runtime* (CLR), which is the core of the framework [10].

CLR is also known as the Mono runtime and we use the terms CLR and Mono runtime interchangeably. CLR is designed to comply with a *common language infrastructure* (CLI), which is a specification for the format of executable code and the runtime environment that can execute that code. CLI is a platform-independent development framework that enables programs written in different languages to run on different types of hardware. No matter which language they are written in, CLI applications are compiled into an intermediate language (IL), which is further compiled into the target machine language by the CLR.

Unity apps are primarily developed using C# and Unity JavaScript (UnityScript). C# source codes are converted to a *Common Intermediate Language* (CIL) which is formerly known as *Microsoft Intermediate Language* (MSIL). CIL code is stored in the form of a DLL file. CIL code is assembled into CLI assembly code and this code will be translated to machine code at runtime using just-in-time (JIT) compilation by the CLR. Unity apps execution can be summarised as follows: (1) source code is converted to CIL, (2) CIL is assembled into a CLI assembly code, similar to Java's bytecode, (3) the CLI assembly code is compiled to machine code for a specific platform, and (4) the processor executes the machine code.

The Mono runtime provides various services such as code execution, garbage collection, code generation using JIT compilation and backend engine, operating system interface, program isolation using Application Domains (AppDomain), thread management, console access, and security system [12]. The Mono runtime also interacts with the existing Android execution environment, DVM (Dalvik Virtual Machine) or ART (Android Runtime).

2.2. Related Work. For program understanding or malware detection, many studies have been conducted on static or dynamic analysis techniques for various types of executable files (DEX, ELF, PE, etc.) on several processors and operating systems (Android, Linux, Microsoft Windows OSes, etc.). Static analysis examines the code of executables to determine control or data flows and certain code patterns of these executables without running them [13–18]. Static analysis can cover the complete program code because it is not bound to a specific execution of an executable and can give guarantees that apply to all execution of the executable. However, static analysis has limitations that it cannot effectively deal with deliberately crafted code with obfuscation and encryption schemes [19–21].

In contrast to static techniques, dynamic analysis monitors the execution of an executable by inspecting runtime behaviours that correspond to selected test cases [21–26]. Dynamic analysis can examine the behaviour of a suspicious program by executing the program in a restricted environment and observing its actions. This dynamic technique can handle packed executables and self-modifying programs and is even immune to code obfuscation attempts to obstruct analysis.

Some researchers have tried to detect malicious Android native apps which use evasion techniques such as various code obfuscations to avoid detection by anti-virus software [27–29]. As a result, several interesting studies [13–29] have been made of static and dynamic techniques for analysing native applications (PE, ELF, JavaScript, etc.) for Microsoft Windows and Linux, or Android malware (DEX), not Unity apps. However, there have been few studies on systematic analysis on Android apps written in C# with Unity.

As the cross-platform development frameworks become more and more popular [30–32], researchers are studying which frameworks and tools are efficient for developing

their mobile apps. Majchrzak et al. [31] performed a comprehensive analysis of the three approaches, React Native, the Ionic Framework, and Fuse using a real-world use case. The analysed results showed that there was no clear winner. Latif et al. [32] conducted an exhaustive survey of cross-platform approaches and tools, provided an overview of recent tools and platforms for each approach, and discussed the advantages and disadvantages of each one.

Cross-platform mobile app development frameworks allow app developers to specify the app's logic once using the programming language and APIs of a home platform and automatically generate versions of the app for other target platforms. It is very hard to develop such frameworks because they must correctly translate the home platform API to the target platform API while preserving the same behaviour and functionality. Inconsistencies and bugs may arise during the translation. To solve the problems, we need a kind of translation testing tool. For example, X-Checker [2] was designed and applied to test fidelity of Xamarin, a popular framework that enables Windows Phone apps to be cross-compiled into native Android apps. X-Checker generates randomized test cases for both platforms and detects inconsistencies by comparing both execution results. X-Checker had found 47 bugs, and 12 of them was fixed after reporting. The X-Checker's approach can be used to dynamically analyse Android apps.

Lee et al. [10] had observed an increase in the number of malware developed with cross-platform frameworks. They reported PhoneGap malware codes and found out that the pieces of malware conceal the malicious code in HTML files or containers loaded by cross-platform frameworks instead of the platform's native codes. They explained the app package structure for native Android and iOS platforms and then emphasized each cross-platform framework's characteristics including Unity application files. Their study analysed a *proof of concept* (POC) app that used cross-platform features to embed malicious code. The POC app was designed to include a malicious plug-in employed to conceal intended bad activities. By creating a Unity POC app, they showed that DLL files in the Unity app, both Android and iOS, shared the same compiled C# code in their package, and the C# code from the `Assembly-CSharp.dll` file invoked the plug-in APIs. Finally, they present a solution to identify an app's framework type and to write a detection signature for malware based on those frameworks.

Malware writers are eager to use the convenient tools such as Microsoft's .NET, high-level programming languages, and PowerSell frameworks to write fresh pieces of malware [11]. In addition, they have considered many measures to slow down malware analysis. For example, PowerSploit framework integrates a collection of PowerShell scripts and modules to be used in the post-exploitation stages of an attack. PowerSploit can execute the scripts to conduct low-level and administrative tasks without the need to drop malicious executables, aiming to avoid timely anti-virus detection. As another example, the Mono-developed projects can be ported from one platform to another using MoMA (Mono Migration Analyser) which gives malware writers a productivity boost in their malware lifecycle efforts. As mobile cross-platform frameworks are getting popular, app portability has also increased the number of pieces of malware that run on multiple platforms. To address these problems, Pontiroli et al. of Kaspersky Lab [11] have briefly reviewed how the .NET framework works and how a .NET PE is built to understand the differences in the analysis of .NET assemblies. For malware analysts, a set of tools including ILSpy decompiler is required to understand .NET malware, and a standardized process that fits their needs.

Dalmasso et al. [33] carried out a survey of "Write Once Run Anywhere" (WORA) tools along with a classification and comparison among the tools. They examined the performance of several cross-platform frameworks in terms of CPU, memory usage, and power consumption by developing Android test apps. They also compared a cross-platform app development approach with a native app development approach by using the nine decision criteria including "security of app". According to their research results, apps developed with cross-platform frameworks are not highly secure while security of the apps developed in a native approach is excellent. They point out that proper research needs to be carried out to secure the tools and apps.

Mylonas et al. [34, 35] evaluated comparatively the security level of popular smartphone platforms, considering their protection against simple malicious apps. They also examined the feasibility and easiness of writing malware by average programmers who could access the official tools and libraries supplied by smartphone platforms. According to their case study findings, (1) the Android malware was developed by the B.S. student in one day using the official development toolkit (SDK), Java, and the documentation of its API, (2) the BlackBerry malware was developed by the B.S. student in one day using RIM's SDK, Java, and the API documentation, (3) the iOS malware was developed by the M.S. student in seven days using Apple's SDK, Objective C, and the API documentation (in iOS case study, the malware developer had no prior experience with Objective C), and (4) on a Windows Mobile 6 and a Windows Phone 7, the malicious app was developed by the B.S. student in 2 days and in one day, respectively, using Microsoft SDK, C#, and the API documentation. Based on this proof of concept study, they had proven that, under circumstances, all examined platforms could be used by average programmers as privacy attack vectors, harvesting data from the smartphone without the users knowledge and consent.

Chen et al. [36] conducted a systematic study on potentially-harmful libraries (PhaLibs) across Android and iOS by observing that many iOS libraries had Android counterparts which could be used to understand their behaviours and the relations between the libraries on both sides. They first clustered similar packages from lots of popular Android apps to identify libraries and analysed the libraries using anti-virus (AV) systems to detect PhaLibs. Then, these libraries were mapped to the code components within iOS apps based on the invariant features shared across two platforms. They examined that 36 of top 38 iOS libraries have Android versions. The top 38 libraries include Unity, Cordova, PhoneGap, and Appcelerator Titanium.

Table 1: Common library files of Unity apps.

Filename	Description
`Assembly-CSharp.dll`	CIL codes generated by compiling developer's C# codes
`Assembly-CSharp-firstpass.dll`	
`System.*.dll`	File I/O support
`UnityEngine.dll`	Unity API support
`libmain.so`	Load and run `libmono.so` and `libunity.so` (redefine load/unload method)
`libmono.so`	Load classes in `Assembly-CSharp.dll` onto Mono runtime
`libunity.so`	Create Unity engine and Mono runtime

HTML5-based mobile apps and JavaScript apps are widely used because they are much easier to be ported across different mobile platforms. This flexibility of the JavaScript makes such apps prone to code injection attacks [37, 38]. Jin et al. [37] showed how HTML5-based apps could be vulnerable, how adversaries could exploit the vulnerabilities, and which damage could be incurred by the attacks. They also demonstrated the attacks using example apps and found that 11 PhoneGap plugins were vulnerable. Mao et al. [38] recently presented a technique to identify malicious behaviours in JavaScript apps and detect injection attacks to the apps and evaluated the effectiveness of their technique.

3. Static Analysis of Unity Apps

Unity is based on Mono project which is a cross-platform, open source .NET framework. It includes C# Compiler, Mono Runtime, .NET Framework Class Library, Mono Class Library, and APIs for each mobile environment. Unity users can develop mobile applications by writing programs in C# language and building them for any target device environment.

To support multiple platforms effectively, Unity apps deploy various types of object codes. For example, C# based Android apps developed with Unity contain Dalvik bytecode in `*.dex` files, .NET framework CIL code in DLL files and machine code in `*.so` files. In this section, we present a technique for static analysis of Unity apps, which includes disassembling or decompilation of these object codes. And we also present some results of the static analysis of an app we downloaded from the market to show the structure and interaction between object codes of Unity apps.

3.1. Structure of Unity Apps. We download a Unity app (Arrow.io) from the market and inspect its APK file. Figure 1 illustrates the structure of the APK file. The structure of Unity app is similar to native Android apps except that additional files and directories are included. In directory `/asset/bin/Data/Managed`, there are PE-format DLL files that contain CIL codes in their text sections. Developer's C# codes are compiled into `Assembly-CSharp*.dll` files, and these files are loaded and executed by Mono runtime. Other DLL files are added to support APIs such as Unity API and Mono embedded API. Directory `lib` contains native libraries for Mono runtime and JIT compilation. Table 1 lists several libraries common to Unity apps.

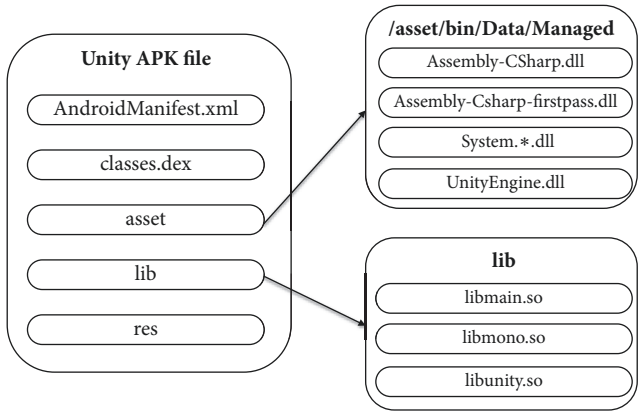


Figure 1: Structure of Unity APK File.

3.2. A Framework for Static Analysis of Unity Apps. As mentioned above, there are three types of execution codes in C# based Android apps built with Unity. They are Dalvik bytecode, .NET CIL code, and native code. Dalvik bytecode can be decompiled using Dex to Java decompilers such as jadx or JEB. CIL code can be decompiled into .NET C# code using .NET decompiler such as dotpeek, ILSpy, or .NET Reflector. Native code can be disassembled using disassembler such as IDA. Figure 2 shows our framework for static analysis.

3.3. Initialisation Process of Unity Apps. This section presents some results of the analysis focusing on the initialisation of Unity apps in order to show the effectiveness of the proposed framework for static analysis. Figure 3 illustrates the process of initialisation of a Unity app, Arrow.io, from the start of the app to loading of native libraries. When the app is launched, `onCreate()` method of `MainActivity` class (the class of activity component specified in `AndroidManifest.xml`) is invoked. `MainActivity`'s `onCreate()` calls its superclass method `UnityPlayerActivity`'s `onCreate()` first. Note that the activity component of Unity apps always inherits `UnityPlayerActivity`. `UnityPlayerActivity`'s `onCreate()` instantiates `UnityPlayer`.

Table 2: Comparison of Android native apps and Unity apps.

	Android native apps	Unity apps
Language	Java, C/C++	Java, C#, IL
APK directories	Lib, res, META-INF	Lib, res, META-INF, assets/bin/Data/Managed
Runtime Environment	Android Runtime	Mono Runtime
Symbol Recovery	Unnecessary	Necessary
Debugger	JDB, GDB	JDB, GDB, SDB
Debugger protocol	Java Debugger Wire Protocol	Soft Debugger Wire Format

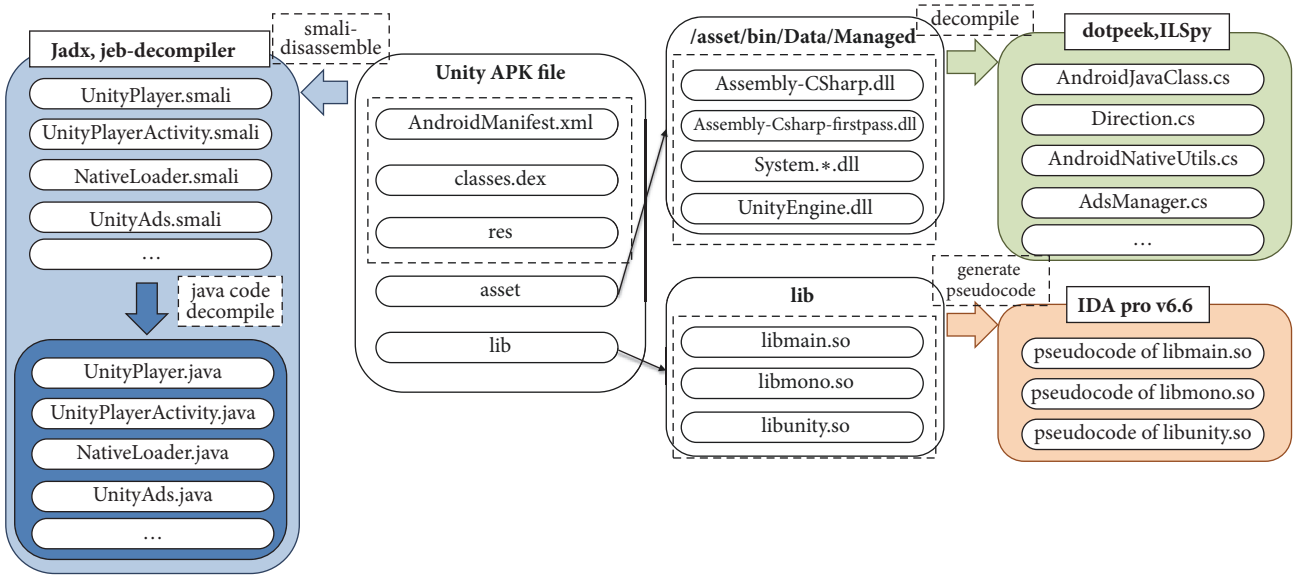


Figure 2: A framework for static analysis of Unity apps.

`UnityPlayer` loads `libmain.so` by calling `System.loadLibraryStatic()` method. When `libmain.so` is loaded, `JNI_OnLoad()` function of `libmain.so` is called and it binds its native function to `load()` method of `NativeLoader` class. `NativeLoader`'s `load()` loads `libunity.so` (a library for the core Unity game engine) and `libmono.so` (a library for Mono runtime). `JNI_OnLoad()` function of `libunity.so` binds its `nativeXXXX()` functions to native methods of `UnityPlayer`. It registers 17 native functions using `RegisterNatives()` function.

One of the registered native functions is `nativeRender()`. Called by `UnityPlayer`'s constructor, (1) it launches Mono runtime by invoking `mono_jit_init_version()` in `libmono.so`. Then (2) it creates an instance of `MonoManager` which is a C++ class declared in `libunity.so`. Finally (3) it invokes `MonoManager::LoadAssemblies()` which loads all DLL files (containing CIL codes) in `/assets/bin/Data/Managed` (see Figure 4).

Many mobile games support authentication using social login such as Facebook login or Google sign-in and/or display advertisement pages to earn money. Such facilities are typically incorporated in apps as a form of library. Since most of them are written in Java, the main game (written in C#) needs to call Java methods.

Unity apps utilise internal calls to find and invoke Java methods. C/C++ code can invoke Java methods using JNI functions such as `FindClass()`, `GetStaticFieldID()`, `GetMethodID()`, and `CallObjectMethod()`. `UnityEngine.dll` binds these functions to C# functions `INTERNAL_CALL_FindClass()`, `INTERNAL_CALL_GetStaticFieldID()`, `INTERNAL_CALL_GetMethodID()`, and `INTERNAL_CALL_CallObjectMethod()`, respectively. These functions are called internal calls and can be invoked by C# codes. They can be registered by calling `mono_add_internal_call()` in `libmono.so`.

The process of invoking Java methods from C# codes is as follows.

(1) Generate the values of options for Java virtual machine (`libunity.so`)

(2) Create Java virtual machine (`libunity.so`)

(3) Search and load Java class (`UnityEngine.dll`)

(4) Get the ID of Java method to invoke (`UnityEngine.dll`)

(5) Generate argument information for Java method (`UnityEngine.dll`)

(6) Invoke method (`UnityEngine.dll`)

(7) Destroy Java virtual machine (`libunity.so`)

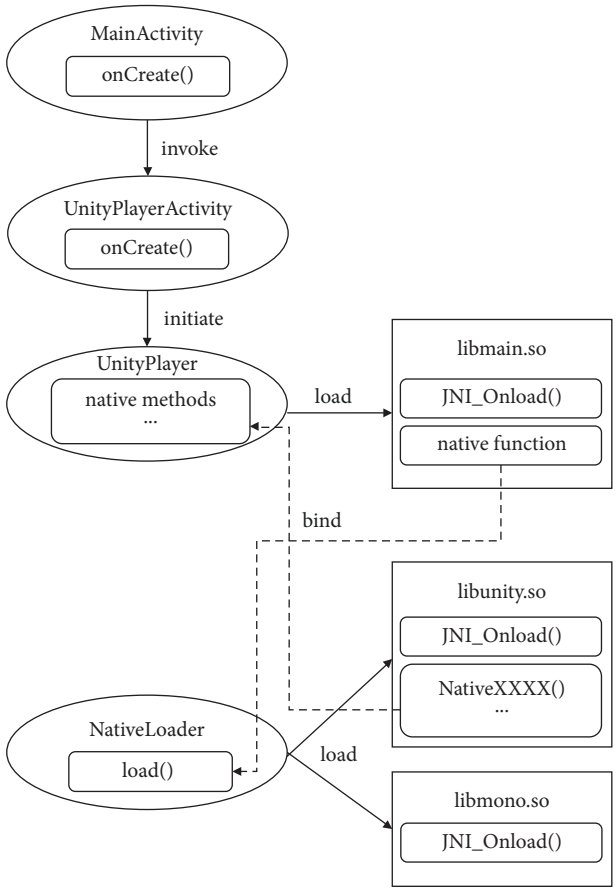


Figure 3: Loading native libraries.

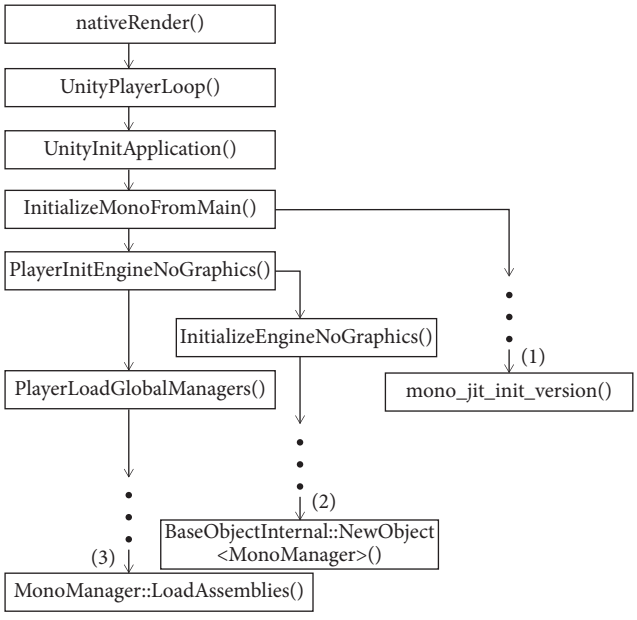


Figure 4: Loading DLL files.

Since it is impossible to create or destroy a Java virtual machine in the domain of C#, the operations related to creation/destroy of Java virtual machine are implemented in the native domain (`libunity.so`). The remaining jobs are implemented as C# functions (in `UnityEngine.dll`) for internal calls such as finding and loading Java classes, obtaining method ID of Java methods to invoke, generating argument information to pass to Java methods, and invoking Java methods.

Now the Unity engine and the Mono runtime are ready to execute developer's codes. `UnityPlayer`, `libunity.so`, and `libmono.so` collaboratively proceed with the main game. `libunity.so` invokes `libmono.so`'s functions to JIT-compile and execute developer's CIL codes.

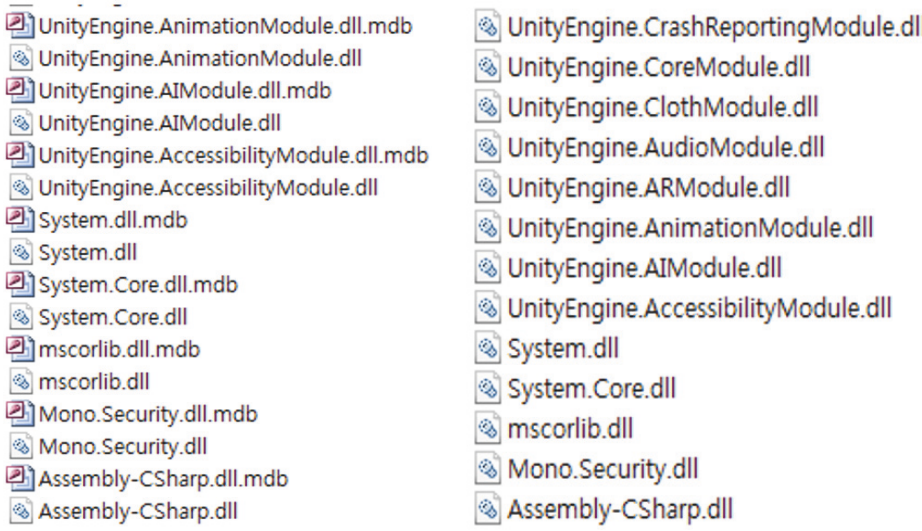
4. Dynamic Analysis of Unity Apps

Developers can build their apps in either a debugging or release mode. The former is called debug build and the latter is called release build. In debug build, apps include both debug symbol files and some libraries that can perform profiling functions. In release build, however, apps do not include debug symbol files or profiling libraries because they do not influence the execution of apps and the size of APK files can be reduced. Hence, the way how to perform dynamic analysis of an app depends on which mode the app is built in. Note that the term development build is used instead of the term debug build in Unity.

4.1. Development Build versus Release Build. This section explains the difference between development build and release build of Unity apps. We conduct both development build and release build on the same source codes of a Unity app and obtain two APK files. Then we analyse each of them within the framework presented in Section 3.

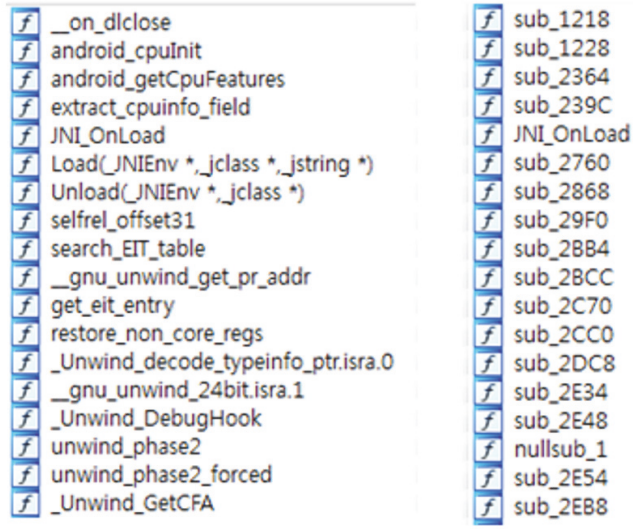
Figure 5(a) shows files in `/asset/bin/Data/Managed` of development-built APK, where there are DLL files and mdb files (files with extension .mdb). DLL files contain CIL codes and each mdb file contains the symbol information for its corresponding DLL file. In MS Windows environment, the symbol file for compiled C# code is pdb file (files with extension .pdb). In Unity apps, the symbol file is mdb file whose format is different from pdb file and is read and understood by Mono soft debugger which is a debugger built in Mono runtime. mdb files contain the information for mapping line numbers and methods in source codes. If Mono soft debugger cannot find mdb file, it cannot perform debugging operation such as setting breakpoints on the codes in the corresponding DLL file. Figure 5(b) shows the files in `/asset/bin/Data/Managed` of release-built APK. There is no mdb file because Unity does not generate mdb files in release build.

Development build and release build also differ in shared libraries. We inspect `libmain.so` of the two APK files using IDA pro. As shown in Figure 6, the symbol names remain in development build but are removed from release build. This is the same for other libraries common to Unity apps such as `libmain.so`, `libunity.so`, and `libmono.so`.



(a) Development build (b) Release build

Figure 5: Files in `/asset/bin/Data/Managed`.



(a) Development build (b) Release build

Figure 6: Symbols in `libmain.so`.

Another difference is the part that creates Mono runtime which is located in `libunity.so`. Our static analysis on `libunity.so` identified the part that initialises Mono runtime. Figure 7 shows the codes that create Mono runtime in development build. Mono runtime is created by invoking `mono_jit_init_version()`. Before invoking this function, `mono_debug_init()` is invoked to prepare Mono soft debugger inside Mono runtime. Then debuggers outside Mono runtime can communicate with Mono soft debugger over a socket connection. In release build, `mono_debug_init()` is not invoked as shown in Figure 8. This implies that we should put some codes invoking `mono_debug_init()` into `libunity.so` in order to connect to Mono soft debugger and dynamically analyse release-built apps.

The abovementioned difference between development build and release build makes it complicated to analyse release-built app dynamically. Since most Unity apps in the market are built in release mode, they neither contain debug symbol files nor initialise Mono soft debugger. The rest of this section presents a technique that can implement a debugging environment for dynamic analysis of release-built apps by generating debug symbol files and executing codes that initialise Mono soft debugger.

4.2. Debugging Release-Built Apps. The proposed dynamic analysis technique proceeds in the following order (as shown in Figure 9). To use Mono soft debugger, we generate debug symbol files (mdb files), modify `AndroidManifest.xml` file,

```
mono_debug_init(1);
v67 = v91 == 0;
if ( v91 )
  v67 = v92 == 0;
if ( !v67 )
  free_alloc_internal(v91, &v94);
v68 = LODWORD(v96) == 0;
if ( LODWORD(v96) )
  v68 = HIDWORD(v96) == 0;
if ( !v68 )
  free_alloc_internal(LODWORD(v96), &v99);
v69 = 0;
mono_set_commandline_arguments(v76);
v70 = mono_jit_init_version("Unity Root Domain", "v2.0.50727");
```

Figure 7: Mono runtime initialisation in development-built APK.

```
mono_set_assemblies_path(v38, v50);
v39 = v50 == 0;
if ( v50 )
  v39 = v51 == 0;
if ( !v39 )
  sub_102614(v50, v53);
sub_2DEA8C(unk_12D2060, v3);
v40 = v54 == 0;
if ( v54 )
  v40 = v55 == 0;
if ( !v40 )
  sub_102614(v54, v57);
v41 = v58 == 0;
if ( v58 )
  v41 = v59 == 0;
if ( !v41 )
  sub_102614(v58, v61);
v42 = 0;
mono_config_parse(0);
mono_set_signal_chaining(1);
v43 = mono_parse_default_optimizations(0);
mono_set_defaults(0, v43);
mono_set_commandline_arguments(v49);
v44 = mono_jit_init_version("Unity Root Domain", "v2.0.50727");
```

Figure 8: Mono runtime initialisation in release-built APK.

and repackage the app. After installing this repackaged app on an Android device, we launch the app and stop it before Mono runtime is created. We may modify `classes.dex` to stop the app or use the Debugger-Wait function with `am` command. When the app is stopped, we hook a native function to execute the code that invokes `mono_debug_init()` and resume the app. Once this code is executed, we can connect to Mono soft debugger for debugging by attaching SDB, a client for Mono soft debugger. The following sections explain this procedure step by step.

4.2.1. Generating Debug Symbol Files. To debug a release-built app, we must first create mdb files. However, as mentioned in Section 4.1, the release build does not create mdb files. We need to create mdb files manually and repackage the app. Figure 10 illustrates this process.

We use the `ildasm` and the `ilasm` commands, which are installed with Visual Studio. To create an mdb file, we need to create a pdb file first. The `ildasm` command disassembles a DLL file and creates an IL (intermediate language) text file with extension .il. The `ilasm` command reassembles an IL text file. Given proper arguments, it creates a new DLL file and a pdb file. We, then, use the `pdb2mdb` command to convert the pdb file to an mdb file.

We modify the `AndroidManifest.xml` file as follows. We set `debuggable` attribute of `application` element to `true` and allow `INTERNET` permission to perform remote debugging. We repackage the app after signing back and then install the app on the device. Now preparing the app for dynamic analysis completes.

4.2.2. Attaching a Debugger for Dynamic Analysis. We describe how to connect SDB to repackaged apps. As mentioned in Section 4.1, a release-built app does not have code to call the `mono_debug_init()` function. To connect SDB to Mono soft debugger, we must call the `mono_debug_init()` function before the Mono runtime instance is created. To do this, we must be able to do two things:

(1) To stop execution of an app before Mono runtime instance is created

(2) To call `mono_debug_init()` function directly

On Android, we can use the am command to run an app in the Debugger-Wait state. If we do so, we can see that the process is suspended waiting for a debugger to attach. When we attach JDB, the process is resumed. The example command line is as follows.

```
# adb shell am set-debug-app -w
<application> --persistent
```

We hook a native function `mono_set_default()` in `libunity.so` using an instrumentation tool such as Frida. We ascertained by static analysis that this function is always called prior to `mono_jit_init_version()`. We inject the following code into `mono_set_default()` to initialise Mono soft debugger by invoking `mono_debug_init()`.

```
char * dbgoption={"--debugger-
agent=transport=dt_socket,embedding=1,
defer=y,address=0.0.0.0:9999",NULL};
mono_jit_parse_options(1,dbgoption);
mono_debug_init(1);
```

To attach SDB to an app, we first connect JDB to the app, execute the above Mono soft debugger initialisation code, and run SDB to connect to Mono soft debugger. We can then debug a Unity app as usual. Figure 11 is a screen shot of SDB. We set breakpoint at the `Test.Update()` method using bp command (fourth line) and connect to Mono soft debugger using `connect` command with the IP address of the target Android device (sixth line). Once the connection is established, Mono runtime process and C# threads are started (8th–15th lines). Then the app stops at the `Test.Update()` method and prompts for debugger command (16th–18th lines). The app stops at line 69 in the IL text file (`Assembly-CSharp.il`) and displays the next code to execute, `ldstr "Test Text"`, which is the first code of `Test.Update()`.

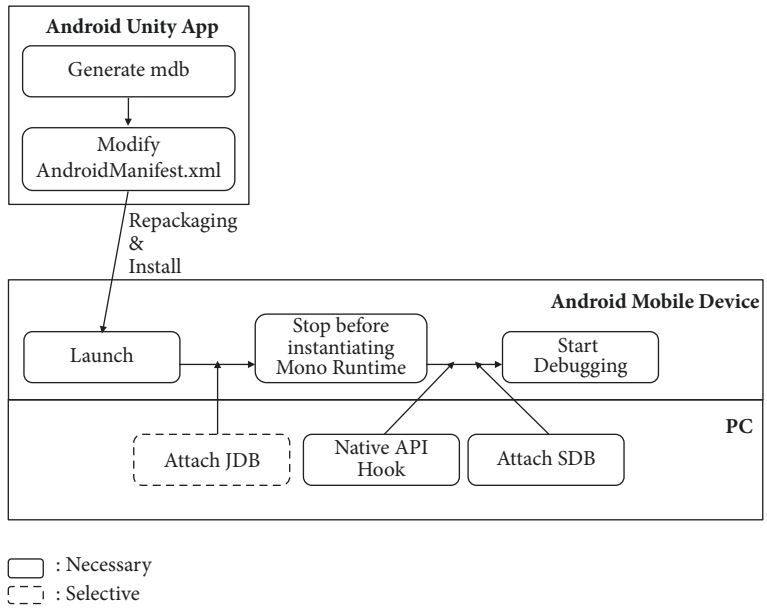


Figure 9: Procedure of dynamic analysis.

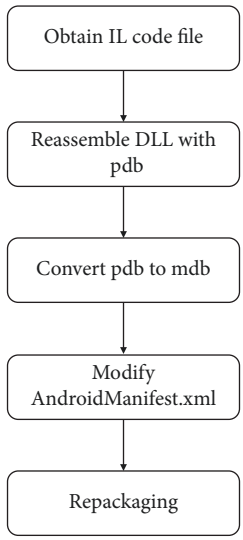


Figure 10: Process of creating mdb files and repackaging apps.

Figure 11: A screen shot of SDB.

5. Discussion

Typical Android apps and Unity apps differ in terms of their app structure and execution environment (Table 2). First, Unity apps have `/assets/bin/Data/Managed` folder and contain DLL files for execution. Second, they have different libraries in their package. Android apps include native libraries only when developers explicitly specify, while Unity apps always include three libraries: `libmono.so`, `libmain.so`, and `libunity.so`. Of course, developers can specify libraries as needed. Third, the code runs in different runtime environments. Android apps run on the Android Runtime, while Unity apps run on the Mono runtime, which executes the IL codes. Fourth, Android apps can be analysed dynamically if we merely modify `AndroidManifest.xml`, but Unity apps can be analysed only after restoring symbol information of DLL files. Finally, they use different debugger protocols due to different runtime environments. JDB uses Java Debug Wire Protocol (JDWP), which performs debugging on Java code. SDB, however, uses Soft Debugger Wire Format protocol (SDB protocol), which performs debugging on IL codes. Two protocols are similar because SDB protocol is made by referring to JDWP.

We propose a method for dynamically analysing Unity apps on the Java layer, the native layer, and the Mono runtime layer where C # code is executed. JDB is connected to the Java layer, GDB to the native layer, and SDB debugger to the Mono runtime layer to perform operations such as setting breakpoints or querying parameters.

Malware analysis techniques can be classified into two categories: static and dynamic. Static analysis, mostly used by anti-virus program, can analyse target's executable by looking at suspicious patterns or by disassembling and further decompiling it into high-level language like C# or Java. Static analysis can give a complete result of the executable by covering different execution paths including static control flow and data flow. However, static analysis may have limitations

Table 3: Static analysis tools for Unity apps.

Tool	Description
jadx	Command line and GUI tools for producing Java source code from Android DEX file
jd-gui	GUI tool for displaying Java source codes of ".class" files
ILspy	Open-source decompiler and static analyser for software created with .NET Framework
dotpeek	Decompiler and static analyser for software created with .NET Framework
.NET Reflector	Commercial decompiler and static analyser for software created with .NET Framework
IDA pro	Disassembler for computer software which generates assembly language source code from machine-executable code
ildasm	Tool for generating Common Intermediate Language (CIL) code from portable executable (PE) file
ilasm	Tool for generating a portable executable (PE) file from Common Intermediate Language (CIL) code
pdb2mdb	Command line tool for convert .NET debug symbol files (namely, pdb files) to debug symbols files (namely, mdb files) that can be understood by Unity's scripting engine

Table 4: Dynamic analysis tools for Unity apps.

Tool	Description
JDB	Command line debugger for Java classes
GDB	Portable debugger for tracing and altering the execution of computer programs
SDB	Command line client for Mono's soft debugger

due to the complexity of pointer aliasing, the prevalence of indirect jumps, and the lack of types in executables. It is also known that static analysis is vulnerable to code packing and obfuscation methods, which are commonly applied to malware to avoid anti-virus detection. Some malicious apps are distributed even in a form of components, which are decrypted at runtime.

In case malware is obfuscated or packed to hinder static analysis, dynamic analysis or behaviour-based analysis can be effective. Dynamic analysis can observe execution traces of the target app behaviour and analyse its properties by executing the malware in an isolated and controlled environment. It can also monitor actions of the target app during execution with the help of a specialized tool such as a debugger. Therefore, dynamic analysis is resilient to code obfuscation and packing and is able to monitor malicious behaviour on an actual execution path. The disadvantage is the lack of code coverage as it explores a single execution path at a time.

We summarise useful tools for analysing Unity apps statically and dynamically in Tables 3 and 4, respectively. Static and dynamic analysis both have their own merits and demerits. A few researches have tried to combine static and dynamic analysis whenever possible to have the benefits of both. In this paper, we present an effective static and dynamic analysis technique for detecting Android malware written with Unity framework.

6. Conclusion

Cross-platform mobile app development tools allow developers to write only one mobile app that is then deployed to all the supported target platforms, to reduce the cost and time for developing mobile apps, and to reach out to maximum users across several platforms. While app authors use cross-platform mobile app development tools for the "*Write once, Run everywhere*" feature, malware authors use them for the "*Write once, Infect everywhere*" feature. Unity is one of the most popular cross-platform mobile app development tools. A recent study conducted by SophosLabs reported that the number of malware and *potentially unwanted apps* (PUAs) written with Unity had been increasing in proportion to the number of mobile apps developed with Unity. Moreover, the pieces of malware conceal its malicious code in specific containers (e.g., DLL files) loaded by Unity instead of the target platform's native codes. Therefore, security researchers are confronted with great challenges to detect these pieces of mobile malware.

To address the challenges, in this paper, we first present a systematic technique that statically and dynamically analyse *Android apps developed with Unity framework* (Unity apps). Our static analysis focuses on the initialisation of target apps to examine the structure and interaction between object codes of the apps. The static analysis consists of the following behaviours: understanding executable file format, decompiling DEX, Microsoft .NET DLLs, and native codes, and inspecting static control flow of the decompiled codes. Next, we propose an effective dynamic analysis technique for Unity apps that are built in release mode. Such apps cannot be debugged as they are, since they do not contain any debugging symbol. To debug a release version of Unity app, we generate the debugging symbols from the release version, repackage the release version with the generated symbols, and then run the repackaged app. We can stop the execution of the Unity app before the Mono runtime instance is created and call debugging preparation function directly. After attaching the debugger, we can set breakpoints and inspect the parameters. The proposed techniques are effective and efficient for determining if a suspicious app written with Unity is malicious or benign and can be used to discover bugs or vulnerabilities in the suspicious app.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning [no. 2015R1A2A1A15053738]. This research was also supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program [IITP-2018-2015-0-00363] supervised by the IITP (Institute for Information & Communications Technology Promotion).

References

[1] M. Willocx, J. Vossaert, and V. Naessens, "A Quantitative Assessment of Performance in Mobile App Development Tools," in *Proceedings of the 3rd IEEE International Conference on Mobile Services, MS 2015*, pp. 454–461, usa, July 2015.

[2] N. Boushehrinejadmoradi, V. Ganapathy, S. Nagarakatte, and L. Iftode, "Testing cross-platform mobile app development frameworks," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp. 441–451, usa, November 2015.

[3] M. Martinez and S. Lecomte, "Towards the Quality Improvement of Cross-Platform Mobile Applications," in *Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017*, pp. 184–188, arg, May 2017.

[4] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba, "Taxonomy of Cross-Platform Mobile Applications Development Approaches," *Ain Shams Engineering Journal*, vol. 8, no. 2, pp. 163–190, 2017.

[5] H. Rocky, *Designing Platform Independent Mobile Apps and Services*, John Wiley & Sons, 2016.

[6] S. Blackman, *Beginning 3D Game Development with Unity 4: All-in-one, multi-platform game development*, Apress, Berkeley, CA, USA, 2013.

[7] J. Haas, *A History of the Unity Game Engine*, Worcester Polytechnic Institute, Worcester, UK, 2014.

[8] P. P. Patil and R. Alvares, "Cross-platform Application Development using Unity Game Engine," *International Journal of Advance Research in Computer Science and Management Studies*, vol. 3, no. 4, pp. 19–27, 2015.

[9] Unity Engine - Official Site, "Unity," http://unity3d.com.

[10] W. Lee and X. Wu, "Cross-platform mobile malware: write once, run everywhere," in *Proceedings of the Virus Bulletin Conference*, 2015.

[11] S. M. Pontiroli and F. R. Martinez, "he Tao of .NET and PowerShell Malware Analysis," in *Proceedings of the Virus Bulletin Conference*, 2015.

[12] "The Mono Runtime - Official Site," http://www.mono-project.com/docs/advanced/runtime/.

[13] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications," in *Proceedings of the 6th International Conference on Malicious and Unwanted Software, Malware 2011*, pp. 66–72, pri, October 2011.

[14] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "New android malware detection approach using Bayesian classification," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013*, pp. 121–128, Spain, March 2013.

[15] B. Jean et al., "Static detection of malicious code in executable programs," *International Journal of Advanced Structural Engineering*, vol. 79, pp. 184–189, 2001.

[16] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 1–3, 2002.

[17] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[18] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pp. 259–269, ACM, June 2014.

[19] S. Schrittwieser and S. Katzenbeisser, "Code Obfuscation against Static and Dynamic Reverse Engineering," in *Information Hiding*, vol. 6958 of *Lecture Notes in Computer Science*, pp. 270–284, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[20] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, pp. 421–430, December 2007.

[21] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal of Computer Virology and Hacking Techniques*, vol. 2, no. 1, pp. 67–77, 2006.

[22] G. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[23] A. Lanzi, M. I. Sharif, and Lee. Wenke, "K-Tracer: A System for Extracting Kernel Malware Behavior," *NDSS*, 2009.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," *ACM SIGPLAN Notices*, vol. 45, no. 6, p. 1, 2010.

[25] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[26] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, no. 2, article 6, 2012.

[27] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks, SIN 2013*, pp. 152–159, tur, November 2013.

[28] J. Lim and J. H. Yi, "Structural analysis of packing schemes for extracting hidden codes in mobile malware," *EURASIP Journal on Wireless Communications and Networking*, vol. 2016, no. 1, 2016.

[29] T. Cho, H. Kim, and J. H. Yi, "Security Assessment of Code Obfuscation Based on Dynamic Monitoring in Android Things," *IEEE Access*, vol. 5, pp. 6361–6371, 2017.

[30] D. Andrade, R. M. Paulo et al., *Cross platform app: a comparative study*, 2015, arXiv:1503.03511.

[31] T. Majchrzak and T. Grønli, "Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks," in *Proceedings of the Hawaii International Conference on System Sciences*, 2017.

[32] M. Latif, Y. Lakhrissi, E. H. Nfaoui, and N. Es-Sbai, "Review of mobile cross platform and research orientations," in *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems, WITS 2017*, mar, April 2017.

[33] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *Proceedings of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC '13)*, pp. 323–328, July 2013.

[34] A. Mylonas, S. Dritsas, B. Tsoumas, and D. Gritzalis, "On the Feasibility of Malware Attacks in Smartphone Platforms," in *E-Business and Telecommunications*, vol. 314 of *Communications in Computer and Information Science*, pp. 217–232, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[35] A. Mylonas, S. Dritsas, B. Tsoumas, and D. Gritzalis, "Smartphone security evaluation—the malware attack case," in *Proceedings of the 8th International Conference on Security and Cryptography (SECRYPT '11)*, pp. 25–36, Seville, Spain, July 2011.

[36] K. Chen, X. Wang, Y. Chen et al., "Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pp. 357–376, San Jose, CA, May 2016.

[37] J. Xing et al., *Code injection attacks on HTML5-based mobile apps*, 2014.

[38] J. Mao, J. Bian, G. Bai et al., "Detecting Malicious Behaviors in JavaScript Applications," *IEEE Access*, vol. 6, pp. 12284–12294, 2018.