Dynamic Analysis of Android Apps written with PhoneGap Cross-Platform Framework

Jaewoo Shim[1], Minjae Park[1], Seong-je Cho[1], Minkyu Park[2], and Sangchul Han[2]*
[1]Dankook University, Yongin, Korea
{tlawodn94, parkminjae, sjcho}@dankook.ac.kr
[2]Konkuk University, Chungju, Korea
{minkyup, schan}@kku.ac.kr

Abstract

In this paper, we propose an effective technique that can perform dynamic analysis for Android app written with PhoneGap cross-platform framework. For a systematic study, we have written a malicious Android app using PhoneGap framework. We compare the structural differences between a basic Android app (a native app) and the other malicious Android app built in release mode on PhoneGap framework, and also analyze the malicious app dynamically. The proposed technique first copies the web root directory of the target malicious app into a writable directory inside the smartphone. When the app is executed, its web pages and Javascript files are loaded from the copied directory using a dynamic instrumentation. Finally, we dynamically change the flag for WebView debugging so that a remote debugger can successfully be attached to the app built in release mode. Using our proposed technique, a malware analyst can debug a malicious PhoneGap app built in release mode without repackaging, which cannot be debugged as it is by Chrome remote debugger. She/he can also utilize the debugging features supported by the remote debugger. The technique allows the analyst to bypass the repackaging detection method that malicious apps use to avoid antivirus detection.

Keywords: PhoneGap framework, Cross-platform, Android app, Dynamic analysis

1 Introduction

Nowadays there exist various mobile platforms in the market including Android, iOS, Huawei, and Xiaomi. Since the development framework for these mobile platforms are different from each other, app developers have to develop the same native app repeatedly using different development frameworks in order to cover many platforms. Note that a native app refers to an application that is developed for use on a particular platform. Developing the same app for each platform using their native SDK requires in-depth knowledge of each platform, dedicated SDK, and programming skill for different programming language. In addition, developers might not share source codes across platforms, resulting in writing similar codes multiple times. This incurs a high development cost.

Cross-platform development frameworks [20, 8, 18, 13] can reduce the development cost if developers want to develop an app for multiple platforms. When developing an app using cross-platform development frameworks, developers usually write the source codes in a platform-independent programming language. The codes are interpreted or translated by run-time library that supports app execution. Well-known cross-platform development frameworks are PhoneGap, Xamarin, Unity, Qt, Appcelerator Titanium, and Ionic.

Among those frameworks, PhoneGap is a very popular cross-platform framework for mobile app development [6, 12, 4, 5]. PhoneGap is an implementation of the renamed Apache Cordova open source

*Corresponding author: Department of Software Technology, Konkuk University, 268 Chungwondaero, Chungju-si, Chungcheongbuk-do, 27478, Korea, Tel: +82-43-840-3605 project. It takes advantages of the standardized web technologies used for mobile web development and brings them to native web applications. PhoneGap supports all major platforms such as Android, iOS, Windows Mobile, Blackberry 10, Tizen, etc. According to the study [6] that compared the three frameworks: PhoneGap, Titanium, and Xamarin, PhoneGap is preferred for cross-platform development in terms of its ability to deliver apps with high capabilities.

Malware writers also utilize cross-platform development frameworks to create malware for multiple platforms easily. [17] reported that malicious apps developed with the frameworks increase rapidly and that among the collected Android malware in 2015 the number of malware and potentially unwanted app (PUA) samples written with PhoneGap were 203 and 91, respectively.

The logic of the malicious activity for the PhoneGap sample is implemented in HTML/Javascript. Since the Javascript codes packed in APK files can be easily extracted and analyzed, malware writers usually obfuscate the Javascript codes so as to hide the malicious activity and/or prevent the Javascript codes from static analysis. In this case, dynamic analysis can be effective rather than static analysis. In order to analyze dynamically Javascript codes in a PhoneGap app, analysts might want to attach a debugger to PhoneGap app and execute the Javascript codes line by line while monitoring its behavior. However, debugging Javascript codes packed in PhoneGap app is not straightforward. To the best of our knowledge, no such demonstration has ever been offered without repackaging the app.

In this paper, we present a technique for dynamic analysis of Android apps written in HTML/-Javascript with PhoneGap framework. We focus on debugging a PhoneGap app deployed in release mode which cannot be debugged as it is. We first extract and copy the web root directory of a target PhoneGap app into a writable directory in the mobile device. Then, we can load web pages and Javascript files from the copied directory by modifying method parameters using a dynamic code instrumentation tool. Finally we change the flag for WebView debugging, and a remote debugger can be attached to the target app without repackaging the app.

The rest of this paper is organized as follows. Section 2 explains related research work. Section 3 explains structure of PhoneGap apps, and Section 4 proposes an effective technique for dynamically analyzing PhoneGap apps and describes experiments. Conclusions are presented in Section 5.

2 Related Work

A native app is defined as an app that is developed in a development environment and with tools unique to a specific smartphone environment and is executable only in that environment [14, 22, 19]. Developers use the development tools and programming languages chosen by the smartphone platform to develop their native apps. Examples of development tools include Eclipse, Xcode, and the Windows Mobile Development Tool. Java, Objective C, and C# are used in those environments, respectively. In contrast, cross-platform apps are developed on one cross-development platform and transformed to run on a specific smartphone platform. This reduces the effort required to develop an app that must be run on multiple platforms.

Dalmasso et. al. compared and evaluated development methods of native apps, mobile web apps, and cross-platform apps based on app quality, app development costs, and app security and the like [11]. Native apps provide good UX and reliability and are powerful, but developers must use different programming languages for each platform. Cross-platform apps are less secure than native apps.

Corral et al. Analyzed the process of performing OS codes and the execution time of each call when PhoneGap API was called [10]. However, they did not describe how to perform such a dynamic analysis on PhoneGap apps. We propose a method to perform dynamic analysis on an PhoneGap app.

PhoneGap developers can debug an app using Visual Studio IDE. Typically they build an application by incorporating the PhoneGap Tool to the Visual Studio IDE. They can perform dynamic debugging by running an app in the emulator of the target Android device. They set up a breakpoint and print out the specific value dynamically on the source code level [16]. However, they cannot analyze without source codes.

Developers can debug the UI using a web server. This server starts to run when the project management tool named PhoneGap Desktop is executed [15]. Since the changes of html source is reflected directly on the UI, developers can debug a graphical component easily. However, analyst do not have a project file used for building a app and cannot analyze apps in the same way.

Developers can take advantage of the remote debugging capabilities of the Chrome browser [1]. To use the remote debugging feature of the Chrome browser, you must have a Chrome browser installed on both the developer's PC and the Android device, and you must connect the PC and the Android device directly [2]. However, this method cannot be used for apps built for release mode because apps do not have any useful information for analysis.

Burguera et al. [9] proposed a new framework to obtain and analyze smartphone application activity. The data collection app, Crowdroid, collects data of apps on the device installed and sends them to a server. The data include API calls, which well describe the activity of apps, The server anayse the data and cluster the behavior into two clusters: benign and malicious applications. However, they only target native apps.

Wu et. al. [21] proposed DroidMat, a static feature-based mechanism to detect the Android malware. DroidMat considers permissions, deployment of components, Intent messages passing and API calls for characterizing the Android applications behavior and uses different kinds of clustering algorithms. DroidMat extracts the information from each application's manifest file, and traces API Calls. it, then, decides the number of clusters by Singular Value Decomposition (SVD) method. Finally, it uses kNN algorithm to classify the application as benign or malicious. However, they also only target native apps and analyze information statically.

Arp et. al. [7] proposed DREBIN, a method for detection of Android malware that enables identifying malicious applications directly on the smartphone. DREBIN combines concepts from static analysis and machine learning. It is lightweight in that it requires less than a day to analyze 100,000 unknown applications. It, however, has limitations of static analysis.

In this paper, we propose a method to solve the problems in the remote debugging method of Chrome browser and to create an analysis environment so that the analyst can use the same analysis tool as the developer. This has the advantage that analysts can make full use of the elements utilized by developers, and it can take advantage of features already supported by existing debuggers for already developed Android apps.

3 Structure of PhoneGap Apps

This section describes the structure of PhoneGap app APK files. Section 3.1 compares the structure of native app APK file and PhoneGap app APK file. Section 3.2 explains the difference between debug mode build and release mode build of PhoneGap app.

3.1 Native App vs. PhoneGap App

APK files are a zip-format archive file that contains many directories and files. Figure 1 shows the structure of APK file of native app and PhoneGap app, respectively. A typical APK file contains the following directories and files. `META-INF` contains developer's signature. If an APK is modified or repackaged, the signature verification fails and the APK cannot be installed to Android devices. `lib` contains native libraries which are usually developed in C/C++ for fast processing or security reason. Developers can

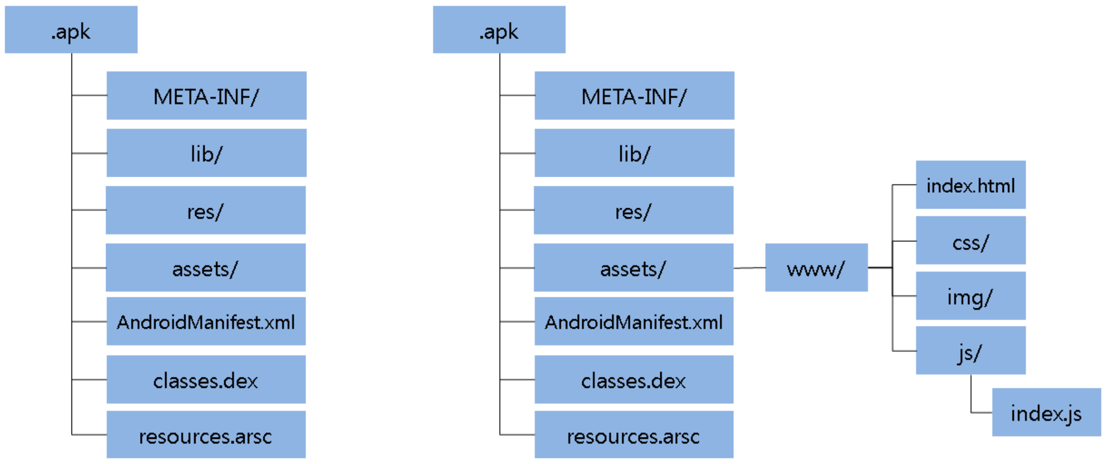


Figure 1: Structure of Native App APK (left) and PhoneGap App APK (right)

Table 1: Difference between Debug Mode and Release Mode

	Debug Mode Build	Release Mode Build
Total number of files	173	171
`AndroidManifest.xml`	3,016 bytes	2,976 bytes
`classes.dex`	702,512 bytes	430,444 bytes

support various architecture such as x86, arm and mips by setting target architecture. If developers do not use native libraries, `lib` may not be included. `res` contains resource-related information that is not compiled into `resources.arsc`. `assets` contains files or directories that are utilized by the app at run-time. Developers can access files inside `assets` using class `AssetManager`. Modifying files in `assets` requires repackaging the app. `AndroidManifest.xml` manifests the information that Android system has to know before the app executes, such as its name, access permission, components, etc. Finally, `classes.dex` contains the compiled execution codes.

Different from native apps, PhoneGap apps have a web-related directory `/asset/www`. In general, `www` has sub-directories `css`, `img` and `js`. The contents of `css` and `img` are CSS files for HTML files and image files, respectively. The image files are loaded locally by the app's web pages. `js` contains Javascript files that implement the program logic and behavior of the app. HTML files are typically located in `/asset/www`. Among them, `index.html` contains HTML codes for the app's first screen (web page) and codes for loading Javascript files. The default name of Javascript file that `index.html` loads is `index.js`. Developers can rename `index.html` by modifying `config.xml` in the PhoneGap project. Developers can also rename `index.js`, `css`, `img`, and `js` as whatever they want.

3.2 Debug Mode vs. Release Mode

To find out the difference between debug mode build and release mode build of PhoneGap apps, we create and build a simple app, `HelloWorld`. The app is created using PhoneGap Desktop 0.4.5. Table 1 compares the debug mode build and release mode build in terms of the number of file and the size of files contained.

The debug mode build has two more files than the release mode build; `CERT.RSA` and `CERT.SF`. These files contain APK signing-related information. They are automatically generated in the debug mode build and APK packaging does not require developer's signing. In contrast, the release mode build does not include those files, and APK packaging requires developer's signing.

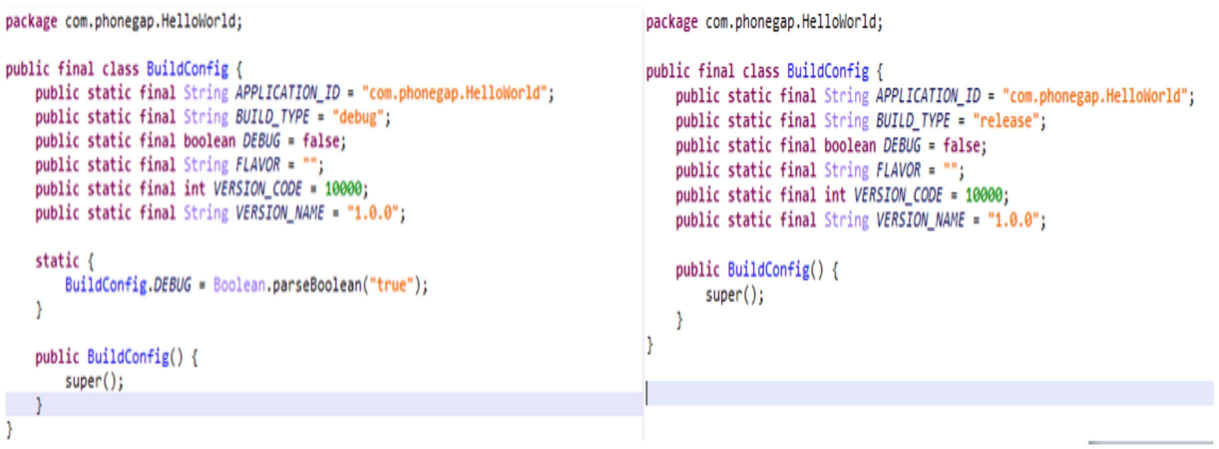
```
<application
    android:debuggable="true"
    android:hardwareAccelerated="true"
    android:icon="@mipmap/icon"
    android:label="@string/app_name"
    android:supportsRtl="true">
```

```
<application
    android:hardwareAccelerated="true"
    android:icon="@mipmap/icon"
    android:label="@string/app_name"
    android:supportsRtl="true">
```

Figure 2: `<application>` tag in the debug mode build (left) and in the release mode build (right)

The size of `AndroidManifest.xml` of the debug mode build is greater than the release mode build. The difference is the atrributes of `<application>` tag as shown in Figure 2. The value of `android:debuggable` is `true` in the debug mode build. In the release mode build, the value is `false` or the attribute itself is not specified.

Finally, the size of `classes.dex` in the debug mode build is greater than the release mode build. This is because some execution codes are added to `classes.dex` in the debug mode build. For example, Figure 3 shows the decompiled codes of class `BuildConfig` using a reverse engineering tool JEB in each build. The code that assigns `true` to `BuildConfig.DEBUG` is added in the debug mode build.



```
package com.phonegap.HelloWorld;

public final class BuildConfig {
    public static final String APPLICATION_ID = "com.phonegap.HelloWorld";
    public static final String BUILD_TYPE = "debug";
    public static final boolean DEBUG = false;
    public static final String FLAVOR = "";
    public static final int VERSION_CODE = 10000;
    public static final String VERSION_NAME = "1.0.0";

    static {
        BuildConfig.DEBUG = Boolean.parseBoolean("true");
    }

    public BuildConfig() {
        super();
    }
}
```

```
package com.phonegap.HelloWorld;

public final class BuildConfig {
    public static final String APPLICATION_ID = "com.phonegap.HelloWorld";
    public static final String BUILD_TYPE = "release";
    public static final boolean DEBUG = false;
    public static final String FLAVOR = "";
    public static final int VERSION_CODE = 10000;
    public static final String VERSION_NAME = "1.0.0";

    public BuildConfig() {
        super();
    }
}
```

Figure 3: Decompiled codes of class `BuildConfig` in the debug mode build (left) and in the release mode build (right)

4 Dynamic Analysis of PhoneGap Apps and Experiments

This section describes our dynamic analysis scheme for PhoneGap apps and experiment results. By performing dynamic analysis, analysts can observe the behavior of codes and identify malicious ones. Especially, dynamic analysis can analyze even packed or obfuscated apps based on their behavior. In our work, the experimental environments are as follows. The Android device used in our experiments is Google Oreo Pixel running Android 8.0.1. We also use a Linux PC running Ubuntu 16.04 for analysis tools such as Chrome Remote Debugger and Frida [3]

Figure 4 shows the overview of our scheme. First, from an APK file we extract and copy the web-root directory tree (`/asset/www`) into a writable directory, in our work, `/data/local/tmp`. Then, we install the APK file. Later we will instrument the app to load web pages and Javascript files from the copied directory. The reason is that during dynamic analysis analysts often need to modify program codes (Javascript files in case of PhoneGap apps) but the files packaged in APK files cannot be modified.

Another way is to decompress the APK file, modify program codes, repackage and re-install the APK file. However, this may not work if the app employs repackaging detection techniques.

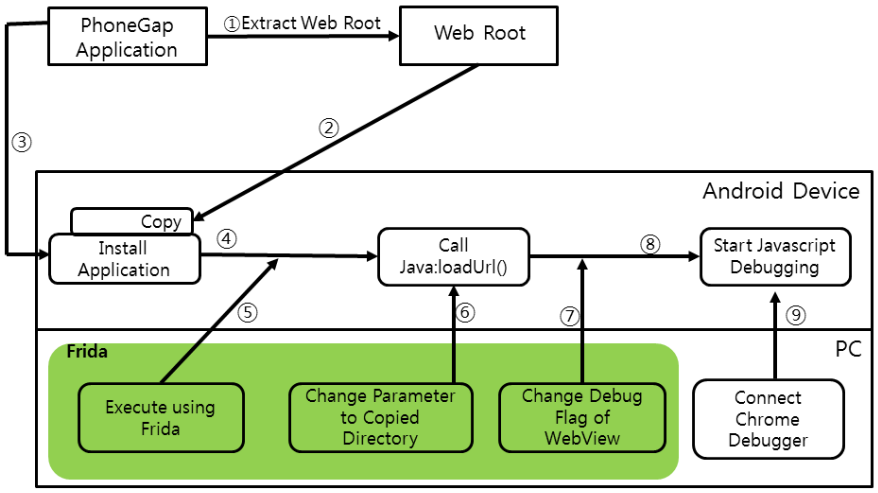


Figure 4: Overview of the Proposed Scheme

We connect the Android device and the PC directly using a USB cable in order to utilize Chrome Remote Debugger. (As explained in Section 2,) Chrome browser should be installed on both sides in advance. If we install and run a PhoneGap app built in debug mode, the Remode Debugger of Chrome browser in the PC can access and debug the PhoneGap app running on the Android device.

Next, we use a dynamic code instrumentation toolkit *Frida* to hook the app's methods. From static analysis of PhoneGap apps, we know that a Java native method `loadUrl()` loads and displays HTML file. Note that static analysis of PhoneGap apps is outside the scope of this paper. `loadUrl()` takes a parameter that references the path of HTML file. We hook `loadUrl()` and replace the parameter value with the path of HTML file which we copied in `/data/local/tmp`. Now we can perform dynamic analysis modifying the Javascript files. The example Frida script is shown in Figure 5.

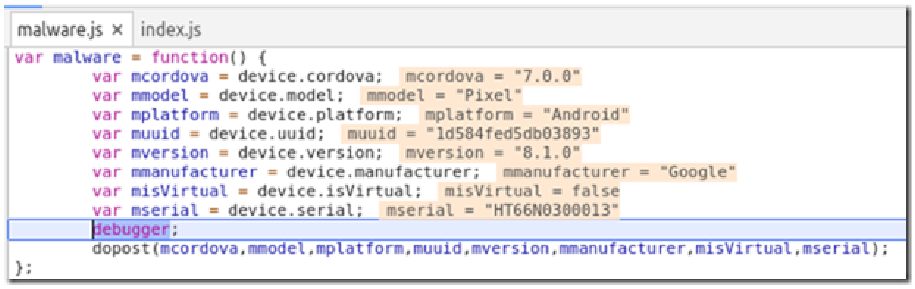
```
1: Java.perform(function () {
2:     var IS_DONE=false;
3:     var Webkit = Java.use('android.webkit.WebView');
4:     Webkit.loadUrl.overload('java.lang.String').implementation = function (v) {
5:         v = "file:///data/local/tmp/www/index.html";
6:         if(IS_DONE == false) {
7:             Webkit.setWebContentsDebuggingEnabled(true);
8:             IS_DONE = true;
9:         }
10:         return this.loadUrl(v);
11:     };
12: });
```

Figure 5: Example Frida Script

Let us look into the script in Figure 5. At line 3, we get the reference to `WebView` class. At line 4, we hook its `loadUrl()` method to execute our codes, line 5-10. In this example, we replace `loadUrl()`'s parameter value as the path of the copied HTML file, `/data/local/tmp/www/index.html`. And we change a flag so that a remote debugger can connect to this app by invoking `setWebContentsDebugging`

`Enable()` (line 7). By changing the flag, the web contents that `WebView` loads such as HTML, CSS and Javascript become debuggable. In general, the value of the flag is `True` for apps built in debug mode but `False` for apps built in release mode. So the script change the flag into `True` compulsorily.

Hereafter, we can modify Javascript files in the copied directory to set breakpoints. Figure 6 shows a screenshot of Chrome Remote Debugger where a breakpoint is inserted and the values of variables are displayed. In this way, analysts can analyze codes in web pages by modifying or de-obfuscating codes and hooking methods to resolve method-call trace.



```
var malware = function() {
        var mcordova = device.cordova;  mcordova = "7.0.0"
        var mmodel = device.model;  mmodel = "Pixel"
        var mplatform = device.platform;  mplatform = "Android"
        var muuid = device.uuid;  muuid = "1d584fed5db03893"
        var mversion = device.version;  mversion = "8.1.0"
        var mmanufacturer = device.manufacturer;  mmanufacturer = "Google"
        var misVirtual = device.isVirtual;  misVirtual = false
        var mserial = device.serial;  mserial = "HT66N0300013"
        debugger;
        dopost(mcordova,mmodel,mplatform,muuid,mversion,mmanufacturer,misVirtual,mserial);
};
```

Figure 6: Screenshot of Chrome Remote Debugger

5 Conclusion and Future Work

In this paper, we analyzed the architectural differences between an Android native app and a PhoneGap; and the structural differences of the apk when an app is built in debug mode and release mode. Based on the results, we proposed a method to dynamically analyze PhoneGap app by setting some variables related to debug. After constructing an environment for accessing malicious processes running on the Android terminal using the dynamic instrumentation framework Frida, the analyst can perform arbitrary operations to find malicious behavior. The proposed scheme has the advantage that it can analyze a malware even although it includes the functionality of detecting the repacking. In the future, we plan to develop a framework that detects the obfuscation of an app and makes it easier for analysts to analyze it.

Acknowledgments

This research was supported by (1) Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and (2) 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] Chrome Remote Debugging. `https://cordova.apache.org/docs/en/latest/guide/next/#chrome-remote-debugging/` [Online; accessed on August 20, 2018].

[2] Chrome Remote Debugging process with android. `https://developers.google.com/web/tools/chrome-devtools/remote-debugging/` [Online; accessed on August 20, 2018].

[3] Frida. `https://www.frida.re/` [Online; accessed on August 20, 2018].

[4] V. Ahti, S. Hyrynsalmi, and O. Nevalainen. An evaluation framework for cross-platform mobile app development tools: A case analysis of adobe phonegap framework. In *Proc. of the 17th International Conference on Computer Systems and Technologies (CompSysTech '16), Palermo, Italy*, pages 41–48. ACM, June 2016.

[5] P. R. M. Andrade, A. Albuquerque, O. F. Frota, R. V. Silveira, and F. A. da Silva. Cross platform app: a comparative study. *International Journal of Computer Science & Information Technology*, 7(1):33–40, February 2015.

[6] F. Appiah. A tool selection framework for cross platform mobile app development. `http://dspace.knust.edu.gh/handle/123456789/8013/` [Online; accessed on August 20, 2018], November 2015.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *Proc. of the 21th Annual Network and Distributed System Security Symposium (NDSS'14), San Diego, California, USA*, pages 23–26. NDSS, Feburary 2014.

[8] N. Boushehrinejadmoradi, V. Ganapathy, S. Nagarakatte, and L. Iftode. Testing cross-platform mobile app development frameworks (t). In *Proc. of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15), Lincoln, Nebraska, USA*, pages 441–451. IEEE, November 2015.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11), Chicago, Illinois, USA*, pages 15–26. ACM, October 2011.

[10] L. Corral, A. Sillitti, and G. Succi. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736–743, 2012.

[11] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *Proc. of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC'13), Sardinia, Italy*, pages 323–328. IEEE, July 2013.

[12] S. Dhillon and Q. H. Mahmoud. An evaluation framework for cross-platform mobile application development tools. `https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.2286` [Online; accessed on August 20, 2018], August 2014.

[13] 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*, 8(2):163–190, June 2017.

[14] H. Heitkötter, S. Hanschke, and T. A. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *Proc. of the 8th International Conference on Web Information Systems and Technologies (WEBIST'12), Porto, Portugal*, pages 120–138. Springer-Verlag, April 2012.

[15] C. Lantz. PhoneGap Desktop. `http://docs.phonegap.com/getting-started/1-install-phonegap/desktop/` [Online; accessed on August 20, 2018].

[16] C. Lantz. Debug your app built with Visual Studio Tools for Apache Cordova. `https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/debug-test/debug-using-visual-studio?view=toolsforcordova-2017` [Online; accessed on August 20, 2018], 2016.

[17] W. Lee and X. Wu. CROSS-PLATFORM MOBILE MALWARE: WRITE ONCE, RUN EVERYWHERE. `https://pdfs.semanticscholar.org/3f6e/8aafc110fa2958c5216500d771021958a4bd.pdf` [Online; accessed on August 20, 2018], June 2015.

[18] M. Martinez and S. Lecomte. Towards the quality improvement of cross-platform mobile applications. In *Proc. of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems(MOBILESoft'17), Buenos Aires, Argentina*, pages 184–188. IEEE, May 2017.

[19] K. Selvarajah, M. P. Craven, A. Massey, J. Crowe, K. Vedhara, and N. Raine-Fenning. Native apps versus web apps: Which is best for healthcare applications? In *Proc. of the 15th International Conference on Human-Computer Interaction (HCI'13), Las Vegas, Nevada, USA*, volume 8005 of *Lecture Notes in Computer Science*, pages 189–196. Springer-Verlag, July 2013.

[20] M. Willocx, J. Vossaert, and V. Naessens. A quantitative assessment of performance in mobile app development tools. In *Proc. of the 2015 IEEE International Conference on Mobile Services, New York, New York, USA*, pages 454–461. IEEE, July 2015.

[21] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proc. of the 7th Asia Joint Conference on Information Security, Tokyo, Japan*, pages 62–69.

IEEE, August 2012.

[22] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proc. of the 6th Balkan Conference in Informatics (BCI'13), Thessaloniki, Greece*, pages 213–220. ACM, September 2013.

Author Biography

Jaewoo Shim received the B.E in the Dept. of Software at Dankook University, Korea in 2017. He is currently a master student at Dept. of Computer Science and Engineering in Dankook University, Korea. His research interests include computer system security, reverse engineering, and software vulnerability analysis.

Minjae Park received the B.E in the Dept. of Information and Communication Engineering at National Institute for Lifelong Education, Korea in 2015. He is currently a master student at Dept. of Computer Science and Engineering in Dankook University, Korea. His research interests include computer system security, and reverse engineering.

Seong-je Cho received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1989, 1991 and 1996, respectively. In 1997, he joined the faculty of Dankook University, Korea, where he is currently a Professor in Department of Computer Science & Engineering (Graduate school) and Department of Software Science (Undergraduate school). He was a visiting research professor at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. His current research interests include computer security, mobile app security, operating systems, and software intellectual property protection.

Minkyu Park received the B.E., M.E., and Ph.D. degree in Computer Engineering from Seoul National University in 1991, 1993, and 2005, respectively. He is now a professor in Konkuk University, Rep. of Korea. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI. He has authored and co-authored several journals and conference papers.

Sangchul Han received his B.S. degree in Computer Science from Yonsei University in 1998 and his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now a professor in the Dept. of Software Technology at Konkuk University. His research interests include real-time scheduling and computer security.