This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2693388, IEEE Access

Security Assessment of Code Obfuscation Based on Dynamic Monitoring in Android Things

Taejoo Cho, Hyunki Kim, and Jeong Hyun Yi*

***Abstract*—Android-based IoT devices with excellent compatibility and openness are constantly emerging. A typical example is Android Things that Google supports. Compatibility based on the same platform can provide more convenient personalization services centering on mobile devices, while this uniformity-based computing environment can expose many security vulnerabilities. For example, new mobile malware running on Android can instantly transition to all connected devices. In particular, the Android platform has a structural weakness that makes it easy to repackage applications. This can lead to malicious behavior. To protect mobile apps that are vulnerable to malicious activity, various code obfuscation techniques are applied to key logic. The most effective one of this kind involves safely concealing application programming interfaces (API). It is very important to ensure that obfuscation is applied to the appropriate API with an adequate degree of resistance to reverse engineering. Because there is no objective evaluation method, it depends on developer judgment. Therefore, in this paper, we propose a scheme that can quantitatively evaluate the level of hiding of APIs, which represent the function of the Android application, based on machine learning theory. To perform the quantitative evaluation, API information is obtained by static analysis of a DEX file, and the API-called code executed in Dalvik in the Android platform is dynamically extracted. Moreover, sensitive APIs are classified using the extracted API and Naive Bayes classification. The proposed scheme yields a high score according to the level of hiding of the classified API. We tested the proposed scheme on representative applications of the Google Play Store. We believe it can be used as a model for obfuscation assessment schemes because it can evaluate the level of obfuscation in general without relying on specific obfuscation tools.**

***Index Terms*—Android Things, mobile security, security assessment**

I. Introduction

THE Internet-of-Things (IoT) era in which everything is connected to the Internet is coming. According to Gartner [1], by 2020, 28 billion devices will be connected. Let us take a look at the computing environment of most IoT devices that will be introduced in this super-connected society. Today, desktops or mobile devices dominate most markets with certain processors (ie, Intel, ARM). It is true that Windows, Linux, and Android dominate most of the operating systems running on these hardware. Basically, IoT devices are embedded devices. In order to provide users with convenient services, it is necessary to link them with smart phones that contain a lot of personal information. From this point of view, Android-based IoT devices with excellent compatibility and openness are constantly emerging. A typical example is Android Things [2] supported by Google.

This uniformity-focused computing environment, on the other hand, can expose many vulnerabilities from a security perspective. For example, new mobile malware running on Android can instantly transition to all connected devices. In addition, from the viewpoint of the reverse engineering analyst, if the uniformization of the OS is accelerated, the attacker can understand all the connected devices easily by understanding the kernel structure and the system call and the execution file format. What is the power of Android in this situation? The Android mobile operating system has the largest worldwide share of the smart device platform market. In terms of application deployment, Apple's iOS has a closed policy involving the use of the official store, whereas Google's Android has an open policy that allows for third-party stores, and permits the installation of applications via Android application package(APK) files. The platform has a similar open-source policy. Such an open architecture renders the platform structurally vulnerable and allows malicious behavior in devices using the Android platform because it is easy to repackage applications. For example, a malicious agent decompiles a application consisting of bytecodes into smali codes and analyzes the functional part of the application. Following this, malicious code, such as leaked personal information or manipulated financial information, is inserted, and repackaging is carried out to distribute the malicious code to the user of the mobile platform. It has been reported that users' personal assets might hence be compromised, and financial application tampering [3], login credential tampering [4], messenger application tampering [5], and malicious behavior scenarios through cloud devices [6] become possible.

The Android platform controls access to such system resources as contacts, e-mail, personal user information, GPS, and network through a permission-based security mechanism. Permissions are granted to the application with the user's consent when installing the application. However, most users are unaware of the implications of Android permissions and the effects of these permissions on the device. Moreover, they tend to have little interest in how permissions are accessed and used, which suggests that the Android permission system does not adequately protect users from malicious applications. Therefore, risk assessment schemes that can adequately recognize the risk posed by an application to the user are being researched. Application risk assessment is a scheme that can assess the risk of an application using various metadata. The main goal of this assessment is to systematically assess the

T. Cho, Department of Computer Science and Engineering, Soongsil University, Seoul, 06978, Korea, e-mail: taejoo90@gmail.com

H. Kim, Department of Software Convergence, Soongsil University, Seoul, 06978, Korea e-mail: hitechnet92@gmail.com

J. H. Yi, School of Software, Seoul, 06978, Korea, e-mail: jhyi@ssu.ac.kr

*Corresponding Author

risk posed by the vast amount of applications deployed in the market. Known risk assessment schemes can provide meaningful information to some users, mainly based on permission [7], description [8], user reviews [9], or static analyses of the application [10]. However, these evaluation techniques are disadvantageous in that they are disabled by static analysis prevention techniques, such as an attacker's unnecessary permission request, a developer's careless description creation, repetitive and malicious user review manipulation, and code packing.

Code obfuscation techniques are applied to key logic and APIs to protect mobile apps vulnerable to these risks. The semantics of obfuscation mean to reduce readability to prevent reverse engineering analysis of the source code, but a more effective way to prevent malicious attacks is to conceal the APIs used for key function calls. From this point of view, while current obfuscation schemes elevate some islets of static analysis, such as changing layout of the source code, changing control flow, and modifying data, they are easily exposed to reverse engineering analysis due to a lack of API concealment. Therefore, a quantitative evaluation scheme is needed to ensure that obfuscation is applied to an appropriate API with an adequate degree of resistance to reverse engineering.

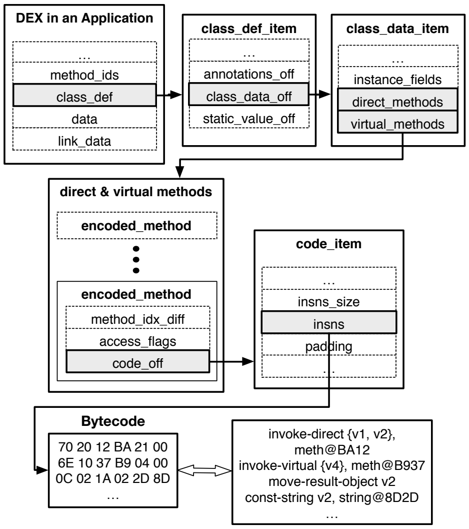
In light of the above, in this paper, we propose a scheme that can quantitatively evaluate the level of concealment of APIs, which represent the function of Android applications, based on machine learning theory. To perform the quantitative evaluation, API information is acquired by static analysis of a DEX file, and APIs called in the code executed in Dalvik are dynamically extracted. Sensitive APIs are classified by applying the extracted API and Naive Bayes. The proposed scheme yields a high score according to the level of concealment of the classified API. Min-Max normalization is also used to ensure that the value generated by the formula has a value that is intuitive between 0 and 1. The proposed scheme provides quantitative values according to the level of obfuscation applied to the given API and the level of its resistance to reverse engineering from the viewpoint of information hiding. The proposed scheme is also designed to be independent of specific obfuscation techniques.

The paper is organized as follows. Section 2 deals with code extraction and explains how Dalvik bytecodes are accessed and executed. We also describe Naive Bayes, which is applied to sensitive API classification and API protection from the point of view of obfuscation. Section 3 describes the quantitative obfuscation evaluation scheme proposed in this paper. In Section 4, we apply the proposed scheme to practical applications to test it, and offer our conclusions in Section 5.

II. Background

A. Execution of Dalvik Bytecode

An APK file is composed of a compressed file, and contains resource and code information need to execute the application. A file with actual code is a DEX file stored in the APK file with the extension `.dex`, and contains class and method information. The DEX file structure consists of a `header`, `string_ids`, `type_ids`, `proto_ids`, `field_ids`, `method_ids`, `class_def`, `data`, and `link_data` fields. Of the fields, data are stored in the `data` field, and `offset` information stored in each field is used to calculate the location of the data.



```
70 20 12 BA 21 00
6E 10 37 B9 04 00
0C 02 1A 02 2D 8D
...
```

```
invoke-direct {v1, v2},
meth@BA12
invoke-virtual {v4}, meth@B937
move-result-object v2
const-string v2, string@8D2D
...
```

Fig. 1. Bytecode access process

To access the bytecode used by the application, `class_def field` is used as shown in Fig. 1. `Class_def_item` exists in the `class_def` field in the DEX file structure and contains information about classes. The `class_data_off` field in `class_def_item` points to the location of `class_data_item`. `Class_data_item` contains field and method information. The `method` field is used to find the method information inside. The `method` field is composed of `encoded_method`. It locates `code_item` using the `code_off` field inside `encoded_method`. `Code_item` is stores the instruction information of the method, and the `insns` field has an array of bytecodes. Using the information in the field, it is possible to extract the API called in the DEX file structure in a static way.

When executing the application, the Android platform loads the DEX file into memory as well as the necessary permission information and resources for execution. At this point, the interpreter in the Dalvik virtual machine interprets the bytecode in the DEX file and executes the instruction. The Dalvik virtual machine loads the DEX file described above and accesses the instruction to execute it. Fig. 2 shows the process of interpreting and executing bytecodes on the Dalvik virtual machine.

A detailed description of each process is as follows:

1) Read the instruction at the address pointed to by the given PC. The instruction stored at the address pointed to by the PC was `70 20`.
2) The `70` of `70 20` is the opcode. An opcode table is used to determine the instruction of the opcode. In the figure above, the instruction table of `invoke-kind` is shown. We confirmed that the example opcode `70` was `invoke-direct`.

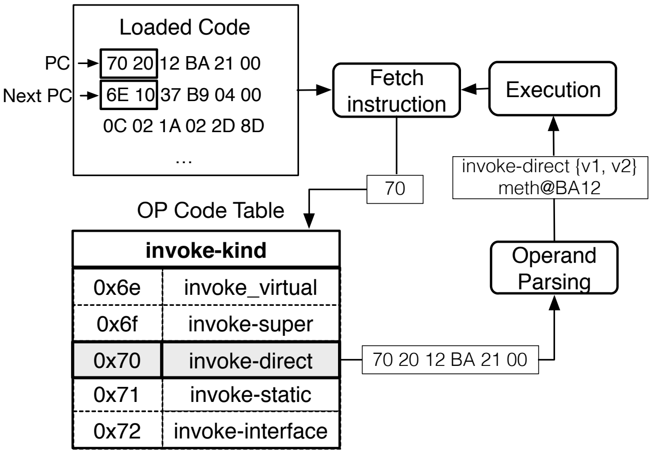


Fig. 2. Bytecode interpretation and execution process

3) Parse the required bytes according to the instruction. The example parses the bytes as shown below:
`(op AG B3B4 B1B2 DC FE` → `[A=2] op vC, vD, kind@B1B2B3B4)`
`(70 20 12 BA 21 00` → `invoke-direct v1, v2, meth@BA12)`
4) Execute the parsed instruction.
5) Move the PC point to the next instruction.
`(PC` → `6E 10)`
6) Repeat Steps 1 through 5.

B. Naive Bayes Classification

Naive Bayes classification is a machine learning algorithm and a supervised learning method. It uses the Naive Bayes theorem and determines classification based on the probability of the object. Nave Bayes is calculated using conditional probability as shown in Equation (1).

$$\begin{aligned} C &= argmaxP\left(x_1, x_2, x_3, \cdots, x_k\right) \\ &= argmax \prod P\left(x \mid c\right) P\left(c\right) \end{aligned} \tag{1}$$

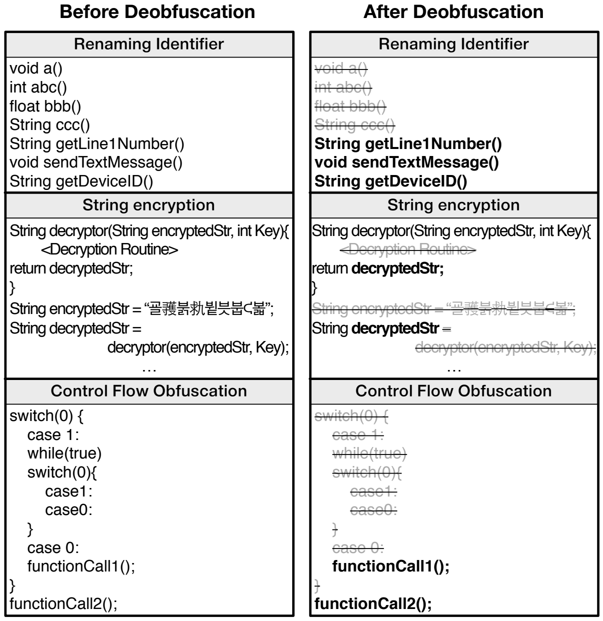
In Equation (1), x_n represents the characteristic (dependent variable) of the objects to be classified, and is divided into n characteristics, and C represents the class to be classified. Naive Bayes classification uses maximum likelihood estimation, which uses the class with maximum likelihood as the result. Likelihood is here $\prod P\left(x \mid c\right)$ in the above formula, and is calculated as the product of conditional probabilities when each characteristic belongs to a specific class.

Naive Bayes classification is commonly used to categorize word-based documents, such as spam. The amount of training data required to estimate input values is at least highly accurate, and can be suitably considered in an environment where the feature space dynamically changes [11][12]. The training data of the sensitive API classification scheme used in this paper is composed of words and the feature space changes dynamically. Therefore, applying Naive Bayes to sensitive API classification can help us infer accurate and appropriate results for the training data.

C. Execution of Bytecode

Previous studies have shown that after decompiling an Android application, such malicious behavior as certain features and inserted malicious code for repackaging can be identified. To perform such a functional analysis, it is necessary to analyze the method that the application is calling. From a functional point of view, the application consists of a method created by the user and an API provided by the Android platform. User-defined methods and APIs determine the functionality and characteristics of the application, where the functionality of the application is crucial according to the kind of method used and combination called. The actual part of the function is the API, and a user-created method also calls the API and executes the function. Even when malicious actors attempt malicious behavior through reverse engineering, they need to analyze methods they have created to identify the functional parts of these applications and, ultimately, the APIs used. In order for a malicious agent to be immune to reverse engineering, it is necessary to protect the above-mentioned API calls, and various obfuscation techniques have been applied for protection [13][14].

Such obfuscation techniques as renaming, control flow obfuscation, and string encryption do not have sufficient reverse engineering resistance in terms of functional protection following application. Each obfuscation technique is analyzed from a reverse engineering point of view, as shown in Fig. 3.



Before Deobfuscation	After Deobfuscation
Renaming Identifier	**Renaming Identifier**
void a() int abc() float bbb() String ccc() String getLine1Number() void sendTextMessage() String getDeviceID()	~~void a()~~ ~~int abc()~~ ~~float bbb()~~ ~~String ccc()~~ **String getLine1Number()** **void sendTextMessage()** **String getDeviceID()**
String encryption	**String encryption**

String decryptor(String encryptedStr, int Key){
<Decryption Routine>
return decryptedStr;
}
String encryptedStr = “골護붉執뷥븟붑ᑕ붊”;
String decryptedStr =
decryptor(encryptedStr, Key);
...
	String decryptor(String encryptedStr, int Key){
	~~<Decryption Routine>~~
	return **decryptedStr;**
	}
	~~String encryptedStr = “골護붉執뷥븟붑ᑕ붊”;~~
	String **decryptedStr** ~~=~~
	~~decryptor(encryptedStr, Key);~~
	...

Control Flow Obfuscation	**Control Flow Obfuscation**

switch(0) {
case 1:
while(true)
switch(0){
case1:
case0:
}
case 0:
functionCall1();
}
functionCall2();
	~~switch(0) {~~
	~~case 1:~~
	~~while(true)~~
	~~switch(0){~~
	~~case1:~~
	~~case0:~~
	~~}~~
	~~case 0:~~
	functionCall1();
	~~}~~
	functionCall2();

Fig. 3. Limitations of obfuscation schemes

In the case of renaming, user-created methods can be obfuscated, but the obfuscation technique cannot be applied to API calls. As shown in Fig. 3, the original method name of the user method cannot be found, such as `a ()`, `abc ()`, `bbb ()`, and `ccc ()`, whereas naming cannot be applied to methods, such as `getLine1Number ()`, `sendTextMessage ()`, and `getDeviceID ()`.

In the case of string encryption, it can encrypt the string using complex encryption logic, which is sufficiently resistant to reverse engineering when viewed statically. It also hides strings used in parameters or application layouts. However, even if encryption is applied, the encrypted string must be restored to the original string through the decryption routine. Therefore, if the decryption logic is found and the return value tracked by searching the return part, it is not a very difficult part. String encryption can be applied to the string input as an API parameter, but reverse engineering is possible if the return value is tracked by excluding the string decryption routine and other encrypted strings, as shown in Fig. 3.

Control flow obfuscation contains unnecessary logic, but it can be difficult to analyze its execution flow from a static point of view. Once again, this can be easily accomplished without knowing the actual execution logic and excluding unnecessary logic. In the figure, after performing unnecessary logic through control flow obfuscation, `functionCall1 ()` and `functionCall2 ()`are called. If only `functionCall1 ()` and `functionCall2 ()` parts remain following the removal of the unnecessary logic, there is no difficulty in terms of reverse engineering.

Obfuscation schemes such as renaming, control flow obfuscation, and string encryption exhibit some degree of reverse engineering resistance from a static point of view but do not interfere with functional analysis. The schemes described above have recently been used, such schemes as API hiding, class encryption, packing [15][16], and DEX file separation have been used [17]. These have a greater degree of reverse engineering resistance than the conventional method because the analyst must dynamically analyze extract instructions or use techniques such as a memory dump. Such techniques as API hiding, class encryption, and packing can conceal the functional parts or characteristics of the application by hiding the API call and functions written by the user. If the analyzer does not have sufficient knowledge of the obfuscation scheme, the analysis needed for reverse engineering takes a long time. However, such schemes also do not provide sufficient resistance to reverse-engineering analysis of applications unless applied at appropriate locations.

III. Proposed Obfuscation Evaluation Scheme

As mentioned earlier, to protect the functionality of applications, obfuscation techniques with adequate resistance to reverse engineering should be applied for appropriate coverage. The goal of the proposed scheme is to evaluate whether APIs can be adequately protected, and provide a quantitative measure of whether obfuscation is applied at the appropriate locations. An evaluation score is generated according to the degree of functional concealment of the API call, and is designed to be monotonic with higher scores as resistance to reverse engineering increases. To design the scheme, static code extraction, dynamic code extraction, and sensitive API classification are required. The following explains each in detail.

A. Static Code Extraction

Static code extraction is used to extract an API call instruction from an application file. The extraction process parses the DEX file that stores the code of the application. It generates a data structure and only the framework API calls of codes in the data structure through the framework of API classification.

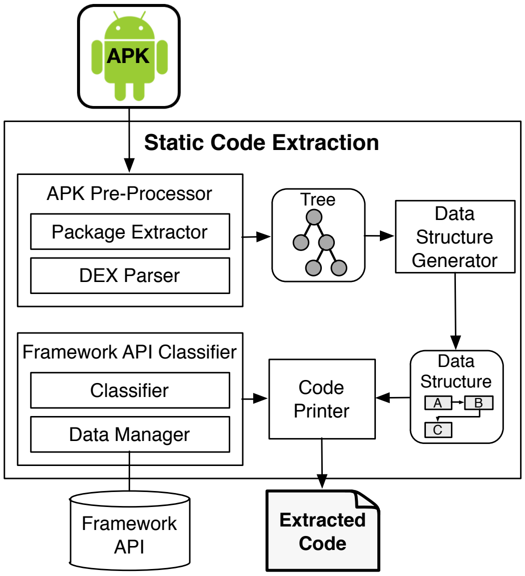


Fig. 4. Static code extraction scheme

The structure of the module and the flow of static code extraction are shown in Fig. 4. The `APK Pre-processor` consists of a `Package Extractor` and a `DEX Parser`. The `Package Extractor` receives an application file to be extracted. Following input, it extracts the DEX file, which stores code in the structure of the application. The `DEX parser` parses the DEX file. A parsed DEX file consists of a tree, which further consists of `package`, `class`, and `method` information. The `Data Structure Generator` generates a data structure by parsing the tree generated from the `APK Pre-processor`. The data structure is in the form of a list, and consists of information that can indicate the characteristics of the API, such as package information, class information, method information, parameters, and return values of the extracted API. The `Framework API Classifier` determines whether the input API is a framework API based on the `Framework API DB` that stores information concerning the framework API. The `Data Manager` handles framework API data and the classifier matches API information retrieved from the database manager and the desired API information. The `Code Printer` is responsible for generating code classified by the Framework API Classifier.

B. Dynamic Code Extraction

This procedure extracts the API call instruction of the application being executed. When the application to be extracted is loaded on the platform, the extraction option is set. When

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2693388, IEEE Access

the `method` is executed, only the API is extracted through framework API classification. The output API list is generated as a log file and used for analysis.

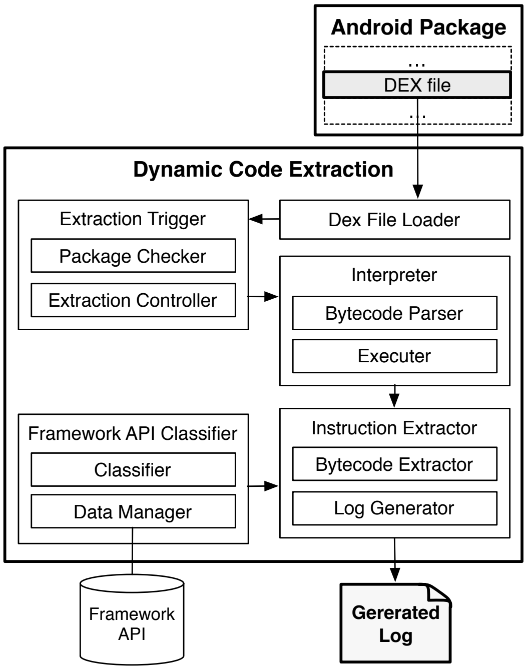


Fig. 5. Dynamic code extraction scheme module structure

The module structure and flow of the dynamic code extractor are shown in Fig. 5. The `DEX File Loader` loads the resources required to run the application. The `Extraction Trigger` consists of a `Package Checker` and an `Extraction Controller`. The `Package Checker` determines whether the application is a target application for analysis. If it is, the `Extraction Controller` sets analysis points and options. The `Interpreter` consists of a `Bytecode Parser` and an `Executer`. The former reads and parses the bytecode of the DEX loaded through the `DEX File Loader`, where the parsed bytecode is executed by the `Executer`. The `Instruction Extractor` consists of the `Bytecode Extractor` and a `Log Generator`. For an application to be analyzed, the former extracts the bytecodes that are executed by the application. The `Log Generator` extracts bytecode in the form of a log containing thread number, PC, frame number, and instruction information. The `Framework API Classifier` classifies only the `Framework API` of the methods called in dynamic code extraction. It consists of a `Classifier` and a `Data Manager`. The classifier queries the framework API using the `Data Manager` and outputs the command called in the `Dynamic Code Extractor` based on the retrieved data in the case of framework API. For the purpose of objective data extraction, API is extracted by executing XML based function in dynamic code extraction.

C. Sensitive API Classification

The sensitive API classification scheme determines whether an API is sensitive. In this scheme, package name, class name, API name, and API description information are used, and the Naive Bayes classification method is applied by taking these data as parameters.

TABLE I
INFORMATION OF `getLine1Number` API

Type	Description
`API Name`	getLine1Number
`Package Name`	androidTelephony
`Class Name`	TelephonyManager
`API Description`	Returns the phone number string for line 1

TABLE II
CONDITIONAL PROBABILITY BY API WORD

word	$P(x_i \mid SensitiveAPI)$	$P(x_i \mid NormalAPI)$
get	0.027	0.004
returns	0.027	0.022
phone	0.009	0.002
number	0.009	0.002
telephony	0.009	0.002
string	0.004	0.039
line	0.004	0.002
android	0.002	0.003
manager	0.024	0.002
...	...	...

Examples of such classification techniques are given below. In the example, the class of the `getLine1Number` API [18], which obtains its own telephone number, is determined through the sensitive API classification scheme. The information of `getLine1Number` API is shown in Table I.

As seen in Table I, `getLine1Number` contains package name, class name, API name, and API description. Nave Bayes classification can be applied by using this information as characteristic information. Pre-processing is required before applying the classification. In the case of data composed of a plurality of words, such as an API name, `getLine1Number`, it is divided into words. In this case, `getLine1Number` can be divided into `get`, `Line`, and `Number`. The package and class to which the API belongs also need to be divided into separate words. For API description, this example has the value returns the phone number string for line 1, which divides each word excluding such parts as articles and numbers. Through the pre-processing, a vector is constructed so that there is no overlap. The constructed vector can be expressed as $\{x_1, x_2, x_3, \cdots, x_k\} = \{get, line, number, android, telephony, manager, returns, phone, string\}$.

We perform Naive Bayes classification on the vectors constructed above. We select the vector of learning objects and obtain the conditional probability for each element constituting the learning vector. Table II shows the pre-calculated conditional probability of each API word.

The classification of `getLine1Number` uses Table II. The Naive Bayes formula determines the probability of belonging to the sensitive API class and the general API class. At this time, it is assumed that $p\,(SensitiveAPI) = p\,(NormalAPI) = 0.5$.

1) The probability that the `getLine1Number` API is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2693388, IEEE Access

classified as `sensitive API`:

$$\begin{aligned} p\left(SensitiveAPI \mid X\right) &= \prod p\left(x_i \mid SensitiveAPI\right) p\left(SensitiveAPI\right) \\ &= 0.027 \cdot 0.004 \cdot 0.009 \cdot 0.002 \cdot 0.009 \cdot \\ &\quad 0.024 \cdot 0.027 \cdot 0.009 \cdot 0.004 \cdot 0.5 \\ &= 2.041 \cdot 10^{-19} \end{aligned} \tag{2}$$

2) The probability that the `getLine1Number` API is classified as `normal API`:

$$\begin{aligned} p\left(NormalAPI \mid X\right) &= \prod p\left(x_i \mid NormalAPI\right) p\left(NormalAPI\right) \\ &= 0.004 \cdot 0.002 \cdot 0.002 \cdot 0.003 \cdot 0.002 \cdot \\ &\quad 0.002 \cdot 0.022 \cdot 0.002 \cdot 0.039 \cdot 0.5 \\ &= 1.647 \cdot 10^{-22} \end{aligned} \tag{3}$$

The Naive Bayes classification uses the maximum likelihood method, which classifies data according to increasing likelihood. Since the probability of sensitive API $(2.041 * 10^{-19})$ is higher than the probability of normal API $(1.647 * 10^{-22})$, the `getLine1Number` API is classified as a `sensitive API`.

D. Obfuscation Score Calculation

We design and use static and dynamic extraction methods designed to quantitatively evaluate the obfuscation techniques applied to applications. Using the APIs extracted through the two techniques, the results of the above-mentioned sensitive API classification method are reflected in the quantitative score. The obfuscation score is obtained using the quantitative obfuscation-point arithmetic technique proposed in this paper as below. The function of the variables in the equation is as follows: the *Dynamic API Obfuscation Factor (DF)* is a numerical value that indicates the rate of obfuscated API with reverse engineering resistance among the API that is actually executed dynamically. The *Obfuscation False Negative Factor (OF)* is the number of exposed instruction sets sensitive to obfuscation. The *Static Analysis Disturbance Factor (SF)* is a measure of the disturbance in static analysis.

$$ObfuscationScore = \frac{a \cdot DF - b \cdot OF + c \cdot SF + b}{a + b + c} \tag{4}$$

As in Equation (4), the obfuscation score consists of a dynamic API obfuscation factor, an obfuscation false negative factor, and a static analysis disturbance factor. We use coefficients a, b, and c for the obfuscation score. The static code extraction method and the instruction set that uses dynamic code extraction are shown in Fig. 6.

Each instruction set in Fig. 6 consists of an API called in the application. The instruction set extracted by the dynamic code extraction technique consists of `Instruction Sets A`, `B`, and `C`, and the instruction set extracted by the static code extraction technique is composed of `Instruction Sets B`, `C`, and `D`.

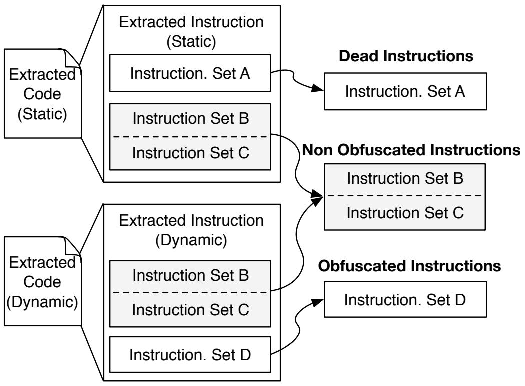


Fig. 6. Composition of instruction sets through static and dynamic code extraction

To calculate each value that constitutes the obfuscation score, we can use the instruction set shown above, and it can be judged whether or not to obfuscate according to whether or not each instruction set is extracted. `Instruction Sets B` and `C` are the intersection of statically extracted instruction set $(A+B+C)$ and dynamic extracted instruction set $(B+C+D)$. The instruction set is observed in both static code analysis and dynamic code analysis, therefore it is not an obfuscated instruction. Also, it means that is has no resistance in terms of dynamic reverse engineering analysis. `Instruction Set D`, which is not observed in statically extracted instruction, is observed on dynamic extracted instruction, so it is necessary to perform dynamic reverse engineering analysis. Thus obfuscation is applied to API with sufficient reverse engineering resistance. `Instruction Set A` is observed in static extracted instructions but not in dynamic extracted instructions. In this case, `Instruction Set A` is a instruction which is not actually executed and can be called dead code and has resistance in static reverse engineering analysis.

Thus, using the instruction set, it is possible to calculate the dynamic API obfuscation factor, the obfuscation false negative factor, and the static analysis disturbance factor. The number of constructors of the API is excluded at the time of calculation, and is calculated by using only the number of APIs that are actually called and perform functions. Moreover, the instruction set extracted by the dynamic code extraction technique contains the system management code, which is also excluded from the calculation.

1) Dynamic API Obfuscation Factor: The DF is a numerical value indicating the ratio of the obfuscated API with reverse engineering resistance to the number of APIs that are executed dynamically. The DF can be expressed by the following Equation (5).

$$DF = \frac{\#ofInstr.D}{\#ofInstr.B + \#ofInstr.C + \#ofInstr.D} \tag{5}$$

`Instruction Set D` is a dynamic obfuscation-based instruction that requires dynamic reverse engineering analysis, as mentioned above. The number of instructions in

`Instruction Set D` can be called # of `Instruction D`, where DF is a code that requires dynamic reverse engineering analysis, and is a measure of resistance to dynamic analysis. Therefore, DF can be calculated by dividing the number of dynamic obfuscated instructions ($\#ofInstructionD$) by the number of extracted dynamic instructions ($\#ofInstructionB + \#ofInstructionC + \#ofInstructionD$).

2) Obfuscation False Negative Factor: The OF is a measure of the percentage of APIs used that are sensitive to APIs that do not have sufficient reverse engineering resistance. It can be expressed by the following Equation (6).

$$OF = \frac{\#ofInstr.B(Sensitive)}{\#ofInstr.B(Sensitive) + \#ofInstr.C} \tag{6}$$

`Instruction Sets B and C`, which are the intersection of the statically extracted instructions and the dynamically extracted ones, are observed in both dynamic and static extraction; thus, obfuscation is not applied in terms of reverse engineering analysis. Dividing the intersection code into a set of sensitive APIs and a set of normal APIs, `Instruction Set B` can be called a `sensitive API set` and `Instruction Set C` a `normal API set`. The set of APIs can be obtained through the sensitive API classification method described above. The number of instructions in `Instruction Set B` is # of `Instruction B`. The OF is a number according to the ratio of APIs used that should be obfuscated in the exposed instruction set. Therefore, OF can be calculated by dividing the number of sensitive instructions ($\#ofInstructionB$) by the number of exposed instructions ($\#ofInstructionB + \#ofInstructionC$).

3) Static Analysis Disturbance Factor: The SF is a measure of the resistance of the analyst to static analysis. The SF can be expressed by the following Equation (7).

$$SF = \frac{\#ofInstr.A}{\#ofInstr.A + \#ofInstr.B + \#ofInstr.C} \tag{7}$$

`Instruction Set A` is not detected in the dynamically extracted instruction that is executed as mentioned above. Since it is a instruction that is not executed, it can be called dead code and is resistant to static reverse engineering. The number of instructions in `Instruction Set A` is # of `Instruction A`, where SF is code that requires static reverse engineering analysis, and is a measure of resistance to static analysis. Therefore, SF can be calculated by dividing the number of static analysis resistance instructions ($\#ofInstructionA$) by the number of extracted static instructions ($\#ofInstructionA + \#ofInstructionB + \#ofInstructionC$).

4) Obfuscation Score Generation: To generate the obfuscation score, we use DF, OF, and SF. Since the DF and the SF are proportional to the observer's score, the factors are added, and the OF is inversely proportional to reverse engineering resistance. It is reflected in the equation by subtraction from the calculation. The corresponding numerical value is expressed by including each coefficient as followed in Equation (8).

$$-b \leq a \cdot DF - b \cdot OF + c \cdot SF \leq a + c \tag{8}$$

The expression has a minimum value of $-b$ and a maximum value of $a + c$. These values are normalized to between 0 and 1 to express it as an intuitive value. Min-Max normalization can be used for normalization which is shown in Equation (9). The obfuscation formula calculated by applying Min-Max normalization is as followed in Equation (10).

$$X_{new} = (X - Min)\,\frac{newMax - newMin}{Max - Min} + newMin \tag{9}$$

$$ObfuscationScore = \frac{a \cdot DF - b \cdot OF + c \cdot SF + b}{a + b + c} \tag{10}$$

Algorithm 1 Calculation for Obfuscation Score

```
$SevSet \leftarrow \{StaticExtractedAPIs\}$
$DevSet \leftarrow \{DynamicExtractedAPIs\}$
$SenApiSet \leftarrow \{SensitiveAPIs\}$
$NonObfuscatedSet \leftarrow null$
$ObfuscatedSet \leftarrow null$
$DeadSet \leftarrow null$
$SystemSet \leftarrow null$
$SensitiveSetgetsnull$
for $api \in \{DevSet \cup SevSet\}$ do
    if api in $DevSet$ && api in $SevSet$ && api not in $SystemSet$ then
        $NonObfuscatedSet \leftarrow NonObfuscatedSet \cap api$
        if api in $SenApiSet$ then
            $SensitiveSet \leftarrow SensitiveSet \cap$ api
        end if
    end if
    if api in $DevSet$ && api not in $SystemSet$ then
        $ObfuscatedSet \leftarrow ObfuscatedSet \cap$ api
    end if
    if api in $SevSet$ && api not in $SystemSet$ then
        $DeadSet \leftarrow DeadSet \cap$ api
    end if
end for
$DF \leftarrow getDF(ObfuscatedSet, NonObfuscatedSet)$
$OF \leftarrow getOF(SensitiveSet, NonObfuscatedSet)$
$SF \leftarrow getSF(DeadSet, NonObfuscatedSet)$
$ObfuscationScore \leftarrow getScore(DF, OF, SF)$
```

Algorithm 1 shows the process of generating obfuscation score described above. The algorithm uses a set of statically extracted APIs, *SevSet*, and a set of dynamically extracted APIs, *DevSet*. It eliminates the statically extracted set through `removeInitMethod` and the dynamically extracted set

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2693388, IEEE Access

through the constructor method, and obtains the system-managed code through getSystemSet on the dynamically extracted instruction using *DevSet*. We then begin API classification using *DevSet* and *SevSet*. As mentioned above, *api* is a dynamically extracted API set and a statically extracted API set. If it is not part of the system management code, the algorithm stores it in *nonObfuscatedSet*. If the API is included in the set of sensitive APIs, it stores the corresponding value in *SensitiveSet*. Similarly, if it is not a system-managed code, it stores api in the *ObfuscatedSetl*; if it is not system-managed code, it stores the *api* in *DeadSet*. This yields DF, OF, and SF using the extracted values; the algorithm then calculates *ObfuscationScore*, which is the obfuscation score, by using the calculated values.

IV. Implementation and Experiments

A. Implementation

The scheme proposed here was implemented as follows. The static code extractor was implemented using the ASMDEX library [19] and the dynamic code extractor by modifying the Dalvik machine in Google's AOSP [20][21]. The sensitive API classifier was applied using the NLTK library [22] and the obfuscation score calculator on Python and Java using a previously extracted API and a sensitive API classifier. MariaDB was used to manage the data used to implement its system.

To determine the coefficient of the obfuscation score, we implemented a sample phone book application with contacts, a phone number list inquiry, phone number addition, and message transmission functions. The application contained dead code, and DexGuard [23], DexProtector [24], and Ijiami [25] were used as obfuscation tools. API hiding, class encryption, and packing were applied. To compute obfuscation scores, 65 of the categories in the Google Play Store were selected for experiments.

B. Coeffcient Determination

To determine the coefficients of each element that constituted the obfuscation score, a phone book application was used. Functional concealment (scope of obfuscation) was applied in three steps on the phone book application and was divided into `One`, `Core`, and `All`. `One` meant that obfuscation was applied as a minimum unit, and that only one of API hiding and class encryption was targeted. The object to be hidden is the API and the class is driven, and does not mean the API and the class on the dead code. In the case of `Core`, obfuscation is applied to the core functions of the application, and both the APIs and classes of some functions are hidden. Finally, in the case of `All`, obfuscation is applied to all classes and APIs in the application, and dead code is included in the scope of obfuscation. If the obfuscation is listed according to the degree of functional concealment, it can be represented by the original application, `API hiding (One)`, `class encryption (One)`, `API hiding (Core)` or `class encryption (Core)`, or `packing` or `API hiding (All)` or `class encryption (All)`. As the degree of functional concealment increased, the coefficient was changed to establish a monotonic relation with the higher score, and the attempt is as shown in Fig. 7.

We modulated the coefficients of a, b, and c of each numerical value, and expressed the numerical values in a graph. In the graphical representation, when the coefficients were reduced in order of a, b, and c, the graph shows a monotonic graph. Finally, when a was 10, b was 2, and c 1, the higher the degree of functional concealment, the higher the score. The dynamic API Obfuscation factor has the greatest impact on the obfuscation score. It can be seen from the above that the API that has the reverse engineering resistance and the obfuscation applied has the greatest influence on the security score. In this test, experiments were continuously performed using the corresponding coefficients.

C. Obfuscation Score

Based on the previously determined coefficients, we experimented with real data. The scheme we provided extracted samples for applications in the top Android rankings for accurate security score measurement. The experiment targeted 13 categories of applications and extracted statically and dynamically used APIs to calculate DF, OF, and SF for the application. At the time of dynamic extraction, it executed such elements as buttons and text, which served as inputs and functions based on the XML file that formed the layout of the target application, and performed the function of the application based on these elements. Static extraction used data generated following DEX parsing on a static extractor. Table III shows the scores for each category.

Table III shows the average values for the categories of DF, OF, and SF by category, as well as the average of obfuscated scores by category. This graph is shown in Fig. 8.

The graph shows the obfuscation scores and distribution by category. The distribution indicates that personal information was handled or included as a high score in the case of categories containing payment-related functions. The biggest obfuscation points on average were in the `music`, `social`, and `financial` categories. The `music` category contained functions such as ringtone payment and MP3 billing streaming, and functional hiding to prevent malicious behavior through tampering was applied in order to infer that `music` recorded a high score. In `social`, personal information was dealt with, which can be interpreted as a degree of functional concealment. Of these, Facebook applications had the highest degree of concealment, since Facebook applies DEX separation and other obfuscation techniques. In the case of `financial`, we determined that it had high functional concealment because it performed functions related to financial assets, such as bank account inquiry and account transfer, and the corresponding value was reflected in the experimental results. Further, a high obfuscation score was derived for an application that performed the security function of the application in the category `tools` and the function managing personal information. The `communication` and `game` categories, to which the messenger application belonged, were expected to yield

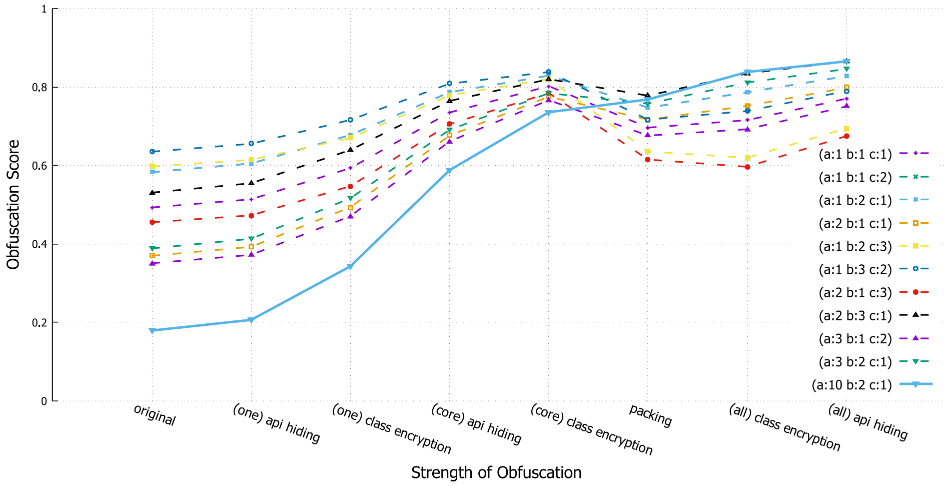


Fig. 7. Coefficient test result

TABLE III
SCORE OF CATEGORIES AND OBFUSCATION SCORES

Category	Dynamic API Obfuscation Factor	Obfuscation False Negative Factor	Static Analysis Disturbance Factor	Obfuscation Score
communication	0.14099	0.09427	0.67331	0.29959
finance	0.26196	0.09858	0.70389	0.39433
games	0.14710	0.11054	0.78629	0.31047
health and fitness	0.18776	0.08258	0.72550	0.34138
media and video	0.13789	0.09234	0.75283	0.30362
medical	0.17202	0.13282	0.73686	0.33781
music and audio	0.36551	0.11724	0.79504	0.47813
photography	0.12272	0.09170	0.68919	0.28716
social	0.28993	0.07715	0.62210	0.41285
sports	0.17954	0.08780	0.76481	0.33728
tools	0.22012	0.11592	0.75794	0.36363
transportation	0.18488	0.10240	0.73675	0.33698
weather	0.16671	0.10288	0.76813	0.32535
Average	0.19978	0.10048	0.73175	0.34835

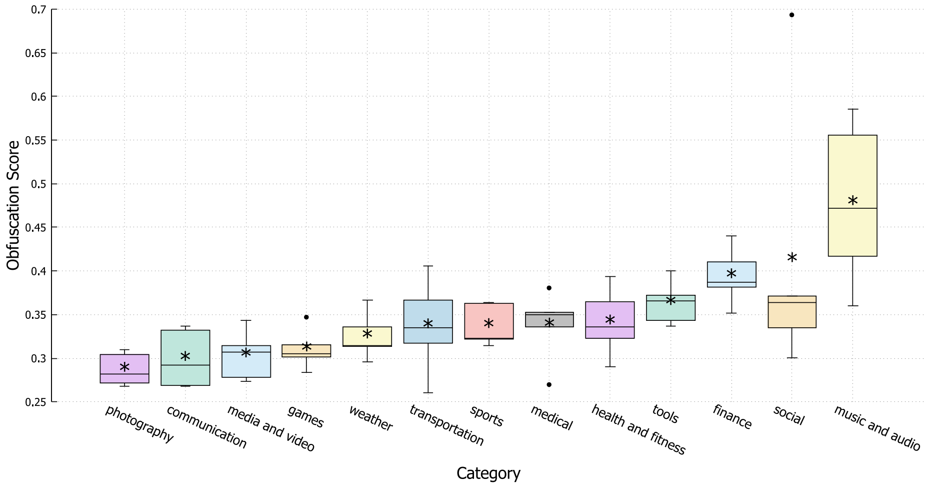


Fig. 8. Obfuscation score and distribution by category

high obfuscation scores, but we confirmed that the messenger application had a relatively low rank, as expected. In the case of game, the function of displaying graphics on the client with data on the server was mainly performed following the native operation. The messenger category also performed some functions through native operations. We could also call the API on the native platform and expect functional hiding on the call. When a native API call was made, the corresponding instruction was also executed on the Dalvik interpreter on the dynamic code extractor. We concluded that the code extracted from the game and communication categories was not concealed well, and was hence not very resistant to reverse engineering in order to prevent tampering. Actually, the low scored applications used many methods that are related to accessing sensitive data and internet access. These methods need to apply more obfuscation technique to prevent malicious behavior.

V. Conclusion

In this paper, we proposed a scheme to quantitatively evaluate obfuscation from the viewpoint of the functional hiding of API calls. To achieve this, we extracted APIs used by applications to be evaluated using dynamic and static code extraction. Based on the extracted API, we can obtain a set with functional concealment: that is, a set of instructions to which obfuscation is appropriately applied, an instruction set without functional concealment, and a set of dead instructions. The DF, OF, and SF were obtained using the extracted set and the sensitive API classification method based on nave Bayes classification, and we generated an obfuscation score calculation formula.

In order to determine the coefficient of each element of the obfuscation score, we applied the obfuscation technique. Coefficients were determined so that the obfuscation technique could be applied to the API to conceal functions from the viewpoint of reverse engineering resistance. That is, the function concealment exhibited monotonicity with a high score. We used the coefficients to measure the obfuscation scores of various applications, and found that applications in such categories as music, finance, and social, which dealt with personal information, recorded high scores.

The quantitative obfuscation evaluation scheme proposed in this paper may not reflect numerically perfect scores. By adding weights to Nave Bayes classification can be a better way to improve the evaluation accuracy of sensitive APIs. More accurate evaluation of sensitive APIs can highly enhance the accuracy of the propose scheme. Scores obtained by using the technique can provide reverse-engineering resistance values that were hitherto unavailable. It is also possible to provide the appropriate obfuscation coverage standard using the score. The proposed scheme is independent of the obfuscation scheme applied through various obfuscation tools. Quantitative and intuitive values can also be provided through normalization. Therefore, it is expected that the proposed scheme can be used as a level evaluation model for future obfuscation techniques by quantifying the degree of vulnerability of mobile applications.

Acknowledgment

This research was supported by the Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029).

References

[1] "Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015." [Online]. Available: http://www.gartner.com/newsroom/id/3165317l

[2] "Android things." [Online]. Available: https://developer.android.com/things/index.html

[3] J. H. Jung, J. Y. Kim, H. C. Lee, and J. H. Yi, "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, vol. 73, no. 4, pp. 1421–1437, 2013.

[4] J. Choi, H. Cho, and J. H. Yi, "Personal information leaks with automatic login in mobile social network services," *Entropy*, vol. 17, no. 6, pp. 3947–3962, 2015.

[5] S. W. Park and J. H. Yi, "Multiple device login attacks and countermeasures of mobile voip apps on android," *Internet Services and Information Security*, vol. 4, no. 4, pp. 115–126, 2014.

[6] T. Cho, G. Na, D. Lee, and J. H. Yi, "Account forgery and privilege escalation attacks on android home cloud devices," *Advanced Science Letters*, vol. 21, no. 3, pp. 381–386, 2015.

[7] Y. Wang, J. Zheng, C. Sun, and S. Mukkamala, "Quantitative security risk assessment of android permissions and applications," in *the 27th International Conference on Data and Applications Security and Privacy*, pp. 226–241, 2013.

[8] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications." *USENIX Security*, vol. 13, no. 20, pp. 527–542, 2013.

[9] D. Kong, L. Cen, and H. Jin, "Autoreb: Automatically understanding the review-to-behavior fidelity in android applications," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 530–541, 2015.

[10] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *the 10th international conference on Mobile systems, applications, and services*, pp. 281–294, 2012.

[11] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752, pp. 41–48, 1998.

[12] D. Pavlov, R. Balasubramanyan, B. Dom, S. Kapur, and J. Parikh, "Document preprocessing for naive bayes classification and clustering with mixture of multinomials," in *the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 829–834, 2004.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, p. 2, 2011.

[14] J. Nagra and C. Collberg, *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[15] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*, vol. 46, no. 1, p. 4, 2013.

[16] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers," in *Security and Privacy*, pp. 659–673, 2015.

[17] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[18] "Android application programming interface reference." [Online]. Available: http://developer.android.com/reference

[19] "Asmdex." [Online]. Available: http://asm.ow2.org/asmdex-index.html

[20] "Android open source project." [Online]. Available: https://source.android.com/source/index.html

[21] D. Bornstein, "Dalvik vm internals," in *Google I/O developer conference*, pp. 17–30, 2008.

[22] "Nltk." [Online]. Available: http://www.nltk.org

[23] "Dexguard." [Online]. Available: http://www.guardsquare.com/dexguard

[24] "Dexprotector." [Online]. Available: https://dexprotector.com

[25] "Ijiami." [Online]. Available: http://www.ijiami.cn

Taejoo Cho received the B.S. and M.S. degree in Computer Science and Engineering from Soongsil University in 2014 and 2016. His research interests include Android, Mobile Security and Application Assessment.

Hyunki Kim received the B.S. degree in Computer Science and Engineering from Soongsil University in 2015. Currently he is taking a masters course at Graduate School of Software Convergence, Soongsil University. His research interests include Android, Mobile security and Application Assessment.

Jeong Hyun Yi is an Associate Professor in the School of Software and a Director of Mobile Security Research Center at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, IoT security, and applied cryptography.