

A Power-Efficient Approach to Detect Mobile Threats on the Emergent Network Environment

CHIA-MEI CHEN¹, YI-HUNG LIU², ZHENG-XUN CAI¹, AND GU-HSIN LAI³

¹Department of Information Management, National Sun Yat-sen University, Kaohsiung 804201, Taiwan

²Department of Computer Science and Information Management, Soochow University, Taipei 11102, Taiwan

³Department of Technology Crime Investigation, Taiwan Police College, Taipei 116081, Taiwan

Corresponding authors: Yi-Hung Liu (randyliu@scu.edu.tw) and Chia-Mei Chen (cmchen@mis.nsysu.edu.tw)

ABSTRACT Mobile and IoT devices are blooming, and their applications are prevailing worldwide. In the meantime, the Industry 4.0 trend converges industrial control systems with the internet environment, which makes it vulnerable. Mobile and IoT applications provide seamless connectivity to the emergent network environment for daily work. As mobile phones contain privacy information, malicious mobile applications could compromise mobile devices and cause financial losses. Moreover, attackers could launch DDoS attacks to exhaust a resource of mobile or IoT devices in the emergent network environment. Static malware analysis consumes less computing resources and power than dynamic analysis. This study proposes a power-efficient solution to identify mobile threats, which applies taint analysis to protect the emergent network environment. The experimental results show that the proposed approach could detect mobile malware efficiently.

INDEX TERMS Internet of Things, malware detection, static analysis, cyberattacks.

I. INTRODUCTION

The Industry 4.0 revolution has dramatically transformed how manufacturing and industrial companies operate. The traditional industrial control systems were deployed in isolation and without security control. As the new emergent network environment interconnects multiple networks such as the Internet, wireless networks, cellular networks, and corporate internal networks, it dramatically expands the attack surface and increases the potential security risk [1].


Mobile devices and Internet of Things (IoT) are growing at a rapid speed. The total number of devices connected to the network will grow to 50 billion in 2020 [2]. A spending guide from International Data Corporation [3] forecasted that IoT spending will experience a compound annual growth rate of 13.6% and reach \$1.2T in 2022. IoT technology is being applied to various fields such as safety, transportation, industrial, healthcare, and building.

IoT devices make the boundary between cyberspace and real-world disappear and IoT related cyberattacks can be extended to the damage of real-world [4]. On the other hand, the use of smartphones is increasing daily as well, and they are extensively integrated with the emergent network environment. IoT providers integrate smartphones to provide

ubiquitous computing and to increase the computing services of IoT networks. Mobile phones have become a control panel for managing IoT environments [5].

While IoT smart devices provide more features, they also introduce new security threats. A common perception is that IoT devices are protected from Internet attacks by the perimeter security offered by home routers or industry internal networks. Attackers could steal privacy information via printers [6] or take control of power switches [7] remotely. The survey [8] shows that mobile malware is one of the major issues in the IoT security. Mobile users would expect an increase in mobile malware, particularly on the Android platform, as it held over 85% of the total market share on mobile devices [9]. The great volume of Android devices now becomes targets for attackers.

The development of demand for seamless connectivity among IoT devices and networks provides ubiquitous computing. On the other hand, the vulnerabilities are normally considered for large infrastructures and little attention was paid to the cyber threats from mobile devices. The mobile security issues are not too dissimilar to those already affecting traditional IT networks. The most significant issue is the emergence of traditional malware such as viruses, worms, Trojan horses, and rootkits. Malicious software in this context behaves similarly to the same threats on traditional IT networks. Mobile malware may be targeted at

The associate editor coordinating the review of this manuscript and approving it for publication was Ilun You .

exfiltrating sensitive data or leveraging the compromised asset to access sensitive information in the emergent network environment.

Mobile developers offer software for free in order to collect user information for data analytics and marketing purposes. An example might be a free social network application for users to communicate with friends and to collect users' contact list for marketing purposes. This behavior is not always malicious to the users, but the users might not be aware that their information is utilized for other purposes.

Mobile users store their personal information such as usernames and passwords on their phone so that they can work efficiently such as checking emails or online banking at anytime and anywhere. Mobile malware FakeToken attacked financial applications as well as apps for booking taxis, hotels, tickets, etc. [10] Mobile malware family FakeInstaller sends SMS messages to premium rate numbers without the user's consent [10]. Therefore, data leakage is a serious security threat to mobile devices.

Anti-virus software is a common static analysis approach for detecting malware with signatures that are efficient for identifying known malware. The study [11] shows that Android turned out to be the fourth architecture with the newest variants of malware. Signature-based approaches might not be able to identify mobile malware efficiently; behavior-based detection approaches could identify variants of malware as long as they exhibit similar malicious behaviors.

According to the McAfee Mobile Threat Report [12], in order to bypass the detection in app stores, mobile malware might infect mobile phones by communicating directly with users via SMS. Some malware might execute a malicious external program. Three samples below illustrate the malicious behaviors aforementioned. The first sample code in Figure 1 outlines that the mobile malware constructs and sends out an SMS text message. The SMS message could contain a malicious link and then infect the victim's contact list. The second one shown in Figure 2 acquires the victim's privacy information, constructs a URL with the sensitive information, and connects and reports to the command and control server through the URL. The HTTP response from the command and control server may contain malware download or attack command such as exploiting IoT devices or infecting more devices. The third sample code in Figure 3 first gets a malicious program, writes it into an array, and then executes this external program.

There are two common approaches for malware detection: dynamic and static analysis. Dynamic analysis executes a program sample in a controlled environment (sandbox) to observe the execution behaviors and to identify malware. It requires a correct input sequence to trigger malicious behaviors and would consume a lot of computing resources in order to capture the execution behaviors. To evade detection, malware might stay dormant in such a sandbox environment or can become active only by specific inputs or conditions.

```

1 // Define function which invokes API function call to send a text message
2 function sendsms(String s, String s1, String s2, Context context)
3 {
4 ...
5 SmsManager.getDefault().sendTextMessage(s, null, s1,
6   PendingIntent.getBroadcast(context, 0, new Intent("SMS_SENT"), 0),
7   PendingIntent.getBroadcast(context, 0, new Intent("SMS_DELIVERED"), 0));
8 ...
9 }
10 // program starts execution from here
11 function onStart
12 {
13 ...
14 }
15 }

```

FIGURE 1. An illustration of sending an SMS message.

```

1 //Class AirHornSoundService method getUserInfo collects user info and stores into variables which pass to
2 another class.
3 class AirHornSoundService method getUserInfo {
4 ...
5 deviceId = telephonymanager.getDeviceId();
6 mobile = telephonymanager.getLine1Number();
7 country = telephonymanager.getNetworkCountryIso().toUpperCase();
8 carrier = telephonymanager.getNetworkOperatorName();
9 ...
10 }
11 // Class AirHornSoundService method handleCommand constructs notifier with user info.
12 class AirHornSoundService method handleCommand{
13 ...
14 notifier = new Notifier(deviceId, mobile, country, carrier, email);
15 ...
16 }
17 //Class Notifier method Notifier passes the user info to URL and invoke read() to make the HTTP connection.
18 class Notifier method Notifier(String s, String s1, String s2, String s3, String s4){
19 ...
20 }
21 //Variable params constructs a string of user info
22 params = "appid=1&deviceId="
23   .append(s).append("&mobile=").append(s1)
24   .append("&country=").append(s2)
25   .append("&carrier=").append(s3)
26   .append("&email=").append(s4).toString();
27 //Variable pollURL is a constructed URL with user info which is sent to read()
28 pollURL = "http://www.ty3studios.com/android_notifier/notifier.php?"
29   .append(params).toString();
30 ...
31 read(pollURL);
32 ...
33 }
34 // Class Notifier method read() sends the URL request and receives the response from the server.
35 // URL(s) is an object of URL string; method openStream() opens the URL
36 // InputStreamReader receives the response from the URL
37 // BufferedReader stores the info into buffer
38 Return new BufferedReader(new InputStreamReader((new URL(s)).openStream()), 8192);
39 ...
40 }

```

FIGURE 2. An illustration of stealing user privacy information.

Taint analysis or taint tracking can track the information flow dynamically or statically. It tracks sensitive "tainted" information of a target application by starting at a pre-defined source (e.g. a method of getting device ID as shown in Figure 2) and following the data flow until it reaches a given sink (e.g. a method writing the information to an URL as shown in Figure 2), the information about where data may be leaked to [13].

```

1 //get a malicious APK file via AssetManager and use InputStream write malicious file to array
  byteArr.
2 AssetManager assetmanager = getAssets();
3 InputStream inputstream = assetmanager.open("malicious_file_name");
4 byte byteArr[] = new byte[inputstream.available()];
5 inputstream.read(byteArr);
6 //create temp file "extract.tmp" in path "rootDirPath"
7 File file = File.createTempFile("extract", "tmp", "rootDirPath");
8 //use FileOutputStream to write content of byteArr[(malicious) into a file
9 FileOutputStream fileoutputstream = new FileOutputStream(file);
10 fileoutputstream.write(byteArr);
11 //call runtime to execute the external malicious program
12 Runtime.getRuntime().exec("rootDirPath"+file.getAbsolutePath());

```

FIGURE 3. An illustration of external program execution.

Static analysis analyzes malware code without execution and consumes fewer resources and less power. Static analysis is a power-efficient approach. Commercial anti-virus is a static analysis by signature matching, but the detection rate relies heavily on the signatures. A new variant of malware could bypass such detection easily, making mobile devices vulnerable.

Taint analysis could identify malware variants as it tracks data movement and does not rely on signatures. Most taint analysis researches focus on data leakage and some are dynamic analysis based. The proposed taint analysis solution could identify data leakage as well as remote execution.

Based on the literature review, it can be observed that concluded that (1) malware variants grow fast; (2) most mobile malware targets on information stealing, sending SMS messages, and remote execution; (3) a power-efficient mobile malware detection is required. The literature review suggests that static analysis approaches are suitable for detecting mobile malware as mobile devices have limited power and computing resources and that taint analysis is good at identifying malware variants. By combining the above two approaches, this study proposes a static-based taint analysis approach to identify misbehaviors and malware variants for mobile devices.

The proposed detection method applies reverse engineering to reconstruct binary applications into source code form, builds the relationships among the accessed data and API calls, and detects anomalous data flow movements by taint analysis.

To establish the relationships among the data and API calls, this study proposes a data model that explores the aliasing and the linkage relationships in multiple levels. Most taint checking addresses information leakage as it tracks information flow. The proposed approach extends it by tracking data as well as methods so that untrusted external execution can be captured.

To the best of our knowledge, this is the first attempt to build a data model for connecting the relationships among the objects, methods, and classes referenced by a mobile application and to develop threat patterns by using a data model and static taint analysis to identify unknown mobile malware.

In summary, this article has the following contributions: 1. A static taint analysis detection method is proposed for identifying malicious mobile applications; 2. The proposed approach could identify unknown malware variants; 3. A multi-level data model is proposed, which extends the taint analysis to identify not only information leakage but external program execution.

The rest of the paper is organized such that the related work is studied in Section 2. The detection approach is discussed in Section 3 followed by performance evaluation in Section 4. The conclusions are drawn in Section 5.

II. RELATED WORK

Dynamic analysis executes applications in a controlled environment in order to observe the behaviors. Isohara *et al.* [14] performed the dynamic analysis under a sandbox with a modified kernel in order to collect the kernel behaviors including system calls, I/O events, and process events. They summarized sixteen signatures in regular expressions to identify suspicious mobile applications. Event logs collected by Isohara's need to be transferred to a computer for further investigation.

Bläsing *et al.* [15] designed a sandbox for executing mobile applications; an analysis method is needed to identify anomalous behaviors collected from the sandbox. Iqbal and Zulkernine [16] proposed a monitoring mechanism that records system call invocations of the running processes and detects suspicious ones according to the pre-defined policies. Shabtai *et al.* [17] proposed a dynamic analysis framework and evaluated several combinations of anomaly detection algorithms and feature selection methods in order to find the combination that yields the best performance. Dini *et al.* [18] proposed a two-level anomaly detection with two-level feature sets: the kernel and application level. The system proposed by Dini needs to be installed on smartphones in order to collect the required features.

Static analysis is a white-box approach in which a target sample is de-compiled into a source code format for further analysis. Some code might apply obfuscation to prevent de-compilation, and the official Android developer website suggests to obfuscate code by renaming identifiers. Tools [19] are available for the purpose, while obfuscation can be de-obfuscated by some open source tools such as DeGuard which is based on a statistical de-obfuscation model to recover method and class names [20], [21].

Android mobile applications should request permission for accessing data, network, or certain system features. Permission-based approaches examine the Android Manifest file to investigate malicious characteristics. Cerbo *et al.* [22] applied an association algorithm, Apriori, to analyze requested permissions and those that an app actually accesses. An application is identified as malicious if the later set is not a subset of the requested. Takayuki *et al.* [23] proposed a risk-score method to assess mobile applications, which considers permissions requested, download rates, and user ratings.

Almin and Chatterjee [24] applied Naive Bayes classification approach by using the requested permissions as

the primary features. Şahin *et al.* [25] proposed a weight function based on permission and applied Naive Bayes and K-nearest Neighbor to identify malware. López and Navarro *et al.* [26] proposed a framework of static analysis based on the requested permissions. The study evaluated multiple machine learning classification approaches and concluded that KNN, SVM, and decision tree perform better.

AppProfiler [27] creates a profile of application behavior and attempts to provide more information than permission by considering system logs. Feizollah *et al.* [28] evaluated how effective is Android intents and permissions as a feature to identify mobile malware. They concluded that intents or permissions should be used with other known promising features.

Adopting the requested permissions to identify mobile malware is easy but not adequate, as applications could execute a privilege without permission request. Likewise, applications could access sensitive data without requesting [29]. In summary, permission-based approaches are not efficient.

Static analysis based features can be broadly categorized into the following types [30]: API, function calls, code structures, sources and sinks, bytes, strings, and permission. Some studies consider API call invocations as important information for identifying malicious behaviors. Yerima *et al.* [31] chose important API calls and system calls by using feature selection and applied the Bayesian classification approach for detection. Wu *et al.* [32] combined permissions and API calls and applied the K-means algorithm to classify mobile applications.

Firdaus *et al.* [30] applied genetic search (GS) to retrieve the optimal and smallest set of static analysis based features, and the features considered are string-based features including permission, the words in the double quotes, function calls, system commands, and directory paths. Their experiments indicate that the detection results after GS out-performs that before GS.

Milosevic *et al.* [33] evaluated the performance of different machine learning models for detecting malware. They concluded that static analysis of source code has higher detection rate than that of permissions and classification models outperform clustering models.

Like other dynamic analysis approaches, dynamic taint analysis executes programs in a virtual environment [34], where it monitors and tracks the flow of sensitive data (aka tainted data), such as users' private information or network communication [35]. TaintDroid [36] implemented the multiple levels of tracking: variable, message, method, and file to ensure persistent information conservatively retains its taint markings. TaintChaser [37] is an improvement work of the above work by testing Android software automatically. The study claimed that it could identify more privacy leakage than the previous work. Most dynamic taint analysis solutions require to be installed and executed on smartphones and consume a lot of computing resources,

such as computing, storage, network, and battery. Therefore, mobile users would experience performance downgrade during analysis and detection. Moreover, advanced malware may circumvent dynamic analysis detection by applying anti-analysis approaches such as anti-rooting, anti-emulating, and anti-debugging [38].

Static taint analysis tracks data with taint tags by following them to their operations also known as sinks. To build the mapping relation between the data and the operation, call graph and control flow graph (CFG) are common approaches along with static taint checking. Static taint checking based approaches [39], [40] were proposed to detect information leakage.

FlowDroid [13] is a static taint checking framework which computes one call graph for each component of the tested application and extends it to include callback functions and method invocations. While this method is more expensive than just scanning for classes implementing the callback interfaces. AndroidLeaks [41] finds potential leaks of sensitive information in Android applications on a massive scale. It creates a permission mapping, a mapping between Android API calls and required permissions and generates a call graph to determine the methods which invoke sensitive methods.

DroidChecker [42] detects Android capability leaks by CFG. The study demonstrated that, with few extra lines of code, an Android application could access a contact list without any permission. DroidSafe [43] combines static and dynamic analysis, where a combination of analysis that scan statically resolve communication targets identified by dynamically constructed values such as strings and class designators. The analysis does not have a fully sound handling of Java native methods, dynamic class loading, and reflection. Based on the literature review, taint tracking based studies mainly focus on information leakage.

III. THE PROPOSED APPROACH

Malicious behaviors of mobile applications mainly can be categorized into three types: sending an SMS message, information leakage, and remote execution as mentioned and illustrated in the introduction. The proposed method adopts static analysis and hence applies reverse engineering technology [44] to retrieve the source code of a target mobile application. However, the source code from reverse engineering loses important information. To overcome the problem, this study proposes a multi-level data model which reestablishes the linkage among the classes, function calls, and data objects. By using the data model, the proposed detection method can track the movement of sensitive data as well as the invocation of sensitive function calls to identify the above suspicious behaviors.

The proposed detection method consists of three phases. The first phase reconstructs an executable mobile application bytecode into source code by reverse engineering technology. The second phase builds up a data model by exploring the relationships among the classes, objects, function calls, and data entities of the target code. The last phase tracks

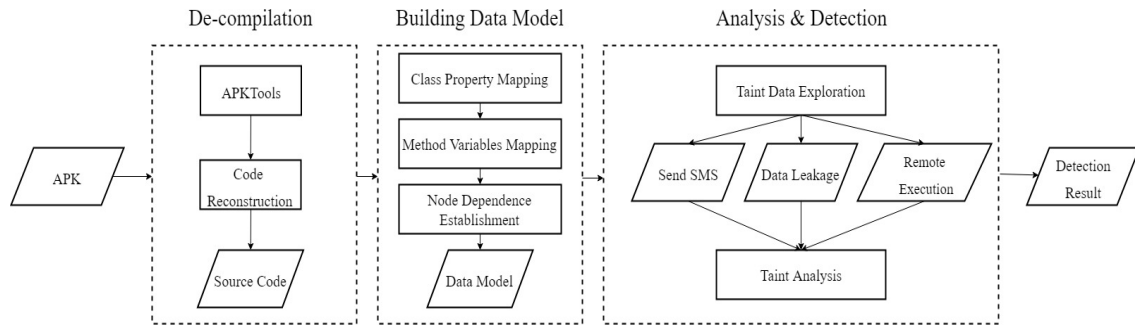


FIGURE 4. The proposed detection method.

sensitive data movement and sensitive function call invocation by using taint analysis and identifies if sensitive information is tainted or not. The system architecture of the proposed approach is depicted in Figure 4.

Phase 1 (De-Compilation): An Android application is in .apk file extension form; apk stands for Android Application Package. An .apk file is an installer package file, containing the following information: app resources, manifest, signature, as outlined in Figure 5. The code reconstruction is to disassemble a given mobile application into java source code. Several open source solutions can be applied such as apktool, dex2jar, and java decompiler. Apktool decodes resources to the original form and transforms Dalvik bytecode (classes.dex) into Smali source; dex2jar converts Dalvik bytecode to java bytecode (.jar); a java decompiler, such as jadx, JDCore, or JAD, decompiles java bytecode to source code.

名稱	大小	封裝後大小
assets	47 207 961	15 198 933
com	17 354	4 101
lib	42 457 676	18 031 607
META-INF	258 768	81 032
org	28 755	11 103
res	2 299 680	1 959 919
AndroidManifest.xml	39 648	8 012
classes.dex	7 221 448	2 928 941
resources.arsc	837 464	837 464

FIGURE 5. Anatomy of an Android app.

Figure 6 illustrates a sample of de-compilation. Each step of de-compilation loses some important information; transforming Dalvik bytecode to java bytecode loses important metadata. Therefore, the next phase of the proposed approach reestablishes the missing relationships among classes, methods, and data objects.

Phase 2 (Building Data Model): This phase examines the source code obtained from de-compilation and creates a data model that builds the relationships among the objects accessed by the code. Based on the object-oriented concept, the proposed multi-level data model consists of three levels of objects: class, method, and variable, as illustrated in Figure 7. Each instance of an object is represented as a node; each level of a node is associated with a set of attributes. The classes and

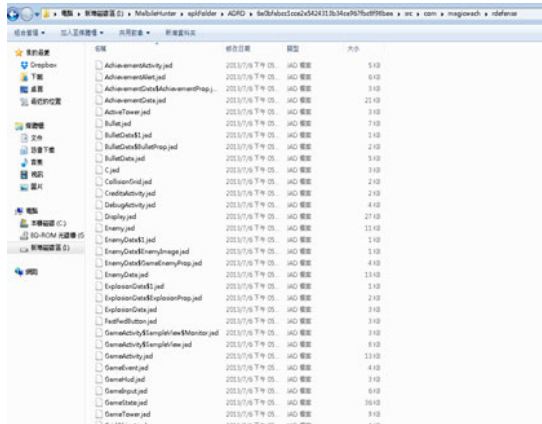
methods in the data model are those defined or accessed by the target code; the variables represent the local variables and parameters referenced by a given method.

New Android operating systems release frequently; new API functions are defined and some dated ones might be removed. To keep the API functions up to date, the proposed method designs a crawler that automatically and periodically fetches APIs from the official developer website, develop.android.com. The crawler first creates a list of web links obtained from the web page of the official online Android documents, grabs the web page from a link in the list recursively, and retrieves all links from each visited web page, adds new unvisited links in the list for further exploration. The API names can be captured from the link by removing the predetermined prefix and postfix.

During the exploration of finding the relationships among classes, methods, and variables, the proposed approach labels sensitive data and sensitive methods in order to track the movement at the next phase. Besides losing some important information after de-compilation, the reverse-engineered source code usually contains many “goto” statements as illustrated in Figure 8. Even though it loses the sequence of the control flow, but the relationships among nodes remain and can be discovered from the code.

To build the relationships among the three levels of nodes in the data model, the proposed approach concludes the following three cases of code execution action: 1. method invocation; 2. variable assignment; 3. flow control. In the case of method invocation, the proposed approach adopts regular expression representation approach to extract method invocation, then examines the parameters and variables of the method. Figure 9 illustrates a case of method invocation; in this sample, the proposed detection approach locates method node mthdB and builds the linkages between variable nodes a, b, c and d, e, f.

In some cases of variable assignments, it is performed through method invocation, as shown in Figure 10. Besides building the linkage of method invocation, the proposed approach also establishes that of the return value of the method with the target variable node. The above two cases (method invocation and variable assignment) might interleave with the control flow statement, as illustrated in Figure 11. Regular expressions are applied to identify this



(a) List of files generated

```
public boolean onOptionsItemSelected(MenuItems menuItem) {
    menuItem.getItemId();
    JVM INSTR tableswitch 1 2: default 28.
    //      1 32;
    //      | 2 45;
    goto _L1 _L2 _L3;
_L1:
    boolean flag = false;
_L5:
    return flag;
_L2:
    SaveSettings();
    finish();
    flag = true;
    continue; /* Loop/switch isn't completed */
}
```

(b) Source code

FIGURE 6. An illustration of de-compilation result.

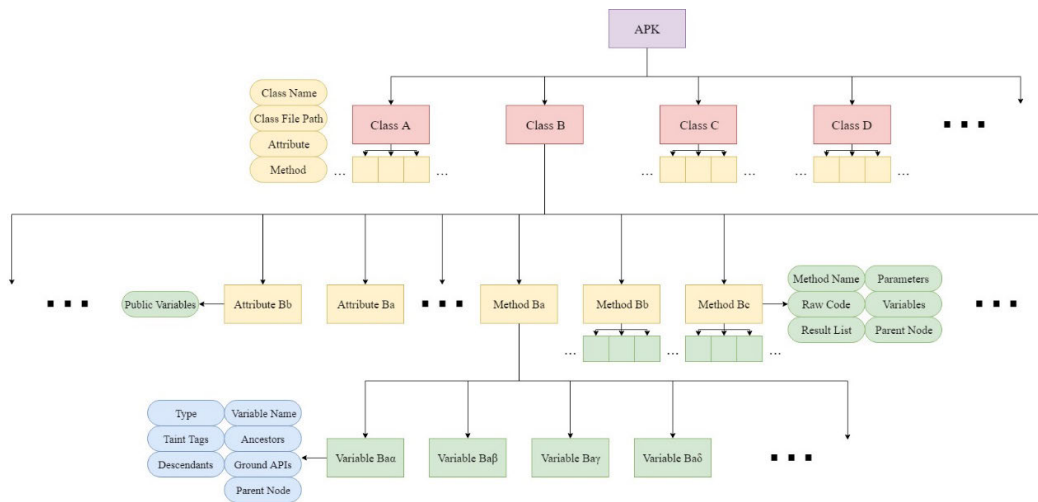


FIGURE 7. The proposed multi-level data model.

```
public boolean onOptionsItemSelected(MenuItems menuItem)
{
    menuItem.getItemId();
    JVM INSTR tableswitch 1 2: default 28
    //      1 32
    //      2 45;
    goto _L1 _L2 _L3
_L1:
    boolean flag = false;
_L5:
    return flag;
_L2:
    SaveSettings();
    finish();
    flag = true;
    continue; /* Loop/switch isn't completed */
_L3:
    SaveSettings();
    RebootRecovery();
    finish();
    flag = true;
    if(true) goto _L5; else goto _L4
_L4:
}
```

FIGURE 8. A piece of decompiled sample code with many goto statements.

Sample code	Comment
mthdB(a,b,c); ...	Define method mthdB which passes a, b, c as parameters.
public void mthdB(obj d,obj e,obj f) ...	Invoke method mthdB and pass d, e, f as parameters.

FIGURE 9. A sample of method invocation.

Sample code	Comment
varA = mthdB(a,b,c); ...	Method mthdB passes a, b, c as parameters and the result is assigned to variable varA.
varD = objB.mthdC(d,e,f); ...	Method mthdC of object objB passes d, e, f as parameters and the result is assigned to variable varD.

FIGURE 10. A sample of variable assignment.

case and then the linkages among the nodes are constructed accordingly.

Extracting method invocation or API calls is an important part of the proposed static taint analysis. There are 3 types of method invocations: methodA(a,b,c), objectA.methodA(a,b,c), and methodA(methodB()), where a,

b, and c are method parameters, methodA and methodB represent a method (API function call), and objectA represents an object. The first type, methodA(a,b,c), is a static method invocation; the second type, objectA.methodA(a,b,c),

Sample code	Comment
for (int i=0;i<n;i++) arrayA[i]=arrayB[i];	Variable assignments in a for loop.
...	
While (doSomething());	Method invocation in a while loop
...	

FIGURE 11. A sample of control flow.

is invoking a method from an object; the third type, methodA(methodB()), is a recursive invocation. To extract API calls from a target application efficiently, the proposed method adopts regular expression to discover the invocations, and the algorithm is outlined in Figure 12.

```
nString = "a-zA-Z_0-9"
oString = "\\+\\-\\*\\/\"
pattern = '['+ nString +']+(\\.['+ nString +'])*\\(['+ nString +
oString +']\\.\\', ]+\\);?'
// objectA.methodA(parameter);
// parameter might be another method code: objectB.methodB()...
for sourceCode in Application:
    for line in sourceCode:
        if regex(line, pattern):
            methodCalls.append(line)
```

FIGURE 12. Algorithm of API call extraction.

The parameter list of a method is important for taint analysis. Hence, the proposed method extract method declaration (definition) in order to understand the method parameters. The syntax of method declaration (definition) begins with the prefix of “public/protected/private” and the rest is similar to that of invocation. The algorithm of extracting method declaration is outlined in Figure 13; like the method invocation, the algorithm scans the code line by line and searches for the matched patterns.

```
for method in methodCalls:
    if "public/protected/private" in method:
        methodDeclaration.append(method)
```

FIGURE 13. Algorithm for extracting method declaration.

Phase 3 (Taint Analysis): Taint analysis [45] can be seen as a form of information flow analysis and tracks information by the concept of source to sink. Information flows from node x to node y, denoted $x \rightarrow y$, when information stored in x is transferred to y, as depicted in Figure 14. It uses tags or labels to track the flows. If an operator uses the value of a tagged node x to derive a value of another node y, y becomes “tainted”. Taint operator is denoted as t(); $x \rightarrow t(y)$ represents node x taints node y. Taint operation is transitive; if $x \rightarrow t(y)$ and $y \rightarrow t(z)$, then $x \rightarrow t(z)$.

Most taint analysis addresses information leakage by its nature concept. The proposed approach extends it by tracking data as well as methods so that untrusted external execution can be captured. The proposed approach generalizes the concept of information flow by tracking the movement

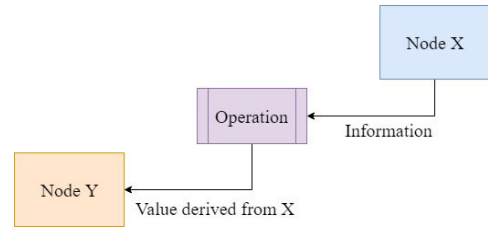


FIGURE 14. Information flow.

of sensitive data and the invocation of sensitive methods. Tainted sensitive nodes in the proposed data model can be data or method. In this research, all the sensitive data and methods are tagged, where tags are used to track the influence of tagged nodes.

The taint operators defined in the proposed research include assignments and sensitive methods. Three types of malicious behaviors are considered in the proposed research: sending an SMS message, executing an external application, and information leakage; the associated methods and data sources are summarized in Table 1.

TABLE 1. Taint analysis.

Sending an SMS message		
Threat	Source data	Sensitive method
Send messages to a fixed number (String)	Constant variables (String)	sendTextMessage() sendMultipartTextMessage()
Send messages to a fixed number (Int)	Constant variables (Integer) toString()	
Send messages from a URL	HttpClient.execute() HttpResponse.getEntity() toString()	
Executing an external application		
Threat	Source data	Sensitive method
Execute external program	AssetManager.getAssets() AssetManager.open() FileOutputStream.write()	Runtime.exec()
Information leakage		
Threat type	Source data	Sensitive method
subscriber info	TelephonyManager.getSimSerialNumber() TelephonyManager.getDeviceSoftwareVersion() TelephonyManager.getLine1Number() TelephonyManager.getSubscriberID() TelephonyManager.getDeviceID()	HttpClient.execute() URL.openStream() URL.openConnection() URL.getContent()
call history	getContentResolver()	
contact info	getContentResolver() ContactsContract.*	
message info	getContentResolver() ContentResolver.query()	

In this phase, the proposed approach scans the tagged (tainted) nodes of the data model based on the data sources listed in Table 1 and propagates the tainted nodes throughout the data model by bread-first search. The taint analysis algorithm is explained in Figure 15. Visiting every method node in the data model, this phase inspects if a variable node of a sensitive method is tainted. The proposed taint analysis-based approach tracks the data and function calls accessed by a mobile application; therefore, it could identify if the target mobile application contains the potential security threats mentioned above.

```

Input:
taintEntryList - a set of tainted variables.

while taintEntryList not empty:
    taintEntry = taintEntryList.pop()
    for descendant in taintEntry.descendantList:
        if descendant.taintTagList != taintEntry.taintTagList:
            descendant.taintTagList = taintEntry.taintTagList

    taintEntryList.append(descendant)
    
```

FIGURE 15. Taint analysis algorithm.

As for the time complexity, the proposed system consists of de-compilation, API invocation extraction, and tainted analysis. It parses the code line by line three times to reconstruct the source code, builds the data model, and extract API calls. Therefore, the time complexity of the first two parts is $O(n^3)$. The time complexity of taint analysis is $O(n^3)$, according to the original study of taint analysis [46]. The taint analysis in the proposed method involves the relationships among variables, methods, and objects, so the time complexity of taint analysis is $O(n^3)$ as well. In summary, the time complexity of the proposed malware detection method is $O(n^3)$.

IV. PERFORMANCE EVALUATION

This study conducted three experiments for system validation and performance evaluation. The purposes of the experiments are stated as follows: (1) to evaluate if the proposed detection approach can identify malware correctly and efficiently; (2) to evaluate if the proposed detection approach could identify benign applications correctly; and (3) to compare the detection performance of new malware variants with the commercial anti-virus software.

The malware samples were extracted from Zhou *et al.*'s research [47], which were collected for more than one-year effort including manual or automated crawling from a variety of Android Markets. 26 malware families with a total of 305 samples were included in this evaluation. The research categorized the malicious behaviors of each family including privilege escalation, remote control, and information leakage as summarized in Table 2. The benign samples were retrieved from Google Play Market with the most popularity from various types of mobile applications, ranging from games, reader tools, translators, and photographic tools.

TABLE 2. The profile of the tested malware families.

Malware family	Privilege Escalation	Remote control (SMS/net/phone)	Info leakage
ADRD			√
Asroot	√		
BeanBot			√
Bgserv			√
CoinPirate			√
CruseWin			√
DogWars			√
Endofday			√
FakePlayer			√
Geinimi			√
GGTracker			√
GingerMaster	√		√
GoldDream			√
Gone60			√
GPSSMSpy			√
HippoSMS			√
Jifake			√
NickiSpy			√
Pjapps			√
RogueLemon			√
RogueSPPush			√
SndApps			√
Spitmo			√
Walkinwat			√
YZHC			√
Zsone			√

TABLE 3. The results of Experiment 1.

Malware family	Detected/Samples	Detection rate
Total Average	290/305	95.1%
ADRD	22/22	100%
Asroot	7/8	87.5%
BeanBot	8/8	100%
Bgserv	9/9	100%
CoinPirate	1/1	100%
CruseWin	2/2	100%
DogWars	1/1	100%
Endofday	1/1	100%
FakePlayer	6/6	100%
Geinimi	65/65	100%
GGTracker	1/1	100%
GingerMaster	4/4	100%
GoldDream	35/42	83%
Gone60	9/9	100%
GPSSMSpy	6/6	100%
HippoSMS	4/4	100%
Jifake	1/1	100%
NickiSpy	2/3	66%
Pjapps	50/56	89%
RogueLemon	1/1	100%
RogueSPPush	9/9	100%
SndApps	10/10	100%
Spitmo	1/1	100%
Walkinwat	1/1	100%
YZHC	22/22	100%
Zsone	12/12	100%

Experiment 1: Experiment 1 consists of two parts: the first part evaluates the detection efficiency of the proposed solution against a malware data set and the second part validates the detection correctness by examining two malware samples manually. According to the detection results delineated in Table 3, the proposed approach could identify the malware

efficiently with the detection rate of 95.1% and could detect the variances efficiently in most cases.

To further validate the correctness of the proposed detection approach, two samples were inspected manually and examined by the proposed system. Malware sample 1 sends out information through SMS message; sample 2 leaks private information through the network. The graphic explanation of the two samples is depicted in Figures 16 and 17. Malware sample 1 first defines a method “sendsms” that invokes an Android API method “sendTextMessage” to send out information through SMS message. Another user-defined method “onStart” is an entry point of the malicious behavior that accesses a sensitive tainted data source. By tracking the information flow to the invocation of the method “sendsms” and then “sendTextMessage”, the proposed system identifies the misbehavior correctly.

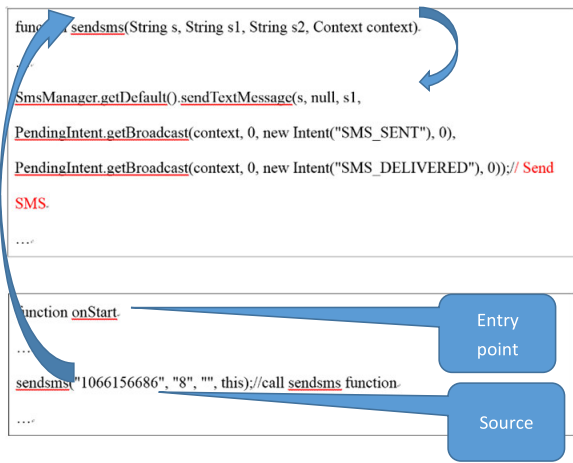


FIGURE 16. Malware sample 1.

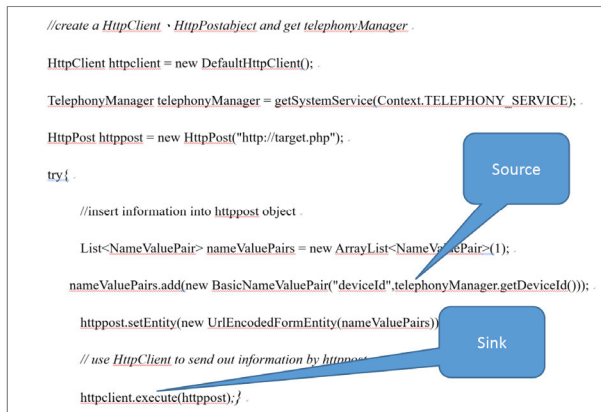


FIGURE 17. Malware sample 2.

Figure 17 outlines how malware sample 2 sends out confidential information via the HTTP protocol. It fills in the content of an HTTP POST by creating an object HttpClient whose parameter invokes the method “TelephonyManager.getDeviceID()” to get the device ID, sensitive tainted data. By tracking the source of the tainted data, the information leaks out by invoking the method “HttpClient.execute()”.

TABLE 4. The results of Experiment 2.

Mobile application	IMEI info leak
akunososhiki.app.flyyyHero-13	
BattleCats	Y
com.adobe.air-3500106	
com.adobe.reader-69805	
com.ainput.activity0-6	
com.catstudio.littlesoldiers-11	Y
com.devuni.flashlight-139	Y
com.dreamstudio.bubblebear-6	Y
com.droidhen.fruit-46	
com.estrongs.android.pop-103	
com.fairytale.yule-2001	Y
com.fingersoft.hillclimb-27	
com.game.JewelsStar-21	
com.gamestar.pianoperfect-609	Y
com.halfbrick.fruitninjafree-1603	Y
com.hotdog.maze-11	Y
com.igs.mjstar31-19	Y
com.imangi.templerun-11	
com.jumplife.tvdrama-17	Y
com.jumplife.tvvariety-9	Y
com.kfactormedia.mycalendarmobile-74	
com.king.candycrushsaga-81	
com.missinggames.rescuerooby.lite-7	
com.mt.mtxx.mtxx-160	Y
com.jin.games.cleverblocks-2	
com.mufumbo.android.recipe.search-116	Y
com.mxtech.videoplayer.ad-46	
com.mybook.jpbookTY-1	
com.nextmediatw-70	Y
com.noodlecake.happyjump-4	Y
com.orangefish.app.delicacy-1069	Y
com.outfit7.talkingtom2free-21	Y
com.pipcamera.activity-5	
com.roidapp.photogrid-85	
com.rovio.angrybirds-3000	
com.shaun.emoticon-6	
com.softstar.Richman-4	Y
com.surpax.ledflashlight.panel-3	Y
com.ted.android-18	
com.vectorunit.yellow-7	Y
com.wooga.diamonddash-102142	
com.xidea.ChineseDarkChess2-44	
com.zdworks.android.toolbox-263	
idv.nightgospel.TWRailScheduleLookUp-108	Y
invoice.cof.tw-82	
jp.co.ponos.battlecats-8	Y
la.droid.qr-532	
opop.phind.twcoupon-16	Y
RailRush	Y
com.webprancer.google.garfieldCoins-7	

Untrusted network connections will be captured by the proposed approach. The detection results of the tested malware samples and the above two case studies conclude that the proposed approach could detect mobile malware correctly and efficiently.

Experiment 2: Experiment 2 is to evaluate if the proposed approach classifies benign applications correctly. Fifty popular free applications were collected from Google Play, ranging from games, reader tools, translators, and photographic tools. The results show that 24 mobile applications (48%) were considered as malicious. These applications leak device information, such as IMEI number for advertisement purposes after manual investigation. Some marketing companies count the number of collected device information to evaluate

TABLE 5. The results of Experiment 3.1.

Malware Name Antivirus Engine Name	Modified HippoSMS	Modified Asroot	Modified Sndapps
Proposed Approach	Sending premium SMS by String	Execute External Binary	Stealing device Id by java API
Agnitum			
AhnLab-V3			
AntiVir		EXP/Linux.Lotoor.A	
Antiy-AVL		Exploit/Linux.Lotoor	
Avast		ELF:Lootor-G[PUP]	
AVG		Android_c.IJM	
BitDefender		Android.Exploit.Asroot.A	
ByteHero			
CAT-QuickHeal		Exploit.Lotoor.C14	
ClamAV			
Commtouch			
Comodo			
DrWeb	Android.SmsSend.351.origi	Android.Gingersploit.3	
Emsisoft		Android.Exploit.Asroot.A (B)	
eSafe			
ESET-NOD32		Android/Exploit.Lotoor.AG	
F-Prot			
F-Secure		Exploit:Android/DroidRooter.E	
Fortinet		W32/Lotoor.U!exploit	
GData		Android.Exploit.Asroot.A	
Ikarus		Exploit.Linux.Lotoor	
Jiangmin		Exploit.Linux.t	
K7AntiVirus			
K7GW			
Kaspersky		Exploit.Linux.Lotoor.u	
Kingsoft			
Malwarebytes			
McAfee			
McAfee-GW-Edition			
Microsoft			
MicroWorld-eScan			
NANO-Antivirus		Exploit.Lotoor.bfneuw	

the effectiveness of a certain type of marketing campaign. Excluding the benign samples with such information leakage, the proposed approach identifies the benign correctly, as delineated in Table 4.

TABLE 5. (Continued.) The results of Experiment 3.1.

Norman			
nProtect			
Panda			
PCTools			
Rising			
Sophos		Andr/DroidRt-F	
SUPERAntiSpyware			
Symantec			
TheHacker			
TotalDefense			
TrendMicro		AndroidOS_ROOTCAGE.B	
TrendMicro-HouseCall		AndroidOS_ROOTCAGE.B	
VBA32			
VIPRE		Exploit.Linux.Generic.Elf	
ViRobot			

Experiment 3: Malware writers tend to make changes in malware in order to evade detection. To evaluate the detection performance on malware variants, Experiment 3 compares the proposed approach with the commercial anti-virus software, which consists of two sub-experiments: the first one evaluates the variants of the malware families from the dataset [39] and the second one evaluates new malware families which are not in the dataset.

In Experiment 3.1, the new variants were created from the following three malware families: Asroot, HippoSMS, and Sndapps. The modified misbehaviors are explained below.

- Modified Asroot: The modified Asroot performs the same behaviors as original malware as well as executes an embedded malware at a specific time of each month.
- Modified HippoSMS: The modified performs the same malicious behaviors as the original malware as well as sends SMS messages to premium-rate numbers one day after the malware is installed.
- Modified Sndapps: The modified Sndapps sends out device information to an external web site through HTTP GET when the user clicks a button 50 times.

The experimental results are summarized in Table 5. The proposed approach detects the variants efficiently, while some commercial software could not identify them. The proposed taint analysis approach can identify the misbehaviors regardless of the variations as long as they exhibit similar misbehaviors, while signature-based might not be able to identify the variants efficiently.

Experiment 3.2 evaluates if the proposed approach could identify new malware efficiently. New malware samples were collected including Trojan, adware, toolkit, and ransomware, as listed in Table 6. The misbehaviors of the

abovementioned malware include background execution without user approval, downloading malware, sending sensitive data to and receiving commands from command and control server, and launching attack according to the command received. The results show that the proposed detection method could identify the new malware efficiently as the proposed taint analysis can identify the misbehaviors by taint patterns, not by signatures.

TABLE 6. Malware evaluated by Experiment 3.2.

Malware family	Misbehaviors
Trojan: Android-PUP/Agent	download malware get root privilege pop up malicious URLs
AdWare.AndroidOS.PushAd	download and install malware uninstall software
RiskTool:Android/Dnotua	retrieve MAC address, ISP information, and device ID access external storage device such as SD card. execute commands
Ransom:Android/Agent	execute code after reboot

V. CONCLUSION

IoT devices have integrated with smartphones by using mobile applications, in which users access or control IoT devices remotely through their smartphones in the emergent network environment. Malicious mobile applications may compromise IoT devices, steal confidential information, and penetrate the network environment. Therefore, mobile application security should be ensured to protect the breaches.

This study proposes a power-efficient mobile malware detection method based on static taint analysis and hierarchical data model. The experimental results show that the proposed method can distinguish malware as well as benign applications effectively. It consumes fewer power and computing resources than dynamic analysis, so it is suitable for massive malware detection or anti-virus software installed on smartphones or IoT devices. The experimental results indicate that it can identify new malware and malware variants, while the commercial anti-virus cannot identify them effectively as the proposed solution is based on misbehavior patterns, not signatures.

In the future, the adaptation of IoT networks and 5G technologies would make mobile attacks even more prevailing. An automated threat pattern generation is needed to provide a better defense of the diversified attacks across multiple heterogeneous networks.

REFERENCES

- [1] R. Best. *Converged OT/IT Networks Introduce New Security Risks*. Accessed: Dec. 12, 2019. [Online]. Available: <https://www.infosecurity-magazine.com/opinions/ot-it-networks-risks/>
- [2] C.-S. Shih, J.-J. Chou, and K.-J. Lin, "WuKong: Secure run-time environment and data-driven IoT applications for smart cities and smart buildings," *J. Internet Services Inf. Secur.*, vol. 8, no. 2, pp. 1–17, May 2018.
- [3] *IDC Forecasts Worldwide Technology Spending on the Internet of Things to Reach \$1.2 Trillion in 2022*. Accessed: Jun. 29, 2019. [Online]. Available: <https://www.idc.com/getdoc.jsp?containerId=prUS43994118>
- [4] M. Park, J. Seo, J. Han, H. Oh, and K. Lee, "Situational awareness framework for threat intelligence measurement of Android malware," *JoWUA*, vol. 9, no. 3, pp. 25–38, 2018.
- [5] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity IoT," in *Proc. 27th Secur. Symp. (Security)*, 2018, pp. 1687–1704.
- [6] *The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History*. Accessed: Jun. 27, 2019. [Online]. Available: <https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/>
- [7] *500, 000 Belkin WeMo Users Could Be Hacked; CERT Issues Advisory*. Accessed: Jun. 23, 2019. [Online]. Available: <https://www.csoonline.com/article/2226371/500-000-belkin-wemo-users-could-be-hacked-cert-issues-advisory.html>
- [8] A. Arabo and B. Pranggono, "Mobile malware and smart device security: Trends, challenges and solutions," in *Proc. 19th Int. Conf. Control Syst. Comput. Sci.*, May 2013, pp. 526–531.
- [9] R. R. M. Chau. *Smartphone Market Share*. Accessed: Jun. 23, 2019. [Online]. Available: <https://www.idc.com/promo/smartphone-market-share/os>
- [10] McAfee Lab. *FakeInstaller' Leads the Attack on Android Phones*. Accessed: Jun. 23, 2019. [Online]. Available: <https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones>
- [11] D. G. Bilić. *Semi-Annual Balance of Mobile Security*. Accessed: Jun. 23, 2019. [Online]. Available: <https://www.welivesecurity.com/2018/08/29/semi-annual-balance-mobile-security/>
- [12] *McAfee Mobile Threat Report*. Accessed: Jun. 23, 2019. [Online]. Available: <https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf>
- [13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
- [14] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for Android malware detection," in *Proc. 7th Int. Conf. Comput. Intell. Secur.*, Dec. 2011, pp. 1011–1015.
- [15] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android application sandbox system for suspicious software detection," in *Proc. 5th Int. Conf. Malicious Unwanted Softw.*, Oct. 2010, pp. 55–62.
- [16] S. Iqbal and M. Zulkernine, "SpyDroid: A framework for employing multiple real-time malware detectors on Android," in *Proc. 13th Int. Conf. Malicious Unwanted Softw.*, 2018, pp. 1–8.
- [17] A. Shabtaï, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012.
- [18] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: A multi-level anomaly detector for Android malware," in *Proc. Int. Conf. Math. Methods, Models, Archit. Comput. Netw. Secur.* Cham, Switzerland: Springer, 2012, pp. 240–253.
- [19] G. Claudiu. *Obfuscapck*. Accessed: Jul. 25, 2020. [Online]. Available: <https://github.com/ClaudiuGeorgiu/Obfuscapck>
- [20] SRILAB. *Deguard*. Accessed: Jul. 25, 2020. [Online]. Available: <https://eth-sri.github.io/deguard>
- [21] Gyoonus. Accessed: Jul. 25, 2020. [Online]. Available: <https://github.com/Gyoonus/deoptfuscator>
- [22] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of malicious applications on Android OS," in *Proc. Int. Workshop Comput. Forensics*. Cham, Switzerland: Springer, 2010, pp. 138–149.
- [23] T. Matsudo, E. Kodama, J. Wang, and T. Takata, "A proposal of security advisory system at the time of the installation of applications on Android OS," in *Proc. 15th Int. Conf. Network-Based Inf. Syst.*, Sep. 2012, pp. 261–267.
- [24] S. B. Almin and M. Chatterjee, "A novel approach to detect Android malware," *Procedia Comput. Sci.*, vol. 45, pp. 407–417, Jan. 2015.
- [25] D. O. Sahin, O. E. Kural, S. Akleyilek, and E. Kiliç, "New results on permission based static analysis for Android malware," in *Proc. 6th Int. Symp. Digit. Forensic Secur. (ISDFS)*, Mar. 2018, pp. 1–4.
- [26] C. Urcuqui-López and A. N. Cadavid, "Framework for malware analysis in Android," *Sistemas Y Telemática*, vol. 14, no. 37, pp. 45–56, 2016.

- [27] S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users," in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2013, pp. 221–232.
- [28] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: Analysis of Android intent effectiveness in malware detection," *Comput. Secur.*, vol. 65, pp. 121–134, Mar. 2017.
- [29] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. NDSS*, vol. 14, 2012, p. 19.
- [30] A. Firdaus, N. B. Anuar, A. Karim, and M. F. A. Razak, "Discovering optimal features using static analysis and a genetic search based method for Android malware detection," *Frontiers Inf. Technol. Electron. Eng.*, vol. 19, no. 6, pp. 712–736, Jun. 2018.
- [31] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttki, "A new Android malware detection approach using Bayesian classification," in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Mar. 2013, pp. 121–128.
- [32] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.
- [33] N. Milosevic, A. Dehghantaha, and K.-K. R. Choo, "Machine learning aided malware classification of Android applications," *Comput. Electr. Eng.*, vol. 61, pp. 266–274, 2017.
- [34] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 317–331.
- [35] T. Lie and P. Ellingsen, "Integrating static taint analysis in an iterative software development life cycle," presented at the 3rd Int. Conf. Adv. Trends Softw. Eng., 2017.
- [36] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, Jun. 2014.
- [37] R. Hou, Z. Jin, and B. Wang, "Investigation of taint analysis for smartphone-implicit taint detection and privacy leakage detection," *EURASIP J. Wireless Commun. Netw.*, vol. 2016, no. 1, p. 227, Dec. 2016.
- [38] J. Lim, Y. Shin, S. Lee, K. Kim, and J. H. Yi, "Survey of dynamic anti-analysis schemes for mobile malware," *JoWUA*, vol. 9, no. 3, pp. 39–49, 2018.
- [39] K. Cao, J. He, W. Fan, W. Huang, L. Chen, and Y. Pan, "PHP vulnerability detection based on taint analysis," in *Proc. 6th Int. Conf. Rel. Infocom Technol. Optim. (Trends Future Directions) (ICRITO)*, Sep. 2017, pp. 436–439.
- [40] X.-X. Yan, Q.-X. Wang, and H.-T. Ma, "Path sensitive static analysis of taint-style vulnerabilities in PHP code," in *Proc. IEEE 17th Int. Conf. Commun. Technol. (ICCT)*, Oct. 2017, pp. 1382–1386.
- [41] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. Int. Conf. Trust Trustworthy Comput.* Cham, Switzerland: Springer, 2012, pp. 291–307.
- [42] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu, "DroidChecker: Analyzing Android applications for capability leak," in *Proc. 5th ACM Conf. Secur. Privacy Wireless Mobile Netw. (WISEC)*, 2012, pp. 125–136.
- [43] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. NDSS*, vol. 15, 2015, p. 110.
- [44] P. O. Fora. *Beginners Guide to Reverse Engineering Android Apps*. Accessed: Jun. 23, 2019. [Online]. Available: https://www.rsaconference.com/writable/presentations/file_upload/stu-w02b-beginners-guide-to-reverse-engineering-android-apps.pdf
- [45] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.
- [46] J. Palsberg, "Efficient inference of object types," *Inf. Comput.*, vol. 123, no. 2, pp. 198–209, Dec. 1995.
- [47] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.



CHIA-MEI CHEN has been with the Department of Information Management, National Sun Yat-sen University, since 1996. She was a Section Chief of the Network Division and the Deputy Director, Office of Library and Information Services, from 2009 to 2011. She has served as a Coordinator of Taiwan Computer Emergency Response Team/Coordination Center (TWCERT/CC), from 1998 to 2013, and then serves as a Consultant. Based on her expertise, she established Taiwan Academic Network Computer Emergency Response Team (TACERT), in 2009. She is the Deputy Chair of TWISC@NCKU, a branch of Taiwan Information Security Center. She continues working for the Network Security Society. Her current research interests include anomaly detection, malware analysis, network security, and cloud computing.



YI-HUNG LIU received the Ph.D. degree in information management from National Central University, Taiwan, in December 2014. Since 2020, he has been with the Department of Computer Science and Information Management, Soochow University, Taipei, Taiwan, where he is currently an Assistant Professor. He has been a Visiting Scholar with the Department of Management Information Systems, University of Arizona, USA, in 2013. His research interests include text mining, machine learning, e-commerce, social networks, and e-health.



ZHENG-XUN CAI received the master's degree from National Sun Yat-sen University, Kaohsiung, Taiwan, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include digital forensics, network analysis, and process analysis.



GU-HSIN LAI received the Ph.D. degree from National Sun Yat-sen University, Kaohsiung, Taiwan, in 2008. He is currently an Associate Professor with the Department of Technology Crime Investigation, Taiwan Police College, Taipei, Taiwan. His research interests include cyber security and cloud computing.

...