SoProtector: Safeguard Privacy for Native SO Files in Evolving Mobile IoT Applications

Guangquan Xu*, Member, IEEE, Weizhe Wang, Litaow Jiao, Xiaotong Li, Kaitai Liang*, Member, IEEE, Xi Zheng*, Member, IEEE, Wenjuan Lian, Hequn Xian, and Honghao Gao*, Member, IEEE

Abstract—Android Apps have become the most important mobile applications in the evolving mobile IoT systems, whose security and privacy are confronted with ever more challenges, since such mobile devices as smartphones involve too much personal privacy information. Meanwhile, the developers prefer to put core functions (e.g., encryption function and T9 search function) of Android applications in the native layer for execution efficiency. However, there are no automated security analysis tools to protect the security and privacy of the Android native layer, especially for those dynamically loaded third-party SO libraries. In order to solve the previous problem, which is confusing, we propose a novel and scalable system, called SoProtector, to prevent privacy from leaking via the analysis of data flow between the Java and native layers. For detection of the malicious function implanted in the SO libraries, SoProtector realizes a real-time engine. We derive the malware features via three steps: 1) present binary files in native family as a grayscale image; 2) with use of the ARM instructions set reversely obtain the code of the SO file and using Python to obtain the opcode sequence; and 3) each file is transformed as the form of assembly language by IDA Pro, which includes a gdl file as an accompaniment. Our experiment, which involved 3400 applications, demonstrates that SoProtector is able to detect more sinks, sources, and smudges. It effectively inspects and blocks at least 82% of the applications that are loading malicious third-party SO dynamically, and it has relatively low overhead in the meantime, compared to most of the existing static analysis tools (e.g., FlowDroid and AndroidLeaks).

Index Terms—Mobile privacy, mobile security, native C/C++ libraries, SO files.

Manuscript received June 26, 2019; revised September 3, 2019; accepted September 22, 2019. Date of publication September 30, 2019; date of current version April 14, 2020. This work was supported in part by the National Science Foundation of China under Grant 61572355 and Grant 1176115, and in part by the National Key Research and Development Program of China under Grant 2018YFB0804402. (Corresponding author: Weizhe Wang.)

G. Xu is with the Big Data School, Qingdao Huanghai University, Qingdao 266427, China, and also with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin 30035, China.

W. Wang and X. Li are with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin 30035, China (e-mail: 767436053@qq.com).

L. Jiao is with the Big Data School, Qingdao Huanghai University, China.

K. Liang is with the Department of Computer Science, University of Surrey, Guildford GU2 7XH, U.K.

X. Zheng is with the Department of Computing, Macquarie University, Sydney, NSW 2109, Australia.

W. Lian is with the College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China.

H. Xian is with the College of Computer Science and Technology, Qingdao University, Qingdao 266071, China.

H. Gao is with the Computing Center, Shanghai University, Shanghai 200444, China.

Digital Object Identifier 10.1109/JIOT.2019.2944006

© 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
native layer of the C/C++ programs that can generate SO libraries may not be detected via dynamic stain analysis. Lindorfer et al. [11] pointed out that the use of dynamic loading and reflection in malicious and nonmalicious Android applications has increased from 43.87% to 78% from 2010 to 2014, respectively. It was more challenging to detect the privacy leakage with current stain analysis tools due to the increasing amount of Android applications using dynamic loading techniques [12].

From the security point of view, Fischer et al. [13] measured over 1000-code segments posted on the stack overflow, which are insecure, that were copied and became parts of more than 1.3 million Android applications sold on Google Play. They showed that the insecure code (C/C++ and Java code) reuse rate and the increase of insecure code segments in the Android ecosystem are extremely high. Inspired by the work of Fischer et al., there are two binary functions: 1) an unknown function and 2) a malicious function, we might be able to detect if they are similar [14]. This is known as the “binary code similarity detection.” Eschweiler et al. [15] and Pewny et al. [16] used a graph matching algorithm to check if the control flow graphs of the given two functions are similar to each other to detect binary code similarity. Feng et al. [17] learned the control flow graphs’ advanced feature representations and embedded the graphs into the high-dimensional numerical vectors. Due to its super-linear runtime in the graph size, the matching algorithm of graph is usually inefficient. Recently, the deep learning algorithm [18] has been applied in analysis, e.g., compared with other approaches, binary analysis [19] has shown better analysis accuracy and performance. Xu et al. [20] proposed a deep neural network-based approach to generate embedded binary functions to launch the similarity detection. Paranthaman and Thuraisingham [21], Krupp et al. [22], and Saracino et al. [23] presented a method that applies the CNN algorithm to images converted from data of binary to extract important byte sequences in the malware samples. Nevertheless, the aforementioned works may not be deployed on the Android platform. There are only a few research papers that consider native library analysis, while most of the existing research focused on the Java bytecode or discussed methodologies to prevent Java native interface (JNI) attacks. References [24] and [25] analyzed 1.2 million Android applications and further generated a sandboxing policy that allows native code to function normally with the permission needed. Though the security policies could be used to block most of the exploits in malware, they are still unable to block the exploits using syscalls, which are commonly invoked by benign applications.

**Motivation:** Scanning applications and system core are the effective approaches to detect malware in Android platform. However, as introduced previously, the static stain analysis cannot distinguish the private data flow between the native layers and Java application, while all current dynamic analysis mechanisms have to “read” the data flowing through the Dalvik, which is not privacy preserving for the data. In addition, the third-party C/C++ library cannot generate .dex files, and thus TaintDroid could not effectively analyze the data flow within the native layer.

In this article, we ask: “Could we design a tool to detect the data flow between the Java application and the native layers without loss of data privacy and, meanwhile, the tool is able to analyze the security of the third-party SO files?”

To answer the question, we propose a malware detection system called SoProtector to scan the Android malware using the native library. SoProtector monitors the interaction between kernel-level and application-level, two levels of abstraction, to detect privacy leakage. We also design a real-time monitor in SoProtector to inspect the suspicious SO files. Our experiment involves in 3400 applications, which demonstrate that compared with most of the other tools of static analysis, SoProtector is effective and can detect more smudges, sources, and sinks; using the three features (code images, opcode sequences, and system calls), it successfully prevents over 82% of applications from dynamic loading SO files that are malicious.

**Contributions:** The contributions of this article are summarized as follows.

1) Based on FlowDroid, we propose an effective analysis tool to resolve an interesting open problem [7], i.e., how to track the interaction of data between the Java API framework and the native C/C++ libraries using the combined static and dynamic stain analysis. Through experiments, we show the effectiveness of our new tool. We state that the tool is part of the SoDetection.

2) We propose an automation tool for reversible SO files to detect if they are malicious by analyzing the combined characteristics of assembly code. The performance and effectiveness of the tool were verified by experiments. We design a method for nonreversible SO files to detect malicious variants by constructing the texture maps using the image processing and machine learning methods. This tool is a part of the SoPlatform.

3) We propose a real-time monitoring platform, called SoPlatform, to monitor the third-party SO files which are modifiable when applications including these files have been updated. This platform’s server is on a computer, but its client is on a mobile device where the Android system has been modified by us. The experiments present the performance of the platform.

4) We eventually create a dataset, including malicious binary SO files and Android source programs, for evaluation of malicious native program.

**II. PROBLEM STATEMENT**

**A. Background**

Some specific functions can be achieved by runtime loading dynamically some nonexistent executable files in the local machine. Through the JNI method, Android realizes some dynamic loading processes, such as loading calling functions and SO libraries (.SO libraries) that run in the native layer, which are compiled from C/C++ .SO files, are selected in the native Java codes to better the overall performance (such as T9 search, or Bitmap decoding [13]) of the whole system because of its good efficiency. It should be pointed out that SO library can be used in other similar scenarios, whose
reason is that the SO library is decompiled into assembly codes, e.g., a root exploit method is used by a new family of malware (Godless) to implant code in a native library [3]. Normally, all SO libraries are packed inside the applications, however, the external files can also load SO files.

B. Motivation

We use an example to demonstrate privacy leakage between the Java and native layers by dynamically loading SO files.

Line 5 (see Listing 1) shows that the libhello-jni.so library is dynamically loaded in the Java application layer; the IMEI number (a type of private data) of the device is fetched from lines 2 and 3, and passed to the native layer’s JNITransmit function; JNITransmit function (see Listing 3) is written by C that locates in the native layer: line 2 obtains the TestClass class through the reflection mechanism; lines 6 and 8 obtain the static and nonstatic functions’ id numbers, and lines 7 and 9 call the nonstatic function sendFaker1 and the static function sendFaker2, respectively. It can be noticed that the device’s IMEI number and the encapsulated character string “11111” shown in Listing 1 are used as the parameters which are passed to the above functions. Listing 2 shows the core code of the TestClass class: the nonstatic function sendFaker1 and the static function sendFaker2 send the content message to the mobile device with the phone number. In combination with the parameters passed in, the caller sends a text message of the IMEI number to the device whose phone number is 11111 through the JNI mechanism and the dynamic loading technology.

Most of the tools fail to distinguish the private data flow between the native layers and the Java application, so that the code above can cause privacy leakage without being detected. Therefore, it is desirable to design a tool to monitor data flow between the layers.

C. Difficulties

1) Binary Files of the Third-Party SO Libraries Which Are of Binary-Safe Field: We are not able to capture their assembly codes if we make use of security measures like shelling.

2) There Is a Little Comprehensive Research Work: Little research has been done for preventing JNI attacks that consider the native library analysis, and there are also few tools which can be used to extract features from the native libraries and model binary files’ behaviors. We, therefore, analyze the interaction of data between the Java and native layers manually, which leads to low analysis efficiency.

3) It Is Not Necessary to Package Third-Party SO Libraries in Applications Directly: The private directory, which is executable, loads the needed SO libraries while a program is running. Static analysis analyzes the flows of data involving the layer of native ineffectively because those files are not in the application. More importantly, at any time after the program runs, the SO files can be updated without reinstalling the application. After installing the security check, it is extremely hard to detect that the malicious application uses malicious SO files to replace the benign one.

III. SYSTEM OVERVIEW

Fig. 1 shows the SoProtector’s overall architecture. Before proceeding, we need to define the following terms.

1) Source File: Denoted by Sw. The file of C/C++ embeds the source method.
2) **Source Method**: Denoted by $S_f$. As shown in Listing 1, the method which called from the native layer (e.g., `JNI Transmit` method).

3) **JNI Interaction Method**: Denoted by $J_h$. The caller invokes the method to execute the mechanism of reflection (e.g., `GetStaticMethodID` and `GetMethodID` methods).

4) **The Target Method**: Denoted by $T_f$. As shown in Listing 3, the function of the native layer, which calls the method of the Java layer.

5) **Target Class**: Denoted by $T_c$. The method of defining the target class.

SoPlatform and SoDetection make up SoProtector. SoDetection is composed of the static analysis and the dynamic execution module. For simplicity, we use SoDetection-x to denote the dynamic module and SoDetection-y for the static module.

**Information Extraction**: After the modified and recompiled source codes of Android application are installed to the Android device, SoDetection runs the SoDetection-x first, which can record the involved information, such as functions of dynamic loading that appear in the log output, and install the tested application onto the device of modified Android. We will explain the proposed changes to the source code of the Android system in Section IV. To get the application’s reflection invocation and dynamic loading information, SoDetection-x executes the application and reads the system log output after installing the application. SoDetection-x extracts the method of JNI interaction named $J_h$, the source method named $S_f$, and $T_f$, which is the method of target matching the call of reflection from the log, and use the form of a triplet $<S_f, J_h, T_f>$ to store the information in the file named SJT repository of computer locally when catching the application’s calling behavior of reflection. Similarly, to download the .dex and SO files to the computer locally, SoDetection-x sends the adb command of download to the device of Android after capturing the dynamic loading behavior of the application. The SO files downloaded are taken as the input for SoPlatform and SJT repository and the .dex files are further used for subsequent analysis of static.

**Data Analysis**: SoDetection executes SoDetection-y which is FlowDroid’s improved variant after the dynamic analysis module is terminated. We inject the SJT information library and .dex files into the process of static stain analysis of SoDetection-y. The necessary files of .dex and JavaClass of the application loaded by SoDetection-y first, and then translates them into Jimple of Soot [18], which is the language of three address. SoDetection-y transforms between the methods of source and reflection to construct the right call graph of functions according to the repository of SJT.

**SoPlatform Detect the Malicious File**: SoPlatform calculates the SO file’s hash value and stores it first. The system call, the code image of malicious binary, and the assembly opcode $n$-gram are used as features. By a lot of test sets, we set a threshold to judge if the SO files are malicious. We further extract the native family of malicious to which it belongs.
functions are not in the Java layer, but there are two ways to capture the functions’ displaying and calling.

1) We consider that the use of four tools, such as Addr2line tools, GCC, Ptrace (open source), and Dot (Fig. 3) for the SO file that is reversible to the code of C and ARM. Using GCC detection functions to label the native functions (Tf), which generate the files of trace named trace.txt after transforming the function’s address into the function name by using Addr2line tools, we use the dot to get the function calling graph and can transform the map into the native layer data-flow diagram added in the FlowDroid diagram. We may find Jh and Sf that Tf matches by analyzing the calling relationship.

2) Compared with the SO files that are reversible, we use IDA Pro to trigger the native function against the SO files which are difficult to reverse, even irreversible. The paragraph information of SO files can be extracted from the process table, and its code segment with execute permission is shown as a series of the DCB data. From the header of the DCB data, we can extract the Jh (JNI interaction method) and Sf (the corresponding source method) that matches Sf.

C. Stage 3: Processing in SoDetection-y

SoDetection-y, which is a FlowDroid’s improved variant, injects SJT information library and .dex files (Fig. 4) to the process of analysis, so that it can handle the dynamically loaded SO files correctly. Our analysis method is based on Jimple language of Soot. It loads all the classes, first into memory, then it builds the call graph of function and generates the main method. SoDetection-y only accepts classes corresponding to the loaded .dex file to reduce the cost of memory storage. SoDetection-y adds the call graph of function generated by the disposal of SO files to the main graph automatically when the function call graph builds.

In short, SoDetection-y handles the reflection methods of source files (in the Java application and native layers) to create a complete control flow graph with two layers of data interaction. Fig. 4 shows the processing SJT library’s algorithm flowchart. For the mapping of SJT, SoDetection-y
confirms whether the method is the invoke method or the getxxxid method first. It further gets the target parameter Tp converted by the target reflection parameter Sp. If the object of target reflection is empty and confirmed by SoDetection-y, it will be defined and assigned. As shown in Fig. 4, the getxxxid method mainly determines the initialization parameters from the source data. The track of taint between the Java API framework and native C/C++ libraries is shown in Fig. 5. We state that our tool can distinguish the private data flow between the layers of native and Java application.

D. Stage 4: Processing in SoPlatform

SoPlatform is designed as a server terminal remotely, and it is used to detect the SO files which are further stored in the server. The server stores and calculates its hash value if an SO file is uploaded first. Using the system call, the assembly opcode n-gram, and the code image of malicious binary as the features. We use the ML algorithms, such as the random forest and the decision tree with the DNN classifier for classification.

It is difficult to extract the malicious behavior characteristics of the malicious code, especially for the binary code. But we can use heuristics to scan with binary code which is unknown.

Heuristics is a method of static detection, but the efficiency of running the binary file is actually not high (experimental part). Note that SO files will map various information of segment to memory (including the segment of code and data, which are shown as a number of SO subparagraphs on the process table), thus we have to detect the segment of code with segment of SO which is executive.

Then, we describe in detail the selected three main features. Feature 1 (The Malicious Binary Code Image): Fischer et al. [13] pointed out that the segments of insecure code (C/C++ and Java code) are widely spread in the Android system, and hence the reuse rate becomes high. And the virus feature library of the native family is not huge in terms of the size. Due to these reasons, we can present binary files in native family as a grayscale image, and make the texture features as an input of a machine learning algorithm. The values of all bytes fall into the interval of $[00 \sim FF]$, whose grayscales are $[0 \sim 255]$. Converting a binary file into a matrix is the benefit of transforming into a grayscale, whose process is as follows.

1) Hexlify function is used in converting a binary file into a hexadecimal string.
2) After segment by byte, we generate rectangle as per the set width through adopting the reshape function.
3) Transform the above rectangle into an image by using fromarray function.

The reason for selecting the features lies in the fact that the malicious codes belong to the same family, which is represented as a similar texture. By the GIST technology, we use a perception vector with five dimensions to represent the image. Once such vectors are gotten, the classification of mobile apps can be completed with the help of machine learning algorithms.

Feature 2 (Opcode Sequence Frequency of Appearance): Opcode sequence frequency of appearance is used in the area of language-based security [14]–[16]. With the use of the set of ARM instruction to obtain the SO file’s code reversely...
Fig. 6. Sequence of API Call in the RageAgainstTheCage binary file found in DroidKungfu.

Listing 4. Example code in hello-jni.c which makes the libhello-jni.SO.

```c
1:    jclass class = (*env)->FindClass(env, "com/zhangning/nidemo/TestClass");
```

Listing 5. ARM code converted by the C code in hello-jni.c.

```c
.text:00000FF8  PUSH  {R4-R7,LR}  
.text:00000FFA  ADD   R7, SP, #0xC  
.text:00000FFC  SUB   SP, SP, #0xC  
.text:00000FFE  PUSH  {R0}  
.text:00001000  POP   {R4}  
.text:00001002  LDR  R0, [R4]  
.text:00001004  LDR  R2, [R0,#0x18]  
.text:00001006  LDR  R1, -(aComZhangning: nidemo - 0x100C)  
.text:00001008  ADD   R1, PC
```

Listing 6. Getting opcode query from the ARM code by Python.

```python
1:    def getOpcodeNgram(ops, n=3):
2:        oplist = [tuple(ops[i:i+n]) for i in range(len(ops)-n)]
3:        ngram = Counter(oplist)
4:        return ngram
```

Fig. 7. Sketch map about ARM opcode n-gram for Listing 6.

For instance, if we extract the 3-gram in Listing 7, we can get the 1-gram, 2-gram, and 3-gram in Fig. 7. Fig. 6 shows that this call’s sequence allows us to model the malicious behavior of the malware effectively and provide the system with a feature to apply classification algorithm. According to the length of the subsequence as n(1, 2, 3) to process the sequence, next, we calculate the TF result of each opcode sequence. We obtain the vector \( S = (\Delta_1, \Delta_2, \ldots, \Delta_n) \) which includes the frequency of opcode sequence. We combine the above values into the weight vector \( W = (\text{wtf}_1, \text{wtf}_2, \ldots, \text{wtf}_m) \). We also calculate the SO files that are malicious to be tested (vector \( V_1 \)) and the other \( m \) various kinds of malicious samples (vectors \( V_2 \cdots V_{m+1} \)) separately.

We also combine Features 1 and 2 (see Listing 8) to increase the test accuracy.

**Feature 3 (Sequences of System Calls):** Using system calls (syscall) as a feature for machine learning is a possibility as most malware use a series of syscall to exploit
The assembly language and generate a gdl file containing the
using a disassembler tool IDA Pro. Disassemble each file into
guage. In the third stage, each file is being disassembled
or files that contain native code, such as assembly or C lan-
stage, each application folder is searched for the native library
reverse-engineered back to the source code. In the second
files of SoPlatform. In the first stage, the APK files are being
in Table I. Fig. 9 shows the overview for analyzing the SO
count the sink number and the taint path number as shown
9) Geinimi; 10) ADRD; and 11) BeanBot.
2) DroidKungFu2; 3) DroidKungFu3; 4) DroidKungFu4;
the native families of malware include: 1) DroidKungFu1;
for invoking mechanism, it
is one of the most important features of Java. Malicious appli-
cations put core malicious functions in the native layer. These
core functions are invoked by the Java layer. For this purpose,
the proportion of nonmalicious applications with invoke mech-
anism is higher than malicious applications. Also, we can see
that malicious and nonmalicious applications are often used
with GetXxxID or GetStaticXxxID functions for interactions
from Table II.
The disposal phase for every application takes 2 s in our
experiment and it includes the preprocessing time for applica-
tions. Dataset-based application static analysis is processed
in 23 threads concurrently. We think such time expense to be
quite acceptable since security analysts or customized system
vendors are SoProtector, which mainly targets.

V. EMPIRICAL STUDY
A. Empirical Settings
Dataset: We crawl 3000 applications covering 15 cate-
gories which preclassified from Wandoujia Store, and from
VirusShare we crawl 400 applications with the native layer
which from 2013 to 2017, Contagio Mobile [22] (Contagio
database’s 13 malware families) and reference [21], Genome
(applications are divided into 49 families of malware).
We download the top 200 applications from each of the
categories which preclassified from Wandoujia Store, and from
VirusShare we crawl 400 applications with the native layer
which from 2013 to 2017, Contagio Mobile [22] (Contagio
database’s 13 malware families) and reference [21], Genome
(applications are divided into 49 families of malware).
We download the top 200 applications from each of the
categories which preclassified from Wandoujia Store, and from
VirusShare we crawl 400 applications with the native layer
which from 2013 to 2017, Contagio Mobile [22] (Contagio
database’s 13 malware families) and reference [21], Genome
(applications are divided into 49 families of malware).

B. Comparison With FlowDroid
The following types of the private data that can be leaked are
represented by the numbers from 1 to 10 in column “Content
of Privacy” in Table I: call records, geolocation information,
message records, contacts, mobile phone identification, Baidu
accounts, Wi-Fi information, Bluetooth information, base sta-
tion information, and browser information. The mark in “Ways
of Privacy Leakage” column indicates that the private data can
be leaked in the following ways, from 1 to 4: network, short
message, log, and file. The apps on top 14 are benign and oth-
ers are malware. It can be seen that compared with FlowDroid,
SoDetection can detect more smudges, sinks, and sources,
since it can handle the reflection mechanism and dynamic
loading effectively. Because nonmalicious applications could
not load the SO files of third-party, we can see that it could
not use the dynamic loading mechanism from Table II, mean-
while, the applications of malicious with code of malicious in
the SO files completely adopt the mechanism of dynamic load.
Therefore, all of the malicious applications we tested with
dynamic loading mechanism. As for invoking mechanism, it
is one of the most important features of Java. Malicious appli-
cations put core malicious functions in the native layer. These
core functions are invoked by the Java layer. For this purpose,
the proportion of nonmalicious applications with invoke mech-
anism is higher than malicious applications. Also, we can see
that malicious and nonmalicious applications are often used
with GetXxxID or GetStaticXxxID functions for interactions
from Table II.

C. Discussion
We compare our tool with the following state-of-the-art.

1) Comparison With TaintDroid: In terms of accuracy,
both FlowDroid and SoProtector cannot be compared to
TaintDroid [7]. However, SoProtector outperforms TaintDroid
in terms of program coverage and can spot tainted paths that
TaintDroid cannot. Although both tools rely on the actual
operation of the apps to handle the Android dynamic loading
and reflection mechanisms, SoProtector just needs to get
the app dynamically loaded .dex files and reflection calling information. Besides, it only monitors Android’s dynamic loading and reflection calling behavior to make a more comprehensive stain analysis of apps. But TaintDroid requires the application in the dynamic process to run the complete path from the source to sink. In addition, TaintDroid can only alert users when a privacy breach occurs, rather than outputting a full path from a source of pollution to a point of entry, and in comparison, SoProtector can output a full path from a source of contamination to a point of entry.

2) Comparison With AppFence: Much like TaintDroid, AppFence [8] accurately detects leaks of private data that occur in the Android systems. However, it has the same drawbacks as TaintDroid in terms of code coverage and false negatives. AppFence may hinder the normal operation of the application. The normal operation of the application is like to send some private information (e.g., IMEI) to the external network. However, AppFence’s data masking strategy may not allow the application to send data out or carry out other normal operation of the applications.

<table>
<thead>
<tr>
<th>APK ID</th>
<th>FlowDroid</th>
<th>Taint Propagation Path Number</th>
<th>Sink Number</th>
<th>Taint Propagation Path Number</th>
<th>SoDetection</th>
<th>Content of Privacy</th>
<th>Ways of Privacy Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(1,3)</td>
<td>{1,2,4}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>12</td>
<td>74</td>
<td>19</td>
<td>[6,10]</td>
<td>{1,1}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>44</td>
<td>21</td>
<td>44</td>
<td>26</td>
<td>[1,2,1,2]</td>
<td>{1,3,1,3,4}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>86</td>
<td>0</td>
<td>86</td>
<td>0</td>
<td>[7,10]</td>
<td>{1,1}</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>5</td>
<td>17</td>
<td>7</td>
<td>[3,6,3,6]</td>
<td>{1,3,1,3}</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>29</td>
<td>14</td>
<td>30</td>
<td>15</td>
<td>[2,3,2,3,4,5]</td>
<td>{1,2,1,2}</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>3</td>
<td>30</td>
<td>7</td>
<td>[3,5,3,5]</td>
<td>{1,2}</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>5</td>
<td>16</td>
<td>6</td>
<td>[4,4,4,9]</td>
<td>{1,1,2,3}</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>4</td>
<td>17</td>
<td>9</td>
<td>[1,1,1,7]</td>
<td>{1,3,1,3,4}</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>12</td>
<td>25</td>
<td>17</td>
<td>[2,2,2,8]</td>
<td>{1,1}</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>31</td>
<td>14</td>
<td>35</td>
<td>19</td>
<td>[3,3,3,6]</td>
<td>{1,2,4,1,2,4}</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>26</td>
<td>16</td>
<td>31</td>
<td>18</td>
<td>[4,6,4,6]</td>
<td>{1,1}</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>39</td>
<td>23</td>
<td>42</td>
<td>25</td>
<td>[3,4,3,4,5]</td>
<td>{1,4,1,4}</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>72</td>
<td>41</td>
<td>74</td>
<td>41</td>
<td>[1,2,1,2]</td>
<td>{1,2,3,1,2,3,4}</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>(5,10)</td>
<td>{1}</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>7</td>
<td>17</td>
<td>9</td>
<td>[2,8,2,8]</td>
<td>{1,2,1,2}</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>9</td>
<td>23</td>
<td>13</td>
<td>[3,9,3,9,10]</td>
<td>{1,3,1,3,4}</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>11</td>
<td>19</td>
<td>12</td>
<td>[6,9,2,6,9]</td>
<td>{1,1,2}</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>13</td>
<td>26</td>
<td>14</td>
<td>[7,3,7,10]</td>
<td>{1,1,4,4}</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>34</td>
<td>13</td>
<td>36</td>
<td>15</td>
<td>(1,3)</td>
<td>{1,2,1,2,4}</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>29</td>
<td>21</td>
<td>31</td>
<td>23</td>
<td>[6,9,6,9]</td>
<td>{1,3,1,3,4}</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>41</td>
<td>32</td>
<td>45</td>
<td>35</td>
<td>(8,10)</td>
<td>{1,1,1}</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>43</td>
<td>20</td>
<td>47</td>
<td>23</td>
<td>[1,3,1,2,4,5]</td>
<td>{3,3}</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>32</td>
<td>17</td>
<td>36</td>
<td>19</td>
<td>(2,3)</td>
<td>{2,3,2,3}</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>33</td>
<td>21</td>
<td>37</td>
<td>24</td>
<td>[2,1,6]</td>
<td>{1,3,1,3,4}</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>16</td>
<td>29</td>
<td>18</td>
<td>[4,5,4,5]</td>
<td>{1,2,4,1,2,3,4}</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>23</td>
<td>12</td>
<td>25</td>
<td>15</td>
<td>[3,4,3,4,5]</td>
<td>{1,3}</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>21</td>
<td>11</td>
<td>27</td>
<td>14</td>
<td>[1,1,6]</td>
<td>{1,2}</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>19</td>
<td>8</td>
<td>32</td>
<td>11</td>
<td>[3,3,9]</td>
<td>{2,2,4}</td>
<td></td>
</tr>
</tbody>
</table>
Thereof, it may not monitor privacy leaks of the application effectively.

3) Comparison With DroidLegacy: In DroidLegacy [26], Deshotels applies an approach that scans the Android code for API calls and uses this feature to classify the malware. DroidLegacy stresses the importance of API calls which are the core functionality of the app file. The DroidLegacy system generates a signature by partitioning the app into modules to create a class dependency graph (CDG). Each node in the graph represents a class, and each edge shows the relationship between the classes. The graph will group the identical classes together to form a single module, then these modules will be checked for similarity using the Jaccard similarity to the known malware and the selected shared features, such as API calls frequency, are used to generate the malware family signature. Malware is detected and classified based on each malware family signature. We also used the API calls as an important feature to classify, and monitored some important API calls. However, we considered both API calls by reflection in the native layers and OpCode sequences which make our tools better than DroidLegacy.

4) Comparison With DroidNative: In DroidNative [27], Alam et al. applied a static control flow graph analysis to detect and classify malware. The system can process the Java bytecode and native libraries that are included in applications that provide an important vision on the possible methods exploited by malware. An application is determined to be a malware if it matches a high percentage of its call flow graph to a known malware sample. The system uses two techniques: 1) annotated control flowgraph (ACFG) and 2) sliding window of difference (SWOD), which are used on the generated signatures from the call flow graph to determine the similarity of the graph to those generated by the malware training dataset. However, the obfuscation techniques can diminish the performance of the detector.

Our tool presents a binary file as the image of grayscale. We combine two techniques, which are ACFG and SWOD, against the obfuscation technique so as to improve its validity (see Fig. 10). We also use cases of DroidBench to compare these tools in Table III. We can see SoProtector can detect more test cases.

D. Quantitative Comparison With Different Features

We compute the recall and precision of SoProtector based on the sum of FNs, FPs, and TPs (53, 375, 2972) as follows:

\[
\text{Recall} = \frac{TP}{TP + FN}, \quad \text{Precision} = \frac{TP}{TP + FP}.
\]

Fig. 10. Combination of two techniques.

In general, most of the applications could be precisely identified by SoProtector with 98.25% recall and 88.79% precision.

The malware DroidKongFu, which is popular in recent years, is an important class of tested malware, and its core code is in the native layer. SoProtector successfully identifies it in the dataset as malicious. SoProtector could detect the sending of SMS message, which is misbehavior. DuanXinGongJiQi is the SMS Trojan which is famous for avoiding most anti-virus in China, but it was detected as an SMS Trojan by SoProtector. SoProtector is more precise than malware whose core functions are in the native layer. These results show that it is an effective alternative to static stain analysis methods.

In Section IV-D, each result is not identical because of the randomness in the process of forest training. But, in general, the accuracy is much higher and the basic accuracy is not less than 72% with the combination of the two methods. Fig. 11 describes the precision and recall rate of dealing with related data by .ASM file image features, Opcode 3-gram features, and the combination of the two features. In case of a small sample size, we use Python’s numpy, pandas, PIL, scikit-learn (convenient to use random forest algorithm) library to train subsets (60% training set and 40% test set), and feature extraction; a random forest of 500 decision trees is constructed. Combined with Features 1 and 2, the test set’s accuracy will increase by 30% in all.

About Feature 3 in Section IV-D: we mainly test its accuracy of classifying malware and f1_score to its malware native families (see datasets). We also calculate the true positive rate (TPR) and the false positive rate (FPR) by machine learning algorithms. The results are listed in Table IV. We can see

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>With GetStaticXxxID mechanism</th>
<th>With GetXxxID mechanism</th>
<th>With invoke mechanism</th>
<th>With dynamic loading mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>App number</td>
<td>Proportion</td>
<td>App number</td>
<td>Proportion</td>
</tr>
<tr>
<td>Malicious applications</td>
<td>400</td>
<td>218</td>
<td>0.545</td>
<td>203</td>
<td>0.5075</td>
</tr>
<tr>
<td>Non-malicious applications</td>
<td>3000</td>
<td>1713</td>
<td>0.571</td>
<td>1612</td>
<td>0.5373</td>
</tr>
</tbody>
</table>
TABLE III
TEST RESULTS BY DROID BENCH

<table>
<thead>
<tr>
<th></th>
<th>TaintDroid</th>
<th>SoProtector</th>
<th>DroidLegacy</th>
<th>DroidNative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays and Lists</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayAccess1</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ListAccess1</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Callbacks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnonymousClass1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Button1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MethodOverride1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Field and Object Sensitivity</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IntentSink1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ActivityCommunication1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BroadcastReceiverLifecycle1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ActivityLifecycle1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>serviceLifecycle1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lifecycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SourceCodeSpecific1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>StaticInitialization1</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Miscellaneous Android-Specific</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrivateDataLeak1</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DirectLeak1</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Fig. 11. Precision and recall rate of dealing with related data.

that the DNN classifier is used to classify malware, which results in higher accuracy and f1_score. The reason might be that by stratification, more complex functions in the native layer can be represented with fewer parameters.

VI. RELATED WORK

Feature engineering is now used in many ways, Zhang et al. [34] used it to detect the anomaly of human and Xie et al. [35] used feature engineering in driving condition recognition. At the same time, more research is being done in Android malware detection by feature engineering. As an essential element of evolving mobile IoT applications, Android applications’ security also get more attention, and more novel protocols for mobile IoT are proposed [36]–[38].

Now, Android malware detection often uses stain analysis, and on the other hand, static and dynamic analyses both have advantages and disadvantages. A lot of researchers are working on it.

Android privacy risk raises much attention. Liu et al. [39] noticed privacy risk in libraries in the Android ecosystem and proposed analysis and mitigation. In smart devices or Android-based IoT, Wang et al. [40]–[42] proposed some ways to detect Android malware and got a great result.

Dynamic loading is used in the Android applications. However, this mechanism may yield security risk in applications. Wang et al. [28] studied the vulnerabilities that may be caused by loading execution code and proposed the problem that malicious applications may bypass the security check with the use of the dynamic loading mechanism. To prevent this risk, Zhauniarovich et al. [29] proposed the tool named StaDynA that can download the .dex files which are dynamically loaded when applications are running. It can combine with some stain analysis tools [7]–[10] to generate function call graphs. However, StaDynA is a preliminary tool that takes up a lot of memory.

In the static stain analysis, Yang and Yang [30] proposed the tool named LeakMiner, which takes the life cycle of Android applications into consideration and constructed a function call graph that can cover all the callback methods. But LeakMiner cannot analyze the implicit information flow. Gibler et al. [10] proposed a tool named AndroidLeaks which...
solves the above question, but it cannot distinguish the features between the domain and the object. That is, if one attribute of an object stored the stain data, AndroidLeaks would regard that the whole object as the stain; so its accuracy is not high. Arzt et al. [9] proposed FlowDroid, which is more advanced stain analysis tool. FlowDroid can build an accurate lifecycle model and handle the callback function effectively. It uses the interprocedural finite distributive subset (IFDS) algorithm for efficiency and distinguishes different features by making marks; however, it cannot analyze the data flow among components. In order to solve this issue, Eschweiler et al. [15] proposed the tool named Epicc, which is built on the basis of FlowDroid. Furthermore, in order to enable FlowDroid and Epicc to analyze the data flow among applications, Pewny et al. [16] proposed DidFail which is built on the basis of these tools. In a word, the static tools above have high code coverage but low accuracy to track the taint.

In the dynamic stain analysis, TaintDroid [7] marks the privacy data as a stain, which requires more than twice the space of memory for each object in the system to store the marks of the stain when the application is running. Hornyack et al. [8] proposed AppFence, which can not only detect the privacy leakage in the system but also to prevent the privacy leakage. It takes data hiding technology to replace the privacy data with the shadow data so as to prevent privacy leakage; but its strict security measures reduce the system performance. Schreckling et al. [31] proposed Kynoid, an improvement of the TaintDroid, which can detect more sources of stain. But Kynoid needs more memory than TaintDroid. In a word, the dynamic tools above have high accuracy to track but low code coverage.

Unfortunately, there are only a few published works that consider native library analysis. Most of the existing research works deal with the Java bytecode or for preventing JNI attacks. Afonso et al. [24] analyzed 1.2 million Android applications and generated a sandboxing policy that allows native code to function normally with the permission needed. In the first step, the system applied a filtering technique to identify the applications that use native code. In the next step, the system dynamically analyzes the actions performed by the application, such as system calls or JNI calls. Based on the information gathered, a series of security policies using the least privilege principle were created on System calls (syscall) and Java methods. Though the security policies generated can block most exploits used by malware, it is unable to block exploits that use syscalls, which are commonly invoked by benign applications. An example of such exploits will be the DroidKungfu’s RageAgainstTheCage and Zimperlimch, which uses the fork syscall in its exploit.

In a nutshell, for native library analysis, syscall is often used as a feature for machine learning as most malware use a series of syscalls to exploit vulnerabilities in the system. This view is supported by [32] and [33], which used syscall in the Java bytecode as a feature and fed it into a deep learning network for native library analysis, which can also be seen from Fig. 10 by our experiment. However, a detection system designed for malware using native code is needed to provide a more robust defense against native code attack.

VII. CONCLUSION

Considering the strong demand of more and more diversity and execution efficiency of IoT applications, Android App developers prefer to put Android applications’ core functions in the native layer. To solve the security and privacy problem still faced by the evolving mobile IoT applications, in this article, we proposed SoProtector, a real-time engine as the malware detection system for Android malware interacting with the native library, which is the earlier privacy-safeguarding attempt from the layer of native library. It can detect more sources, sinks and smudges than most static analysis tools and most applications that dynamically load malicious third-party SO files with low overhead. However, there are some limitations in this article. On the one hand, when analyzing malware, we find more and more applications that are protected by the native code added to the shell. We have not introduced how to solve the native code as the addition protection in malicious applications, which will be considered in future work. On the other hand, SoProtector has to deploy relevant servers and may yield the transmission cost of network, which may affect its efficiency in the setup stage to a degree.

REFERENCES


Guangqu Xu (M’15) received the Ph.D. degree from Tianjin University, Tianjin, China, in March 2008. He is a Full Professor with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University. He is also a part-time Professor with Qingdao Huanghai University, Qingdao, China. His current research interests include cyber security and trust management. Prof. Xu is a Member of CCF.

Weizhe Wang received the bachelor’s degree from the Changchun University of Science and Technology, Changchun, China, in July 2018. He is currently pursuing the master’s degree with the College of Intelligence and Computing, Tianjin University, Tianjin, China. His current research interest includes cyber security.

Litaof Jiao received the M.B.A. degree from the Shandong University of Science and Technology, Qingdao, China, in 2016. He is currently an Associate Professor with Qingdao Huanghai University, Qingdao, China. He was awarded the prize of Provincial Educational Achievement in 2018, participated in five major provincial and municipal research projects, and published over ten papers. His current research interests include HR management and information security.
Xiaotong Li received the B.S. degree from the School of Mechanical, Electrical and Information Engineering, Shandong University of China, Jinan, China, in 2018. She is currently pursuing the master’s degree with the College of Intelligence and Computing, Tianjin University, Tianjin, China. Her current research interest includes Web application protection techniques.

Kaitai Liang (M’15) received the Ph.D. degree from the Department of Computer Science, City University of Hong Kong, Hong Kong, in 2014. He is currently an Assistant Professor with the Department of Computer Science, University of Surrey, Guildford, U.K. His current research interests include applied cryptography and information security in particular, encryption, blockchain, post-quantum crypto, privacy enhancing technology, and security in cloud computing.

Xi Zheng (M’16) received the bachelor’s degree in computer information systems from Fudan University, Shanghai, China, the master’s degree in computer and information science from the University of New South Wales, Sydney, NSW, Australia, and the Ph.D. degree in software engineering from the University of Texas at Austin, Austin, TX, USA. He is currently an Assistant Professor/Lecturer of software engineering with Macquarie University, Sydney, NSW, Australia.

Hequn Xian received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2009. He was a Visiting Scholar with the College of Information Science and Technology, Pennsylvania State University, State College, PA, USA. His current research interests include cryptography, cloud computing security, and network security.

Honghao Gao (M’11) received the Ph.D. degree in computer science from the Shanghai University, Shanghai, China, in 2012. He is currently a Distinguished Professor with the Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Shanghai University. His current research interests include service computing, model checking-based software verification, and sensors data application. Dr. Gao is a fellow of IET, BCS, and EAI. He is a Senior Member of CCF and CAAI.