TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 04/13 pp589–603
DOI: 10.26599/TST.2019.9010067
Volume 25, Number 5, October 2020

A Novel Hybrid Method to Analyze Security Vulnerabilities in Android Applications

Junwei Tang, Ruixuan Li*, Kaipeng Wang, Xiwu Gu, and Zhiyong Xu

Abstract: We propose a novel hybrid method to analyze the security vulnerabilities in Android applications. Our method combines static analysis, which consists of metadata and data flow analyses with dynamic analysis, which includes dynamic executable scripts and application program interface hooks. Our hybrid method can effectively analyze nine major categories of important security vulnerabilities in Android applications. We design dynamic executable scripts that record and perform manual operations to customize the execution path of the target application. Our dynamic executable scripts can replace most manual operations, simplify the analysis process, and further verify the corresponding security vulnerabilities. We successfully statically analyze 5547 malwares in Drebin and 10 151 real-world applications. The average analysis time of each application in Drebin is 4.52 s, whereas it reaches 92.02 s for real-word applications. Our system can detect all the labeled vulnerabilities among 56 labeled applications. Further dynamic verification shows that our static analysis accuracy approximates 95% for real-world applications. Experiments show that our dynamic analysis can effectively detect the vulnerability named input unverified, which is difficult to be detected by other methods. In addition, our dynamic analysis can be extended to detect more types of vulnerabilities.

Key words: Android security; vulnerability analysis; static analysis; dynamic analysis

1 Introduction

Android devices have accounted for more than 80% of the world's smartphone market in recent years. The number of Android apps is also reaching millions. Various types of applications substantially enrich our daily life but could also bring new security concerns regarding sensitive information. Given that mobile phones contain substantial users' private information, private data may be leaked by malicious applications or applications with security vulnerabilities.

The analysis of application vulnerability is normally applied to detect the security vulnerabilities incurred by poor development or other reasons. These vulnerabilities may be used by attackers to achieve denial of service attacks, privilege promotion, application data, and user privacy disclosure, resulting in huge losses to ordinary users. Compared with traditional desktop applications, Android applications carry more private information of individual users and are more easily targeted by attackers. With such a large number of Android devices and applications, how to accurately detect various types of security vulnerabilities in Android applications remains a valuable research topic.

Several existing approaches can analyze certain categories of vulnerabilities in detail, but cannot be applied to other types of vulnerabilities. Certain approaches can only detect industry-specific application

• Junwei Tang, Ruixuan Li, Kaipeng Wang, and Xiwu Gu are with School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: jwtang@hust.edu.cn; rxli@hust.edu.cn; kpwang@hust.edu.cn; guxiwu@hust.edu.cn.
• Zhiyong Xu is with Math and Computer Science Department, Suffolk University, Boston, MA 02101, USA and also with Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Shenzhen 518055, China. E-mail: zxu@suffolk.edu.
* To whom correspondence should be addressed.
Manuscript received: 2019-10-30; accepted: 2019-11-04

vulnerabilities, whereas others are based on a specific kind of single static code analysis, which may feature low accuracy and high false positives. Currently, several dynamic analysis methods have been proposed to detect Android application's vulnerabilities based on fuzzy test and sandbox, which also exhibit their own disadvantages. Specific dynamic analysis methods are limited to a small category of vulnerabilities[1] and cannot be extended to the detection of more common vulnerability categories. Other methods require operating system modification[2], their configuration environment is very complex and the results are inconvenient to analyze. In addition, the system overhead would be notably large.

To solve the above problems, we propose a novel hybrid approach to conveniently and accurately detect important types of security vulnerabilities in Android applications. We take advantage of static and dynamic analyses. We combine metadata analysis with data flow analysis during our static analysis process. Our dynamic detection engine is based on automated executable scripts and Application Program Interface (API) hooks. Our dynamic executable scripts can improve the efficiency of dynamic analysis and increase the types of detected vulnerabilities. In addition, these dynamic executable scripts can customize the execution path of the target application instead of the fuzzy test and complete the runtime state analysis of the target application in accordance with its real-time response in collaboration with API hooks.

On the basis of the defined vulnerability judgment rules, the above analysis processes generate vulnerability analysis results for the target application. Then, we can systematically analyze the important security risks of Android applications.

The main contributions of this paper are as follows:

(1) We propose a novel hybrid method, which combines static and dynamic analysis approaches, to analyze security vulnerabilities in Android applications. The static analysis consists of metadata and data flow analyses, whereas the dynamic analysis is based on dynamic executable scripts and API hooks.

(2) The dynamic executable scripts can avoid most manual operations, simplify the vulnerability detection process, and further verify the existence of corresponding security vulnerabilities. The script generation module generates a corresponding executable script based on manual operations. After updating the basic script, it is convenient to verify and detect the vulnerabilities.

(3) We perform successful static analysis on 15 698 applications, including 5547 malwares in Drebin[3] and 10 151 real-world applications. The average analysis time of each application in Drebin is 4.52 s, whereas that of each real-world application is 92.02 s. In addition, we show the relationship between application vulnerabilities and size.

(4) We implement a system using static and dynamic analysis approaches. Among the nine types of vulnerabilities analyzed by our system, eight are detected by static analysis, and one is handled by dynamic analysis. Our system can correctly detect all the labeled vulnerabilities in 56 test applications, and further dynamic verification shows that the accuracy of our static analysis approximates 95% for real-world applications. Our dynamic analysis can detect the vulnerability named input unverified, which is hard to be analyzed by static analysis. The types of the detected vulnerabilities can be enriched and extended in the future.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 introduces our system design and implementation. Section 4 presents the evaluation. Section 5 discusses the limitations of the study. Section 6 introduces the related work. Section 7 concludes the whole paper.

2 Background

2.1 Security vulnerabilities in Android applications

In Android platform, the situation of application security vulnerability is as extremely serious as that of the traditional software. Security vulnerabilities in Android applications could lead to serious security consequences as smartphones carry substantial sensitive user information. Several features of security vulnerabilities in Android applications are similar to those of traditional desktop applications, but they show more difference because of the new features of Android applications themselves. For example, an Android application may incorrectly or not use an encryption function at all to directly transmit and process sensitive information in plaintext. This condition can easily lead to privacy leaks. In addition, if the components of an Android application are exposed to other applications without proper permission restrictions, a privilege attack caused by unrestricted component vulnerability may occur. Several of these vulnerabilities are related to the unique structure of Android applications. Other vulnerabilities

are due to the incorrect usage of APIs.

2.2 Instrumentation and Android test project

The Android test project is based on the instrumentation framework, where the test application can precisely control the target application. Using instrumentation, we can create instrumentation of system objects such as context before the target application is started and control multiple life cycles of components within the target application. We can also send User Interface (UI) events to the application, check application status information during execution, and so on. The instrumentation framework enables this functionality by allowing target and test programs to run in the same process, similar to the hook mechanisms. We can use this mechanism to dynamically analyze target applications.

3 System Design and Implementation

Our system consists of three main processes, including static code, dynamic, and result analyses. The metadata analysis in static analysis decompiles the executable file of an Android application to obtain basic metadata information. Then, we can preliminarily obtain permissions, components, and other security-related information of the application in a lightweight analysis process. Further, we analyze the transmission path of the data of interest within the application by data flow analysis. To overcome the false alarm and omission of static analysis, further verify the results of static analysis, and detect security vulnerabilities triggered only by actual operation, we propose a dynamic analysis method based on the executable script and API hooks. The result analysis process needs to aggregate the static and dynamic analysis results, generating the final security vulnerability analysis results. Our hybrid method takes advantage of both static and dynamic analyses. Compared with running these two methods independently, our method is more accurate and comprehensively detects security vulnerabilities in Android applications. Figure 1 shows the architecture of detection system.

3.1 Vulnerability judgment rules

3.1.1 Vulnerability pattern

We use formal language to describe the corresponding vulnerability pattern instead of natural language. First, a few definitions are given below. The security vulnerability patterns are formally described as follows:

$$V = (v_1, v_2, v_3, v_4) \tag{1}$$

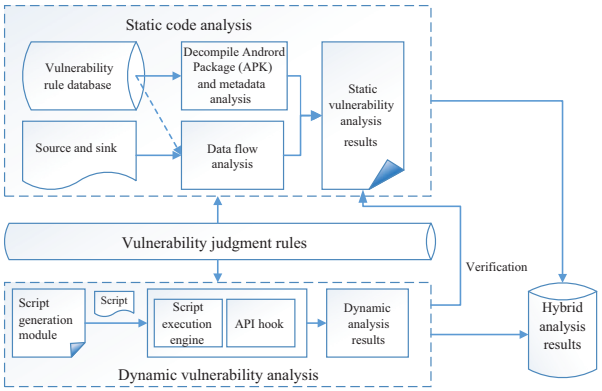


Fig. 1 Architecture of detection system.

where V represents the security vulnerability in Android applications, v_1 denotes the security vulnerability type, v_2 refers to the number of security vulnerability of this type in the application, v_3 indicates specific vulnerability information or vulnerability evidence, and v_4 corresponds to the specific method for detecting this vulnerability.

Definition 1. Vulnerability based on metadata analysis means that these types of vulnerability information are detected by static metadata analysis. The main techniques include conditional judgment and string regular matching.

For example, an application might possess the security vulnerability of insecure JavaScript in WebView. We express the vulnerability as $V =$ (insecure JavaScript in WebView, 1, the addjavascriptInterface function is called in the application and the unsafe JavaScript function may lead to JavaScript injection attack, metadata analysis).

Definition 2. Vulnerability based on data flow analysis means that these types of vulnerability information are detected by static data flow analysis.

Definition 2.1. Source refers to the acquisition point of sensitive data, which is mainly the API function for acquiring sensitive data (such as getInstalledPackages). Sink is the data transmission point, which is mainly a function for data storage, transmission, and other operations.

Definition 2.2. A data transmission path usually refers to a program code execution path connecting the source and sink points; it can be an edge of the data flow graph considered in the study.

Definition 2.3. The vulnerability path represents the propagation path of sensitive data in the application, including the data acquisition point and the data leakage point. It can be expressed as a triple ⟨source, sink, data transmission path⟩.

For example, we perform a static analysis on the application and discover that the application may possess the security vulnerability of loading code from the outside using the class loader technique. We express the vulnerability as $V =$ (dynamically loading file, 1, a path p about dynamically loading file exists, data flow analysis).

Definition 3. Vulnerability based on dynamic analysis means that these types of vulnerability information are detected by dynamic executable script and API hook techniques.

3.1.2 Matching rules

Our vulnerability rule is a set of description specifications based on safety experience and defined in the form of the above formalized expressions. Table 1 shows the nine important security vulnerabilities that are considered in the present study.

Table 1 Vulnerability pattern.

Detection process (v_4)	Type of vulnerability detected (v_1)	Vulnerability description (v_3)
Metadata analysis	Unrestricted component	The component properties in the application are improperly set, causing the component to be invoked illegally.
	Insecure JavaScript in WebView	The unsafe JavaScript function used in the application may lead to JavaScript injection attack.
Data flow analysis	Sensitive data processed in plaintext	There is a path consisting of defined sources and sinks about the sensitive data.
	Privacy leak by log	The logcat interface is called to output sensitive data as logs.
	Dynamically loading file	There is a path consisting of defined sources and sinks about dynamically loading file during runtime.
	Insecure password	Use a simple reversible encoding function to encrypt or hard-code the secret keys into the code.
	Intent exposure	Other applications construct intents that contain malicious content to launch components exposed by the vulnerable application.
	Structured Query Language (SQL) inject	Use SQLite without proper type detection and validation of the parameters passed in the SQL statement.
Dynamic analysis	Input unverified	Input for the login or registration UI of the application is not validated legally.

The vulnerability named unrestricted component refers to the unreasonable setting of the component properties in Android applications; it may cause the component to be illegally called. The Android: exported attribute in its Androidmanifest.xml file is simply set to be true with no valid permissions to restrict the other applications' components from calling it. The insecure JavaScript in WebView vulnerability means that applications with WebView have the ability to communicate between JavaScript code and native code through event response function and other methods. If unsecure JavaScript functions are used, it may lead to JavaScript injection attack and expose the sensitive resources obtained by the application to unknown web content. The sensitive data processed in plaintext refers to the situation that applications process sensitive information in unencrypted plaintext in various ways. In addition, we separately list the situations in which the application calls the logcat interface and outputs sensitive data as logs as the privacy leak by log vulnerability. The vulnerability named dynamically loading file means that the application invokes various techniques to load files from outside itself during running. For example, many applications use dynamic code loading to load external code or Android Package (APK) files through methods such as DexClassLoader. This has been proven that it may lead to malicious code execution. At the same time, if the application uses shell commands or techniques such as Native Development Kit (NDK) and libraries written in C/C++, we will consider that this application may also feature such type of vulnerabilities. Security vulnerabilities can also occur if an insecure password is used in an application. These situations can be that the application uses a simple reversible encoding function to encrypt, or simply hard-code the secret keys into the code without any protections, and so on. Intent exposure refers to the fact that other applications can start an application and make it perform dangerous operations by constructing intents containing malicious content using the components exposed by the target applications. If the IntentFilter information of the application is not configured properly, it is easy for other applications to exploit the vulnerability to attack. SQL inject vulnerabilities are also possible in Android applications. The application may use SQLite (a lightweight database on the Android platform) without proper type detection and validation of the parameters passed in the SQL statement. This condition can lead to collisions and

database leaks. Another type of vulnerability involves user input. If the information entered by the user is incorrectly validated by the application, it may cause serious consequences, such as program logic failure and application sensitive information leakage. We focus on the input to the login and registration UI of the application here. Validation of input information on other UIs can be extended in the future.

3.2 Static analysis

3.2.1 Metadata analysis

Metadata analysis mainly uses decompiling technology to obtain the metadata information of Android applications. Android applications are usually developed in Java language, compiled into dex format files, and run in Android virtual machine instances. However, the packing technology causes considerable difficulty to directly reverse the analysis.

First, we extract and analyze the AndroidManifest.xml file. This file contains metadata, such as application permission request, component construction, and main activity. In the subsequent data flow analysis, we map the different elements to the corresponding Java objects based on these metadata, and complete the analysis and detection of the vulnerabilities in accordance with the vulnerability judgment rules.

The binary code (APK file) is decompiled into a folder containing a bytecode file named dex.class. We decompile the code into smali format code, which can be further decompiled into Java code. However, this Java code will miss several program semantic information and may increase the probability of decompilation failure. Thus, we analyze the permissions, components, and other related security vulnerabilities of Android applications directly from the smali code. Through this intermediate bytecode form, we can directly analyze whether unsafe functions are called in the application through string regular matching. The comprehensive use of Static Android Analysis Framework (SAAF)[4], AXMLPrinter2[5], and baksmali[6] can effectively obtain highly readable and semantically complete bytecode files. The combination of various decompiler tools can effectively improve the decompiler's success rate.

3.2.2 Data flow analysis

The core of the static data flow analysis is to track the information of the marked tainted data, analyze its possible propagation paths, and then perform the corresponding processing. The main principle is to mark the data of interest as "tainted". Thus, a series of new data is generated through operations, such as arithmetic and logic operations, and will inherit the "tainted or not" property of the source data. In this manner, operations, such as jump, call, and assignment of tainted data, are analyzed to build the corresponding propagation path of tainted data.

Soot[7] is the classic Java code data flow analysis tool. FlowDroid[8] expands Soot to the information flow analysis of Android applications. FlowDroid transforms Android applications that are based on component life cycle and callback function running mechanism into a Java program based on a dummy main function. Based on the prepared source and sink, sensitive data flow analysis can be performed for the entire Android application. Whether a path in the application starts with the defined source point and ends with the sink point must be determined. The existence of such a path can indicate that the application features the corresponding security vulnerabilities. This path is called the vulnerability path.

We extend the ideas of detecting such sensitive data leakage, and enrich the range of source and sink points. We have defined several functions that are unsafe for our own application scenarios as source and sink points. This data flow analysis can determine whether vulnerability paths caused by the use of sequences of the corresponding functions exist. Based on the expansion of the above tools, we mark the sensitive information and use static taint analysis technology to obtain the possible propagation path of the taint information to determine whether a vulnerability path exists in the application.

3.3 Dynamic vulnerability analysis

3.3.1 Dynamic executable script

(1) Script generation

We implement an automated test method to drive the target application to execute a customized test path. Two kinds of approaches are used for Android automated testing. One is based on the UIAutomator, and can only simulate and trigger corresponding events to operate on the target application but cannot directly obtain the attributes of each element of the target application to operate. The other kind of approach is based on the instrumentation framework, in which the process being tested and the target application are run as two threads in the same process. Without triggering any events, the individual elements can be accessed internally and the corresponding data can be modified. We select

the second method and extend our system based on Robotium[9], which is an open source automated testing framework using instrumentation. Cafe[10] is based on Robotium with more features and supports real-time recording of user actions in target applications. We perform a secondary encapsulation on the Cafe to further simplify its complex operations and generate custom executable scripts.

We can create rich test cases with instances of the solo object in Robotium. The core difficulty of the dynamic vulnerability analysis is how to reduce the manual operation but achieve a more intelligent and automated generation of executable scripts and drive the target application to run the corresponding program logic. By extending the Cafe framework, we have completed the automatic generation of manual operation script and accurately recorded the manual operations. In this way, we not only avoid the tediousness and mistakes of writing scripts manually but also expand our analysis system on the basis of such scripts. The basic script can be modified based on different vulnerability analysis requirements to complete a more efficient dynamic analysis. The main idea of the automatic recording and generating script is to continuously listen to various operations (including clicking, sliding, and input, etc.) on the view of the target application, and then automatically generate corresponding codes based on the API representation provided by Robotium according to the types of each operation and widget. The view represents an area of the interface that the current user inspects, and it can contain multiple widgets that can interact with the user. For cases where the widget is not recognized, we can use coordinates-based click operation instead. The scripts based on coordinate may not be accurate enough, thus causing failure of operations.

We design new executable scripts to be expressed in the XML format. As shown in Fig. 2, a script mainly contains the main activity and package name information of the target application and the test case information based on the individual operation events of Robotium. These operational events can be clicks, inputs, swipes, waits, screenshots, and so on. The tag TestCases can contain multiple TestCase tags. The attributes in a TestCase tag contain the name of the TestCase, whereas the contents of the tag contain various action events, which are generated automatically by our script generation module. We can extend and modify these generated basic scripts in accordance with the corresponding requirements of security vulnerability analysis. For example, these changes could include the addition of clicks, swiping actions, modification of the input, inclusion of a delay interval of two steps, looping actions, and so on.

```
<TestSuite>
  <targetInfo>
    <actname name="mainActivity "/>
    <pacname name="targetPackage "/>
  </targetInfo>
 <TestCases>
  <TestCase name="Case0">
   Click event
   Input event
   Swiping event
   Wait
   …
  </TestCase>
  <TestCase name="Case1">
   …
  </TestCase>
   …
 </TestCases>
</TestSuite>
```

Fig. 2 Dynamic script in XML format.

(2) Script execution

The process of script execution enables the target application to reproduce the various operations written in our scripts. The main principles are instrumentation and the Android test project (Android developer documentation now recommends AndroidJUnitRunner instead of the Android test project). The test application connects the target applications with the instrumentation tags in its manifest.xml file. The targetPackage attribute determines the package name of the target application. The test and target applications interact through InstrumentationTestRunner and run in the same process. In general, applications with different signatures run in their own independent Dalvik process. Thus, using the same signature to re-sign the test application, the target application is necessary.

In general, the life cycle of components is controlled by the Android system. Functions, such as onCreate, onResume, and onStop in the activity life cycle, are all called by the system itself, and the Android application framework provides no APIs with the permissions for users to call directly. However, by using instrumentation, we can create instrumentation of system objects, such as context, before the target application is formally started. We can control the life cycle of the components in the target application, generate and send simulated user and

system events to the target application, and monitor the running states at the same time.

Our script execution module automatically generates the corresponding Android test project based on instrumentation and extracts the corresponding content of the script. Then, the corresponding Java code of the Android test project is generated. After automatically compiling the project with the command line compilation tool, we generate the test APK file. After installing both the test and target APK files and passing the information of the main activity of the target application when launching the test application, the script drives the target application to perform the correspondiung operations. Figure 3 shows the principle of dynamic executable scripts.

3.3.2 Hook and analyze related methods

To document important information about security vulnerabilities at runtime, we use the Xposed[11] framework to hook related important methods to record information about these vulnerabilities. For example, we hook the findClass method in BaseDexClassLoader and record the call of this function, and then verify the existence of the vulnerability in accordance with our vulnerability patterns. Through our dynamic script, the target application is automatically driven, and several running state information is recorded during the execution process to comprehensively verify the existence of the vulnerability. The hook APIs, call records, and other information to be recorded can be defined based on a specific vulnerability pattern. These information can be easily expanded on the basis of the requirements of verification vulnerabilities.

3.4 Result analysis

The results of our vulnerability detection system based on the hybrid analysis approach are obtained by synthesizing the information of the above analysis process. Among the analysis approaches, the static analysis will give a considerable part of the vulnerability analysis results. Meanwhile, the dynamic analysis plays two roles: It not only performs further dynamic validation of partial results of static analysis but also analyzes several types of vulnerabilities that are hard to be detected directly by static analysis.

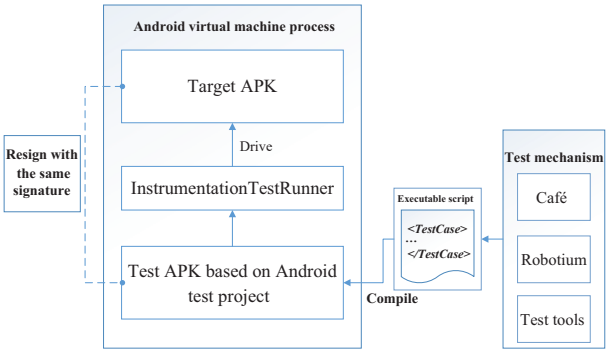


Fig. 3 Principle of dynamic executable scripts.

Static analysis cannot detect certain types of vulnerabilities, and it faces problems such as failure to verify the existence of vulnerabilities. However, the dynamic analysis is not only expensive but also difficult to detect certain types of security vulnerabilities. Our hybrid method can avoid these shortcomings. Avoidance of such issues is the difference between our hybrid method and running two independent analyses. The whole process makes our hybrid approach more comprehensive and effective than individual analysis methods.

4 Evaluation

4.1 Dataset and success rate of decompilation

Table 2 shows the overall situation of the metadata and data flow analyses of our detection system. As long as the decompilation is successful, we can perform the metadata and data flow analyses normally. The results show that the success rate of static analysis of our detection system reaches 99.65% on average.

4.2 Vulnerability detection

To evaluate the vulnerability detection effect of our system, we have designed several experiments to test the static and dynamic analyses of the whole system, and point out two application scenarios of our dynamic executable scripts, including further verification of results of static analysis and detection of vulnerabilities that are hard to be detected by static analysis alone. Finally, by combining with all the results, we give the overall analysis situation of our system.

Table 2 Dataset and percentage of successful static analysis.

Dataset type	Total number	Number of successful decompilations	Success rate (%)
Drebin	5560	5547	99.97
Real-world app (Google Play and App store in China)	10 194	10 151	99.58
Developed test apps	56	56	100
Average	15 810	15 754	99.65

4.2.1 Static analysis results of our system

In this section, we will separately analyze our static vulnerability detection results on the malicious application dataset Drebin and our collection of real-world application datasets. Our static analysis can detect eight types of vulnerabilities.

(1) Static analysis results on Drebin

Among 5560 malicious applications on the Drebin dataset, our system successfully analyzes 5547 malwares. Through metadata and data flow analyses, we separately obtain the number of malicious applications under each category of vulnerability. Figure 4 shows the distribution of these numbers. The statistical results show that most malicious applications have potential vulnerabilities named unrestricted components. This condition also proves to a certain extent that malicious applications considerably reuse codes and call components to each other. Meanwhile, about a third (31%) of these malicious apps have vulnerabilities named dynamically loading file. This finding also shows that many malicious applications may avoid static detection by separating parts of the codes. The codes with specific sensitive features and malicious behaviors are loaded dynamically at runtime. Applications that use functions to execute JavaScript code to access local system resources in the WebView accounts for 27.93% of the total applications. Numerous studies have proven that calling this interface in WebView to execute JavaScript code can cause serious security risks.

Figure 5 shows the percentage of applications with different numbers of vulnerabilities in Drebin. The findings show the presence of 4690 applications with potential vulnerabilities in 5547 Drebin datasets, accounting for 84.55%. A total of 43.95% of applications

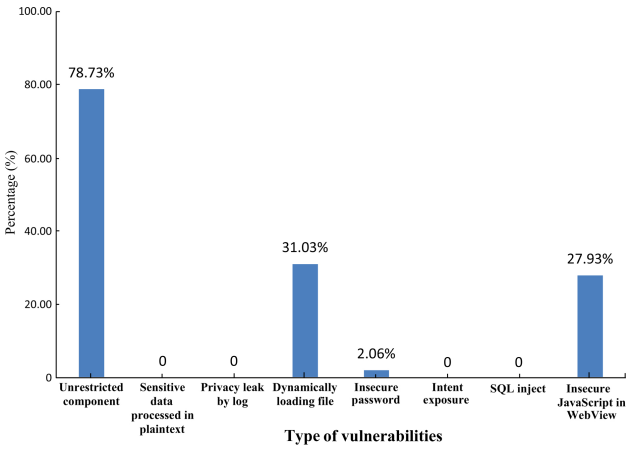


Fig. 4 Percentage of apps detected for each vulnerability in Drebin.

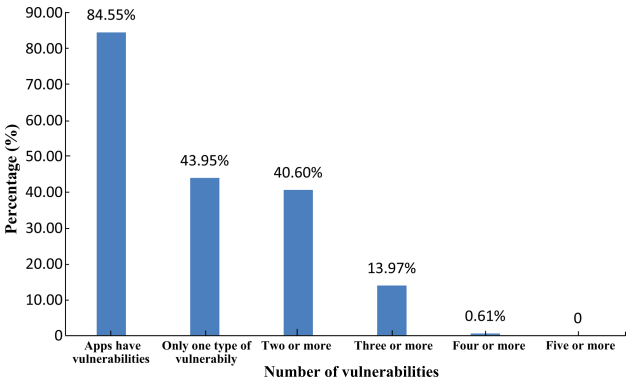


Fig. 5 Percentage of apps with different numbers of vulnerabilities in Drebin.

contain one type of vulnerability; 40.60%, two types of vulnerabilities or more; 13.97%, three types of vulnerabilities or more; 0.61%, four types of vulnerabilities or more. No application includes five or more than five types of vulnerabilities. The analysis of the 5547 malwares lasts for 2064 s, averaging 4.52 s per application.

(2) Static analysis results on real-world apps

In total, we have collected 10 194 real-world applications from various functional categories by the end of 2017. Most of these applications originate from Google Play and others from mainstream application markets of China. Table 3 shows the size and number distribution of these real-world applications. The average size of all applications for analysis is 11.15 MB.

The eight types of vulnerabilities are detected by our static analysis. We successfully analyze 10 151 real-world applications, and all the results are shown in Fig. 6. Dynamically loading file accounts for the largest proportion at 57.60%. More than half of all applications contain another vulnerability, that is, the unrestricted component. Given that Android applications are component-based, the interaction between components can complete complex business logic. Numerous potential vulnerabilities exist in the components of Android apps. Notably, about half of the applications have potential security vulnerabilities

Table 3 Dataset and percentage of successful static analysis on real-world applications.

Application file size (MB)	Total number of application	Number of successful analysis applications
0–1	2503	2496
1–10	2514	2497
10–20	3437	3419
>20	1740	1739

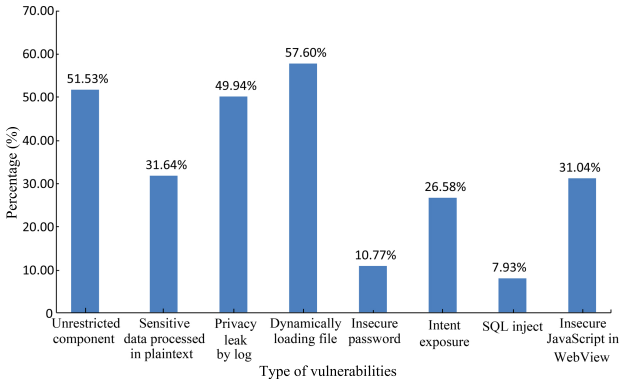


Fig. 6 Percentage of real-world apps detected for each vulnerability.

that leak private data through log messages. Developers might leave behind the logs for the convenience of debugging during the development. In addition, these remaining logs can be obtained by an attacker, which results in a privacy leakage. The applications with the vulnerability named insecure JavaScript in WebView, which executes JavaScript code through the API of the WebView component, also accounts for about one-third of these applications. This condition also shows that as more applications start to use WebView, the proportion of hybrid applications will increase. This kind of vulnerability of sensitive data processed in plaintext is a situation that requires attention. This vulnerability accounts for one-third of the total and causes serious user privacy leaks. Developers need to pay more attention to the security of sensitive data transmission in their applications and avoid direct plaintext data processing.

Figure 7 illustrates the average time of these 10 151 applications, that is, the average consumption time of static analysis for applications of different sizes. The

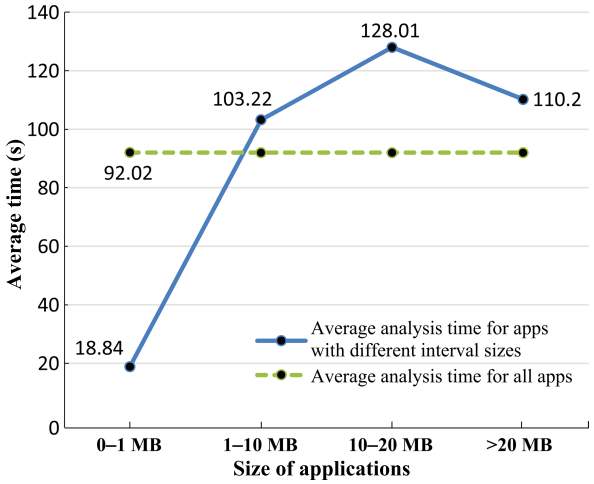


Fig. 7 Average time for static analysis of applications of different sizes.

average time of static analysis for these applications can increase gradually with the increase in their sizes. However, after reaching a certain size, the static analysis time will not increase dramatically. After exceeding 20 MB, the average analysis time of the application decreases compared with the previous 10–20 MB applications, because most of these applications are probably just a bit over 20 MB in size. At the same time, the portion of the increase in file size may be resource files, which we disregard in our static analysis, rather than program codes. Thus, this condition causes no increase in the final static analysis time.

The average analysis time for 0–1 MB applications is about 19 s. Beyond 1 MB, the average analysis time quickly rises to over 100 s. In general, the larger the application, the greater the amount of code is. Therefore, the more complex metadata analysis and data flow analysis required, the longer it will take. The average static analysis time for all applications is 92.02 s.

From Fig. 8, we can statistically analyze the potential security vulnerabilities in applications of different sizes. For applications of different sizes, the eight types of vulnerabilities exhibit different distributions. For smaller applications, the highest percentage of potential vulnerabilities is that of the unrestricted component type. As for larger applications, the main security vulnerability is the dynamically loading file. The main reason for this result is that smaller applications may perform various functions that need to interact with the outside world. Thus, component interfaces may need to be open to others. However, larger applications may need additional code to execute, and their business logic is more complex. Figure 8 also shows that with the increase in application size, the percentage of various potential vulnerabilities in applications gradually increases in most cases.

As shown in Fig. 9, on average, 83.46% of applications have potential security vulnerabilities. More than 60% of applications contain two or more types of vulnerabilities, and applications with one type of vulnerability account for 19.31%. More information can be obtained by classifying these applications based on their sizes. For applications smaller than 1 MB, the percentage of vulnerable applications (61.90%) is considerably lower than the average percentage of all applications. The percentage of applications larger than 1 MB containing security vulnerabilities is notably higher (more than 87%). For applications with sizes less than 1 MB, the proportion containing one type of

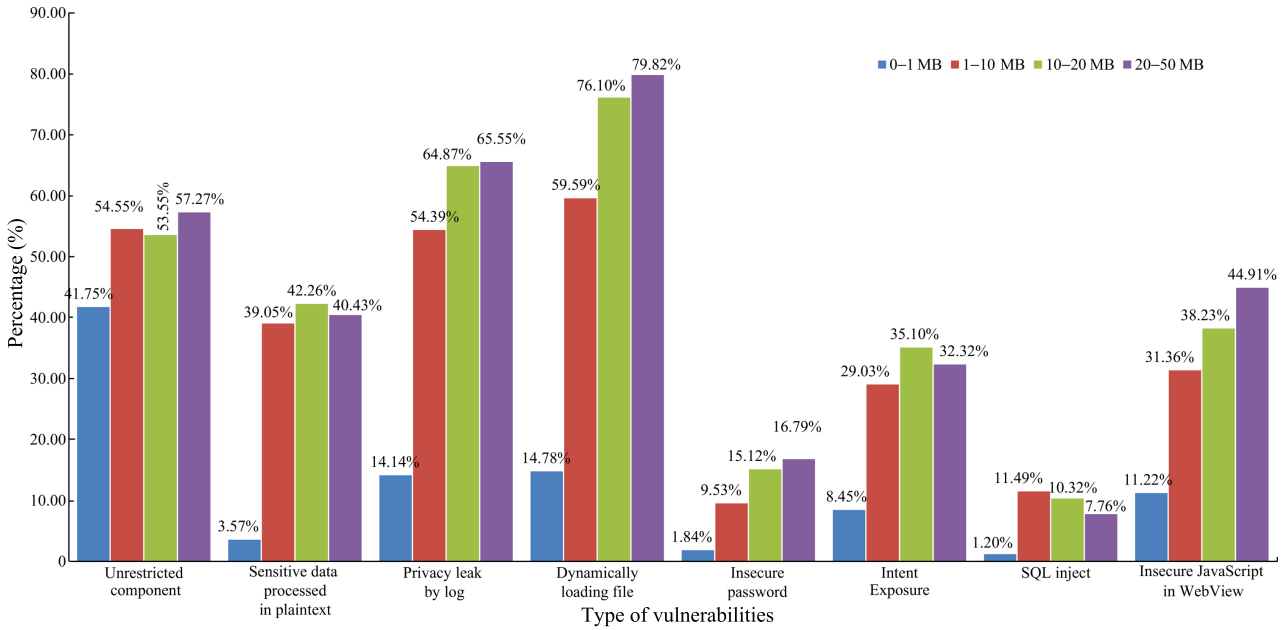


Fig. 8 Vulnerability distribution for each category of applications of different sizes.

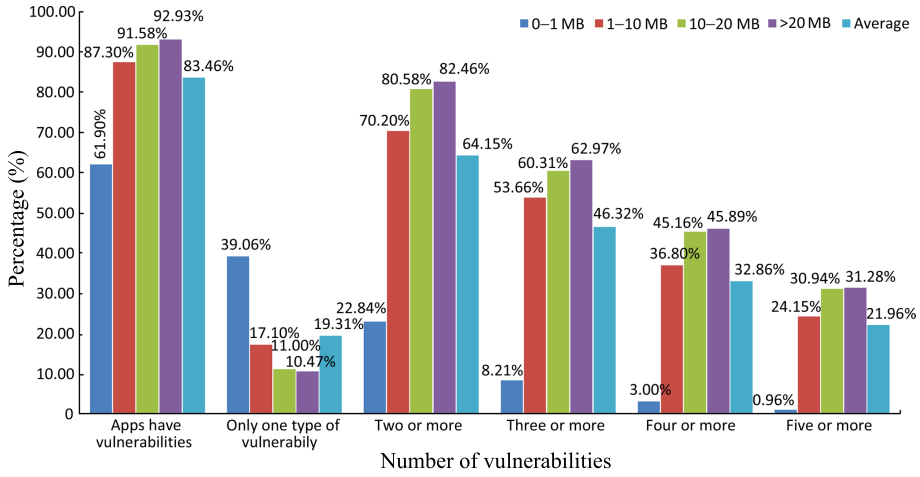


Fig. 9 Distribution of number of vulnerability in applications of different sizes.

vulnerability is higher than that with multiple types. The opposite is true for applications larger than 1 MB. Moreover, as the size of applications increases, the proportion of applications containing potential security vulnerabilities also increases, whereas the proportion of applications containing only one type of vulnerabilities relatively decreases. That is, the larger the application, the more likely it is to contain multiple types of security vulnerabilities.

4.2.2 Evaluate the accuracy of static analysis by a labeled dataset and dynamic executable scripts

First, we have developed a labeled application dataset containing various security vulnerabilities and evaluated the accuracy of our system based on this dataset. Next, we run real-world applications with potential security vulnerabilities detected by static analysis dynamically through our executable scripts, collect their runtime information, and further verify the accuracy of our static analysis for real-world applications.

(1) Accuracy on our labeled set

We have developed applications with corresponding security vulnerabilities and verified the accuracy of the detection system based on the correct detection of the corresponding vulnerabilities. The specific results are described in Table 4. The experimental results can confirm that the static analysis of our system is highly effective and can accurately detect the potential vulnerabilities of 56 applications in our test set.

Table 4 Static analysis results on our labeled dataset.

Vulnerability type	Number of test applications with corresponding type of vulnerability	Number of applications detected
Unrestricted component	15	15
Sensitive data processed in plaintext	12	12
Privacy leak by log	3	3
Dynamically loading file	9	9
Insecure password	7	7
Intent exposure	2	2
SQL inject	3	3
Insecure JavaScript in WebView	3	3
Normal applications without above vulnerabilities	2	0

(2) Further verify accuracy of static analysis for real-world applications by dynamic scripts

We can further confirm the existence of vulnerabilities by taking advantage of the results of static analysis and driving the application to run in accordance with specific executable scripts. We randomly select 20 real-world applications for each vulnerability category detected by static analysis for dynamic verification.

To achieve compatibility with more target applications, we use scripts based on coordinates rather than widgets. First, a target application is manually operated, and the script generation module automatically generates a corresponding executable script. We can apply the basic script in the coordinate format to other target applications without modification or minor modifications (usually replacing the package name and the starting activity name). This avoids substantial manual work. The dynamic scripts primarily enable the applications to execute specific operations of UIs. Finally, combined with information on API hooks and vulnerability patterns, the final verification is completed.

Our static analysis can detect numerous types of vulnerabilities. However, after careful assessment, we have found that not all vulnerabilities need to be further verified, which is a huge and unnecessary workload. The judgment rules of certain vulnerability patterns are very simple (such as unrestricted component). We need to focus on and verify the security vulnerabilities with more complex vulnerability pattern and are prone to incorrect judgment by the program. In the end, we further verify the following three types of vulnerability detection results through dynamic executable scripts.

Table 5 records our core ideas for validating applications that may contain the three types of vulnerabilities by dynamic executable scripts and API hooks. We use the dynamic executable scripts to drive the application execution, hook the corresponding system function in the execution, and record its running status. The results are as shown in Table 6. The data shows that our system static analysis detects vulnerabilities in real-world applications with high accuracy.

4.2.3 Detect other types of security vulnerabilities by dynamic analysis

Given the limited accuracy of static analysis and the incompleteness of vulnerability pattern, certain security vulnerabilities cannot be easily detected by static analysis technology alone. The dynamic executable script analysis can directly run the target application, drive it to execute the corresponding program logic that may trigger the vulnerability, and finally, complete the detection and judgment of the vulnerability.

Table 5 Dynamic analysis logic of 3 types of vulnerability.

Core logic of dynamic script	Running status record	Hook relative APIs function	Vulnerability type
Scripts drive applications to execute their general business processes.	Record the logcat of the app.	Log	Privacy leak by log
Scripts drive applications to execute their general business processes as much as possible.	Record the dynamic file loading of the application.	Runtime.exec ClassLoader ApplicationPackageManager	Dynamically loading file
Script drives the application to execute to the WebView and perform the actions in the interface.	Record the javascript code called from the WebView.	Addjavascriptinterface	Insecure JavaScript in WebView

Table 6 Results of dynamic verification.

Vulnerability type for dynamic validation	Number of applications detected by static analysis	Number of applications that are dynamically verified for vulnerabilities	Accuracy of static analysis for real-world apps (%)
Privacy leak by log	20	20	100
Dynamically loading file	20	19	95
Insecure JavaScript in WebView	20	19	95

We use the input unverified vulnerability as an example to illustrate the specific analysis of the vulnerability by our dynamic scripts. The types of vulnerabilities that can be analyzed by dynamic scripts rather than static analysis can be extended and are not limited to this type of vulnerability. Figure 10 describes the core script used for testing whether an application has input unverified vulnerability.

First, we use the script generation module to record all our manual operations on the target application. We correctly input the required information. Thus, the application completes the input successfully, jumps to the subsequent processes, and finally, generates an executable script for the whole processes. This condition requires a small amount of manual work. Here, we focus on the input verification of the login or registration interfaces.

Then, we arbitrarily construct the input information (one can manually modify the script, or the script generation module automatically generates the input text based on certain rules) and continuously update the newly generated base script. More updated scripts may be present. The updated scripts keep the application running. As long as the arbitrarily constructed input information (neither the standard required e-mail information nor phone number information, etc.) can also pass the verification and enter the subsequent process, we determine that the application has an input unverified vulnerability.

We dynamically analyze 20 applications known to have the input unverified vulnerability by our dynamic scripts and API hooks. Only one of the 20 target applications failed in the analysis. Further analysis shows that one of the application analyses failed because the application added measures that caused the re-signature to fail. Given that our dynamic script is based on the instrumentation, it cannot be dynamically executed if the re-signature fails.

```
...
 <TestCases>
  <TestCase name="Case0">
   ...
  <TestAction name="test.enterText(0, “information to verify”, false);"/>
  <TestAction name="test.recordReplay.clickOn(“id/button”, “0”, false);"/>
   ...
  </TestCase>
...
 </TestCases>
</TestSuite>
```

Fig. 10 A script for analyzing input unverified.

4.3 Comparison with existing systems

Table 7 lists the specific comparison between our system and previous relevant studies, where extension required means that the system cannot be directly used to detect this type of vulnerability, and extension development is needed. And partial means that for the types of vulnerabilities in Table 7, FlowDroid and the method in Ref. [12] do not detect all cases, and there may be other vulnerability situations. FlowDroid can directly detect partial sensitive data leaks. The analyzed vulnerability categories in Ref. [12] partially overlap with the “insecure password” vulnerability. The vulnerability categories detected by our system are relatively more comprehensive. Moreover, our dynamic analysis can further verify the existence of vulnerabilities and detect more types of vulnerabilities.

5 Limitation and Discussion

Our static analysis process, similar to most static analysis

Table 7 Comparison with existing systems.

Item	Our system	FlowDroid	Ref. [12]
Unrestricted component	√	×	×
Sensitive data processed in plaintext	√	Partial	×
Privacy leak by log	√	Extension required	×
Dynamically loading file	√	Extension required	×
Insecure password	√	Extension required	Partial
Intent exposure	√	Extension required	×
SQL inject	√	Extension required	×
Insecure JavaScript in WebView	√	×	×
Input unverified	√	×	×
Can dynamically verify some vulnerabilities?	√	×	×

systems, cannot handle several packed applications. Given that our system is extended and improves the early version of FlowDroid, it also features the corresponding limitations, such as excessive time consumption and false positives. In addition, if certain measures, such as signature verification mechanism, are added into the application, the re-signed application cannot be opened and run normally, which will result in the failure of our dynamic executable scripts. Expectedly, if we have a source code for the target application, no such problem would occur. In the future, we can improve our system to adapt to more applications.

6 Related Work

Most previous vulnerability analyses focused on static analysis. CHEX[13] is an Android application static analysis framework, which can analyze the data flow of Android applications and find the component hijacking vulnerability. Zhou and Jiang[1] mainly focused on content provider component and discovered two security vulnerabilities related to content providers, namely, passive content leak and content pollution, which may lead to the privacy data leakage in Android applications. However, they only focused on the vulnerabilities of content provider component, which have been patched by subsequent applications and systems.

Liu et al.[14] pointed out that current Android applications widely apply a deep link, that is, clicking a link can directly jump to a certain content page in the application; however, this technology faces the risk of being hijacked. To resist hijacking attacks and navigate to the right target content page of the Android application, we need to correctly use the legitimately validated deep link technology. Aonzo et al.[15] pointed out that the problem of phishing attacks is far from being solved. Existing Android password manager app and instant apps may run the risk of phishing attacks. Phishing attacks essentially exploit a vulnerability in the password manager application, and Aonzo et al.[15] offered a solution. However, the solution is only limited to this category of problems and cannot analyze the situation of multiple types of vulnerabilities in Android applications. Clickshield[16] indicates that existing solutions from academia and industry do not solve click-hijacking on mobile devices. Muslukhov et al.[12] reported that numerous Android applications use Java encryption methods incorrectly, resulting in security vulnerabilities in Android applications. Experiments showed that the third party library is the main source of this situation. Pariwono et al.[17] indicated that the discarded internet resources of Android applications can be used as the raw material for attackers to analyze and exploit vulnerabilities in Android applications. Uipicker[18] can automatically detect sensitive input operations, identify semantic information through layout information and program code, and further analyze the location where security-related information will appear.

Several studies are also based on dynamic analysis. Zhang et al.[19] proposed a kind of attack called "transplantation attack", which enables a malicious app to secretly capture private photos and bypass the API audit process of the Android system. Zhang et al.[20] were the first to put forward the runtime-information-gathering problem, which refers to the application's continuous collection of users' private information when running. TaintDroid[2] adds a taint mark to the source of privacy data and then dynamically tracks the spread of taint data through program variables, files, and inter-process communication messages. Finally, when the taint data leak out of the system through the network, they will generate a log record and remind users that the privacy data are leaked by the application. Considerable dynamic taint analysis work is based on this tool.

Instead of directly detecting the component vulnerabilities of Android applications, Ma et al.[21] determined whether there is a component hijacking attack by detecting the sensitive behavior of the applications. They transformed the problem of detecting component hijacking into a problem of determining whether the inter-component communication graph contains a given path. The effectiveness of this method was proven by analyzing 57 real-world applications. Several malicious applications are targeted because of important security vulnerabilities. In addition to traditional machine learning methods for detecting malicious applications, Droiddetector[22] takes advantage of deep learning technology to detect Android malwares. The extracted features include static and dynamic features, and the results show that the method can achieve high accuracy. Yan et al.[23] extracted the abstract syntax tree features of the Javascript code in applications and used the deep learning classification model for training. This method detects code injection attacks from Android hybrid applications with an accuracy of 97.55%. Maier et al.[24] pointed out that many malicious applications adopt a split-personality

behavior to hide malicious codes to avoid detection of antivirus software. Malicious applications use more dynamic code loading techniques than benign ones.

In a summary, the existing methods described above do not comprehensively consider the overall vulnerability analysis of the entire Android application, and the scope of their influence is limited.

7 Conclusion

We propose a novel hybrid approach for analyzing security vulnerabilities in Android applications. Our hybrid approach takes advantage of both static and dynamic analyses, and performs better than running two independent analyses. Our static analysis combines metadata and data flow analyses, whereas dynamic analysis consists of executable scripts and API hooks. We can obtain useful security vulnerability information of target application after our static analysis. Then, through dynamic executable script and hook API technology, we can further verify whether this type of vulnerability exists. In addition, our dynamic executable scripts can analyze vulnerabilities that are difficult to be detected by static analysis. We evaluate our system in detail both on Drebin and real-world applications. The system can effectively detect nine categories of important security vulnerabilities in Android applications, and dynamic executable scripts can avoid most manual operations. In the future, the detection of our system can also be extended to other vulnerability types.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (Nos. 2016YFB0800402 and 2016QY01W0202), the National Natural Science Foundation of China (Nos. U1836204, U1936108, 61572221, 61433006, U1401258, 61572222, and 61502185), and the Major Projects of the National Social Science Foundation (No. 16ZDA092).

References

[1] Y. J. Zhou and X. X. Jiang, Detecting passive content leaks and pollution in android applications, in *Proc. 20th Ann Network and Distributed System Security Symp*., San Diego, CA, USA, 2013.

[2] 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. 1–19, 2014.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, Drebin: Effective and explainable detection of android malware in your pocket, in *Proc. 21st Ann. Network and Distributed System Security Symp*., San Diego, CA, USA, 2014, pp. 23–26.

[4] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, Slicing droids: Program slicing for smali code, in *Proc. 28th Ann. ACM Symp. on Applied Computing*, Coimbra, Portugal, 2013, pp. 1844–1851.

[5] AXMLPrinter2, https://github.com/rednaga/axmlprinter, 2015.

[6] Smali/baksmali, https://github.com/JesusFreke/smali, 2014.

[7] Soot, https://github.com/Sable/soot, 2009.

[8] 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 Not*., vol. 49, no. 6, pp. 259–269, 2014.

[9] Robotium, https://github.com/RobotiumTech/robotium, 2010.

[10] Cafe, https://github.com/BaiduQA/Cafe, 2013.

[11] Xposed, https://github.com/rovo89/Xposed, 2011.

[12] I. Muslukhov, Y. Boshmaf, and K. Beznosov, Source attribution of cryptographic API misuse in android applications, in *Proc. 2018 on Asia Conf, on Computer and Communications Security*, Incheon, Republic of Korea, 2018, pp. 133–146.

[13] L. Lu, Z. C. Li, Z. Y. Wu, W. Lee, and G. F. Jiang, CHEX: Statically vetting android apps for component hijacking vulnerabilities, in *Proc. 2012 ACM Conf. on Computer and Communications Security*, Raleigh, NC, USA, 2012, pp. 229–240.

[14] F. Liu, C. Wang, A. Pico, D. F. Yao, and G. Wang, Measuring the insecurity of mobile deep links of android, in *Proc. 26th USENIX Security Symp*., Vancouver, Canada, 2017, pp. 953–969.

[15] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, Phishing attacks on modern android, in *Proc. 2018 ACM SIGSAC Conf. on Computer and Communications Security*, Toronto, Canada, 2018, pp. 1788–1801.

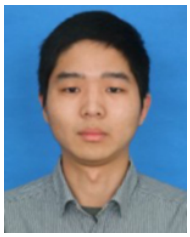
[16] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio, Clickshield: Are you hiding something? Towards eradicating Clickjacking on Android, in *Proc. 2018 ACM SIGSAC Conf. on Computer and Communications Security*, Toronto, Canada, 2018, pp. 1120–1136.

[17] E. Pariwono, D. Chiba, M. Akiyama, and T. Mori, Don't throw me away: Threats caused by the abandoned internet resources used by Android apps, in *Proc. 2018 on Asia Conf. on Computer and Communications Security*, Incheon, Republic of Korea, 2018, pp. 147–158.

[18] Y. H. Nan, M. Yang, Z. M. Yang, S. F. Zhou, G. F. Gu, and X. F. Wang, Uipicker: User-input privacy identification in mobile applications, in *Proc. 24th USENIX Conf. on Security Symp*., Washington, DC, USA, 2015, pp. 993–1008.

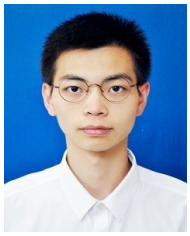
[19] Z. W. Zhang, P. Liu, J. Xiang, J. W. Jing, and L. G. Lei, How your phone camera can be used to stealthily spy on you: Transplantation attacks against android camera service, in *Proc. 5th ACM Conf. on Data and Application Security and Privacy*, San Antonio, TX, USA, 2015, pp. 99–110.

[20] N. Zhang, K. Yuan, M. Naveed, X. Y. Zhou, and X. F. Wang, Leave me alone: App-level protection against runtime information gathering on android, in *Proc. 2015 IEEE Symp. on Security and Privacy*, San Jose, CA, USA, 2015, pp. 915–930.
[21] C. Ma, T. Wang, L. M. Shen, D. K. Liang, S. P. Chen, and D. L. You, Communication-based attacks detection in android applications, *Tsinghua Sci. Technol.*, vol. 24, no. 5, pp. 596–614, 2019.
[22] Z. L. Yuan, Y. Q. Lu, and Y. B. Xue, Droiddetector: Android malware characterization and detection using deep learning, *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, 2016.
[23] R. B. Yan, X. Xiao, G. W. Hu, S. C. Peng, and Y Jiang, New deep learning method to detect code injection attacks on hybrid applications, *J. Syst. Software*, vol. 137, pp. 67–77, 2018.
[24] D. Maier, M. Protsenko, and T. Müller, A game of Droid and Mouse: The threat of split-personality malware on Android, *Comput. Secur.*, vol. 54, pp. 2–15, 2015.



Junwei Tang received the bachelor's degree from Huazhong University of Science and Technology in 2013. He is a PhD student in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include Android security and mobile internet security.

Ruixuan Li received the BS, MS, and PhD degrees in computer science from Huazhong University of Science and Technology, China in 1997, 2000, and 2004, respectively. He was a visiting researcher in Department of Electrical and Computer Engineering at University of Toronto from 2009 to 2010. He is a professor in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include cloud computing, big data management, and system security. He is a member of IEEE and ACM.



Kaipeng Wang received the bachelor's degree from Ocean University of China in 2018. He is currently studying for a master's degree in Huazhong University of Science and Technology. His research interests include android security and web security.

Xiwu Gu received the PhD degree in computer science from Huazhong University of Science and Technology in 2007. He is an associate professor of School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include distributed system, big data, and middleware.

Zhiyong Xu received the BS and MS degrees in computer science and engineering from Huazhong University of Science and Technology, China in 1994 and 1997, respectively, and PhD degree in computer engineering from University of Cincinnati in 2003. He is currently an associate professor in the Department of Mathematics and Computer Science at Suffolk University. His research interests include cloud computing, cloud security, high performance I/O and file systems, parallel and distributed computing. He is a member of IEEE.