Hindawi
Security and Communication Networks
Volume 2019, Article ID 7193684, 20 pages
https://doi.org/10.1155/2019/7193684

Research Article

SSLDetecter: Detecting SSL Security Vulnerabilities of Android Applications Based on a Novel Automatic Traversal Method

Junwei Tang,[1] **Jingjing Li**,[1] **Ruixuan Li**,[1] **Hongmu Han**,[1] **Xiwu Gu**,[1] **and Zhiyong Xu**[2,3]

[1]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China*
[2]*Math and Computer Science Department, Suffolk University, Boston, MA, USA*
[3]*Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China*

Correspondence should be addressed to Ruixuan Li; rxli@hust.edu.cn

Received 25 June 2019; Revised 10 September 2019; Accepted 30 September 2019; Published 31 October 2019

Academic Editor: Petros Nicopolitidis

Android usually employs the Secure Socket Layer (SSL) protocol to protect the user's privacy in network transmission. However, developers may misuse SSL-related APIs, which would lead attackers to steal user's privacy through man-in-the-middle attacks. Existing methods based on static decompiling technology to detect SSL security vulnerabilities of Android applications cannot cope with the increasingly common packed applications. Meanwhile, dynamic analysis approaches have the disadvantages of excessive resource consumption and time-consuming. In this paper, we propose a dynamic method to solve this issue based on our novel automatic traversal model. At first, we propose several new traversal strategies to optimize the widget tree according to the user interface (UI) types and the interface state similarity. Furthermore, we develop a more granular traversal model by refining the traversal level from the Activity component to the Widget and implement a heuristic depth-first traversal algorithm in combination with our customized traversal strategy. In addition, the man-in-the-middle agent plug-in is extended to implement real-time attack test and return the attack results. Based on the above ideas, we have implemented SSLDetecter, an efficient automated detection system of Android application SSL security vulnerability. We apply it on multiple devices in parallel to detect 2456 popular applications in several mainstream application markets and find that 424 applications are suffering from SSL security vulnerabilities. Compared with the existing system SMV-HUNTER, the time efficiency of our system increases by 38% and the average detection rate increases by 6.39 percentage points, with many types of SSL vulnerabilities detected.

1. Introduction

With the rapid development of mobile Internet, smartphones can help users to obtain information and services from the Internet anytime and anywhere. According to IDC, about 1.42 billion smartphones have been shipped worldwide in 2018. Among the mobile operating systems, the Android holds a commanding market share of over 85%, and as of August 2018, there are about 2.8 million Android apps on Google Play. A variety of mobile applications have provided great convenience for our life. For example, social applications help us keep in touch with friends and financial applications help us conduct investment and financial management.

Many applications need to interact with Internet frequently. Users' sensitive information may be transmitted during network communications, which may cause privacy leakage. To protect users' privacy in network communications, the applications can use encryption-based functions and SSL protocols. However, recent research has pointed out that about 88% of Android applications misuse the Java crypto APIs and cause security vulnerabilities [1]. Although the Android SDK implements the SSL protocol (in this paper, SSL means secure sockets layer protocol SSL and transport layer security protocol TLS) to protect communication security, the situation has not been greatly alleviated. This is because SSL protocol also has a series of security problems. First, there are vulnerabilities in the design of SSL protocol itself, such as the "Heartbleed" vulnerability of OpenSSL reported in 2014. Secondly, SSL protocol implementation involves encryption and decryption, identity authentication, certificate management, and other various technologies. And each of them corresponds to some

vulnerability risks, such as weak encryption algorithm, encryption hash conflict [2], certificate compromise [3], and private key leakage. Finally, there are numerous cases of man-in-the-middle (MITM) attacks on users caused by developers' incorrect implementation of SSL-related interfaces. According to a report of Tencent network security center, most Android applications in China that use SSL protocol are not strictly abide by the standard in the Google official technology documents. These apps directly trust all certificates, leading to privacy leakage in communications, even remote code execution. It points out that even popular applications also have this vulnerability, and such security threats are extremely common. Mobile developers use a custom certificate to connect to the background server with SSL but fail to correctly implement the authentication logic of server certificate and host name or just call the wrong APIs, resulting in SSL vulnerabilities.

The Android application ecosystem is made up of app developers, users, and app market. Due to the openness of Android system and the low threshold of development technology, it is difficult for average developers to design applications from the perspective of security development. Thus, the security of the applications is mainly undertaken by the application markets. According to incomplete statistics, there are currently more than 50 Android application markets in China. Due to the uneven scale of the application markets, not all of them have perfect application security audit mechanism. When the application market provides users the applications with SSL security vulnerabilities, users will be attacked by a man-in-the-middle attack without knowing it, resulting in serious security consequences.

Existing methods for detecting security vulnerabilities in Android applications mainly include two types: static code analysis and dynamic execution analysis. However, with the development of Android application packers, there are many bugs and false positives in the existing detection schemes based on static decompilation technology. The static analysis cannot judge whether SSL security vulnerability logic is reachable, leading to high false positives and omissions. In fact, previous static code analysis methods are no longer working in many cases. At the same time, there is relatively little research on dynamic analysis to detect vulnerabilities in Android applications. Dynamic analysis is becoming a new trend in detecting SSL security vulnerabilities in Android applications. However, the existing schemes have the following problems. First, existing dynamic analysis still relies on the results of static analysis to build a collection of target *Activity* components or jump paths. Thus, applications where static analysis is unable to obtain the required information cannot be dynamically analyzed effectively. Secondly, some schemes need to modify the Android system or Android application structure to achieve automated testing. Thirdly, the direct jump of the Activity leads to the lack of contextual semantics related to functional logic in the application runtime, and the exhaustive traversal of the set of operable widgets of each Activity will lead to path expansion and lots of time cost. Finally, previous traversal model based on the Activity component will cause the detection system to obtain some invisible widgets during the running process, thus causing null pointer exception. The existing dynamic execution analysis methods can be hardly used for SSL security vulnerability detection in Android applications.

Therefore, we propose a novel and efficient automatic detection method for SSL security vulnerabilities in Android applications based on our customized UI traversal strategy. The main contributions of this paper are as follows:

(1) We propose a new traversal strategy by optimizing the widget tree according to the UI types and calculating the interface state similarity based on the widget path set. This detection method does not rely on static decompression technology to build a collection of *Activity* components that contains vulnerability points, but adopts heuristic search to realize automatic detection of vulnerability points.

(2) We propose a novel and more granular traversal model by refining the traversal level from the *Activity* component to the *Widget* and implement a heuristic depth-first traversal algorithm in combination with our customized traversal strategy. We believe that our model is not only limited to SSL vulnerability detection in Android applications but also can be extended to more application scenarios.

(3) The detection system enables multiple test devices to run in parallel. To distinguish the network data packets generated by the applications running on each test device and conduct man-in-the-middle attack tests in real time, we extend the man-in-the-middle agent plug-in to implement the real-time attack test based on network port monitoring and return the attack results in time to guide the detection system operation.

(4) We design and implement the Android application SSL security vulnerability automatic detection system *SSLDetecter* and verified the effectiveness and availability of the system through experiments. Compared with the existing system SMV-HUNTER, the time efficiency of our system increases by 38% and the average detection rate increases by 6.39 percentage points, with more types of vulnerabilities detected.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 introduces the related research work. Section 4 describes the UI traversal strategy. Section 5 presents the framework of automatic detection of SSL security vulnerabilities in Android applications and its implementation. Section 6 shows the experimental details and analysis. Section 7 presents conclusions and discusses the future work.

2. Background

2.1. User Interfaces of Android Applications. The user interface of Android applications is mainly displayed through the *Activity* component. Users can interact with the screens they provide. Each *Activity* gets a window to draw its user interface, usually the same size as the screen, but sometimes

smaller and floating on top of other windows. An *Activity* may contain multiple views, which are composed of UI and events. Therefore, view objects are the smallest units of user interface. A widget in an Android application is essentially a view that represents a UI, such as a Button, EditText, and TextView.

We can divide the widgets of the Android application interface into operable and nonoperable categories. Operable widgets are widgets that can be clicked, swiped, or inputted, such as buttons, list bars, and text boxes. Nonoperable widgets are basically static widgets that do not interact with the users. Furthermore, in addition to system widgets, there are also developers' custom widgets. We know that all widgets in Android are subclasses of *View* and all layout containers are subclasses of *ViewGroup*, so if the widgets provided by Android cannot meet developers' needs, they can choose to inherit the *View* or *ViewGroup* to customize their own widgets. Since our SSLDetecter system is based on Appium [4] to obtain the widgets information of the application, as long as it is the UI widget information that Appium can recognize and obtain, our detection system can effectively process it. We represent the widget information obtained on the application interface as a graph (tree) structure, which is convenient to design a heuristic depth-first algorithm to traverse the interfaces of the application.

2.2. SSL Security Vulnerabilities in Android Applications. SSL protocol was developed by Netscape in 1994 and then became the most famous web security mechanism in the world. SSL is a security protocol that provides security and data integrity for network communication. TLS and SSL encrypt network connections at the transport layer. The main functions of SSL protocol are to provide encryption, authentication, and data integrity verification. That is, SSL protocol can encrypt data and prevent data from being stolen during transmission. The SSL protocol also provides optional client-side validation and mandatory server-side validation. This ensures that the data are sent to the correct target client and server. In addition, the SSL protocol can also verify the integrity of the data and ensure that the data are not tampered with during the transmission. The SSL protocol is between the application layer and the TCP layer, where data are no longer passed directly to the transport layer but to the SSL layer. The SSL layer encrypts data received from the application layer and adds its own SSL header to the packets.

Android also uses the SSL protocol to secure data transfers. The SDK (Software Development Kit) at the Android system level has helped ordinary developers implement the details of SSL protocol. Developers only need to call the APIs of the system to use SSL protocol. The SSL protocol involves many processes and is very complex. It is not easy for the average developer to properly use the relevant SSL function interfaces during their developing process. Furthermore, there may also be some bugs in the SSL protocol itself. All of these can lead to SSL-related security vulnerabilities in Android applications.

3. Related Work

This paper summarizes the related research work from two aspects: SSL security and Android application security vulnerabilities.

3.1. SSL Security. There are many researches on SSL security, including the research of the protocol and the use of SSL in Android applications. Wang [5] points out that the serial number in x.509 certificate may be predicted successfully, which needs to be improved. Wei and Wolf [6] point out that many Android application developers do not use SSL protocol properly to protect user data. The reasons include server-side configuration errors, incomplete documentation, vulnerabilities in third-party libraries, complexity of the PKI system, and the fact that users do not pay much attention to HTTPS. Fahl et al. [7] verify the validity of the domain name certificate by extracting URL links (HTTPS and HTTP) in the decomposed code of the Android application, so as to determine whether the application has SSL security vulnerability. They only extract static URLs, and dynamic URL link certificates cannot be obtained effectively. Meanwhile, security experts need to run the application to confirm manually to avoid false positives. Based on this study, they use a similar method to verify the security of SSL implementation in IOS applications [8]. Although the IOS system is a closed source, they still find 98 applications that do not implement correct SSL certificate authentication among 697 applications that use the SSL protocol.

Georgiev et al. [9] first pointed out SSL security vulnerabilities in the SSL libraries, middlewares, and nonbrowser applications by using dynamic black box testing technology. The study reveals the severity and prevalence of SSL security vulnerabilities but do not propose an automated detection scheme. Onwnzurike and De Cristofaro [10] carry out manual man-in-the-middle attack test on 100 applications with more than 10 million downloads in the market. They point out that although SSL security vulnerability is proposed in 2012 and have attracted attention from the security community, there are still 32 applications accepting all certificates and host names by default. Furthermore, when the attacker's certificate is installed on the victim's mobile phone, up to 91% of the apps are at risk of man-in-the-middle attack. Conti et al. [11] propose a dynamic HTTPS monitoring system named Mithys and develop a prototype named MITHYAPP to play the role of VPN on mobile devices. But the system just focuses on the security of user information on the mobile devices and is not suitable for SSL vulnerability detection. However, MITHYAPP will also cause new security problems. For example, Ikram et al. [12] point out the risks of unsafe VPN tunnel protocol, traffic leakage, and third-party library embedding in Android VPN applications. Sounthiraraj et al. [13] apply technologies such as static analysis, UI traversal, and intelligent input to SSL security vulnerability detection of Android applications and design SMV-HUNTER framework. SMV-HUNTER first introduces UI traversal technology into SSL security vulnerability detection, but it relies on static analysis to build

the set of target components, and application failures occur frequently due to the lack of context logic cycle to start up components. Thus, the effect is limited. For the error handling of the *onReceivedSSLError* interface call when the SSL certificate verification fails in WebView, Gagnon et al. [14] realize the visual Android application SSL security vulnerability detection tool AndroSSL from the developer's point of view. The tool is based on AVP (Android Virtual Platform) to execute a manually recorded script that triggers a sequence of link request events to drive application execution.

3.2. Android Application Security. In addition to analyzing SSL usage in Android applications, there are several studies that focus on other security issues in Android applications. Munivel and Kannammal [15] point out that phishing attacks are still very common on mobile smartphones today. A new authentication framework is proposed, which does not require password transmission but is based on zero-knowledge proof to identify communication entities. Li et al. [16] note that many of the repackaged Android apps have a lot of malicious piggybacking. This is a way for malicious code to spread. They analyzed the specific situation in depth and provided supports for further detection of the malicious code. Maier et al. [17] point out that many Android malicious applications currently separate malicious code from normal code and that malicious code runs more covertly and can easily bypass lots of checking methods. Enck et al. [18] propose a TaintDroid system that can dynamically track tag data in Android applications. Many other dynamic analysis systems are based on this. AppSpear [19] is a system for automatic analysis of packed Android applications. The system uses some unpacking strategies to effectively restore some hidden code and generate complete executable codes of the Android application.

In summary, SSL security issues still need to be taken seriously in Android applications. Some of exiting methods cannot effectively deal with the analysis of packed applications [20], and in many cases, it does not work. Some others also has disadvantages, such as time-consuming analysis and system modification, which make it difficult to effectively detect SSL security vulnerabilities. How to automate and effectively detect SSL security vulnerabilities in Android applications is still a challenging problem that has not been well solved. Based on our previous work [21], we have proposed a new automatic traversal method to detect SSL security vulnerabilities of Android applications. Our new method is described in detail below.

4. UI Traversal Strategy

Android applications realize user interaction through the component named *Activity*. Each *Activity* corresponds to one or more interfaces (views). The interface contains a number of widgets that are bound to listen to user events, and the user manipulates these widgets to perform program functions. In a directed graph that only knows the initial node, the interface jump is generated by simulating the user action to operate the widget event of the initial node, then generate a new node, and loop back and forth, thereby continuously expand the traversal of the node graph. That is the whole process of automatic operation of the Android application. How to design the traversal model, customize the processing strategy of interface widget tree, and calculate the interface state similarity will affect the detection efficiency of our system. We call these UI traversal strategies. The following subsections detail our new traversal model, the new interface widget tree optimization strategies, and the new interface state similarity calculation method.

4.1. Our Traversal Model. The traversal model is a finite state machine to represent the interface (view) states and the transition process between them. We build our novel GUI traversal model for Android applications based on the widget level. We give specific definitions of our traversal model below.

Interface state consists of binary group < interface widget tree, Activity name>, and is used to identify different interfaces in the process of application running. The interface widget tree refers to a tree with widgets as its nodes, which is built according to the hierarchical relationship and the dependency relationship of widgets in the interface. Each node holds the widget's structural information (parent widget, child widget, and widget path) and widget property information (such as widget type, text information, and clickable). The Activity name defined by its fully qualified type name represents the Activity to which the interface state belongs.

Actions are specific actions that cause changes in the state of the interface. They can be divided into simple actions and combined actions. Simple actions consist of a triple < widget ID, event type, and additional information>. Widget ID is used to identify the widget. The event type indicates the action performed by the widget. Additional information provides widget action assist information. Combined actions consist of two or more simple actions.

GUI node consists of a binary group < interface state, task list>. The interface state describes the basic information of the GUI node. The task list holds information about the tasks that the GUI node needs to perform.

Task represents by a binary group < action, GUI node>, indicating that the action is executed only under the GUI node.

GUI jump consists of a binary group < task, GUI node> which is used to describe the jump relationship of GUI nodes. The task is a task in the task list of precursor node, and the GUI node is the destination node to jump to after executing the task.

GUI directed graph consists of a binary group <GUI node set, GUI jump set>. GUI node sets constitute all vertex information in GUI directed graph, and GUI jump sets constitute all edge information in GUI directed graph.

The process of building our GUI traversal model is the process of dynamically extending GUI directed graph. After entering the application, the first interface corresponds to the starting GUI node of the GUI directed graph, and the

interface is analyzed to generate a task list. When an application executes a task in the task list, the GUI directed graph is updated according to the interface state after the task is executed. When the interface state remains unchanged, the GUI directed graph does not change. When the interface state is transformed into an existing interface state, the GUI jump occurs. The jump forms a new GUI node and adds the edge between the previous GUI node and the node corresponding to the current interface state, and updates the task list to the task list corresponding to the current GUI node. When the interface state is transformed into a new interface state, a GUI jump occurs too. A new GUI node is formed, and the edge from the previous GUI node to the new GUI node is added. Then analyze the new GUI nodes and build the task list. To dynamically expand the GUI directed graph, the tasks in the current GUI node task list are sequentially executed in a loop.

4.2. Interface Widget Tree Optimization. Each interface corresponds to a widget tree. If we exhaust the simulation of all the actionable widget events on the widget tree, not only will the traversal space swell, but also this is not allowed in the detection time. For example, when we traverse to a slidable page that contains a news list, it would take a lot of time to click on each piece of news. However, in reality, these widgets trigger the same network link only with different parameters. Second, improper handling of widgets can also cause the traversal aborts. For example, when a prompt box pops up on the interface, the prompt box will get the window focus, so that the interface widget behind cannot be clicked. At this time, if the pop-up box cannot be closed correctly, it will directly jump out of the interface to be tested and terminate the traversal. Finally, when there are many types of events on an interface, such as input box, selection box, button, and so on, it is especially important to establish the sequence of event triggers. To deal with the problems above and to trigger as many effective network request events as possible in our system, we propose an optimization strategy of the widget tree according to the interface categories.

After manually analyzing 200 runtime screenshots of applications, we have divided the interface into six categories: welcome interface, popup interface, list interface, detailed interface, input interface, and other interfaces from three aspects of interface area, widget tree structure, and specific widgets. Table 1 describes the feature information of each category of interfaces. During the application running process, the interface category is identified by feature matching and different widget tree optimization strategies are conducted to trigger as many effective network requests as possible to cover the suspected SSL vulnerability points. The following Strategies 1 to 6 correspond to the six categories of interfaces, respectively. At the same time, after each kind of interface is processed, all the actionable widget events in the optimized widget tree are cropped according to Strategy 7 and Strategy 9, and the action sequence is built according to the specified priority.

Strategy 1. When the interface is identified as the input interface, widget nodes such as *EditText*, *SearchView* and *CheckBox* are searched in the widget tree to obtain information such as text, content-desc, and resource-ID of the widget, so as to identify the type of information that the node needs to input. The input library we built is just several XML configuration files, including various information such as mobile phone number, e-mail address, gender, and search keywords. The input library is then matched to generate input actions expressed as a triple < widget ID, "input action," "input contents" >. Simultaneously search for the sibling node of the widget, the parent node, or the sibling node of the parent node of the widget, then match the corresponding button node according to the context information, generate < widget ID, "click" > button action, and encapsulate the two actions into combined action.

Strategy 2. When the interface is identified as a list interface, we first build a path set from the root node to all the leaf nodes. The path set consists of a sequence of widget types, such as "*framelayout-viewpager-listview-framelayout-textview*." Then we filter out the paths that contain the scrollable *ListView*, *RecyclerView*, and *ExpandableListView* nodes from the path set. And those path sets which have more than 10 the same paths will be also filtered out. Finally, the same paths in the list path set are filtered, and only the *Actions* of one of the multiple child nodes under the list widget are generated.

Strategy 3. When the interface is recognized as a pop-up box, we will carry out text matching analysis on the prompt message of the interface. If the text prompt is to update or download information, select the reject button such as "cancel." For AD pop-ups, select the "close" button. For menus and other system prompt boxes, we generate the actions in turn according to the hierarchical traversal order of actionable widgets.

Strategy 4. When the interface is identified as a detailed interface, we search the "return" widget on the interface or add a return action and directly return to the previous interface, without processing other widgets on the interface.

Strategy 5. When the interface is recognized as a welcome interface, we generate sliding actions on the interface, including left sliding, right sliding, up sliding, and down, to make it jump to the real initial interface.

Strategy 6. When the identified interface is other interfaces, generate an action sequence just according to the hierarchy traversal sequence of the interface widget tree.

Strategy 7. Set the processing priority of different actions on each interface: combined action > click action > sliding action.

Strategy 8. Set the priority of the same action processing on each interface according to the hierarchy traversal sequence of the interface widget tree.

Table 1: Description and characteristics of different categories of interfaces.

Interface category	Interface description	Interface characteristics
Welcome interface	The initial display of the AD page when the application is launched	Contains the *ImageView* widget and the widget area is equal to the screen area; less clickable widgets; support for sliding operation
Pop-up interface	Version update pop-ups, AD pop-ups, system prompts, menus, etc.	The interface area is smaller than the screen area; the underlying interface is out of focus
List interface	The interface is the home page of the application or one of its topics, containing a scrollable list or a number of identical widgets	There are a large number of structurally identical subtrees in the interface widget tree; or contain scrollable *ListView*, *RecyclerView*, etc.
Detailed interface	Describe a topic in detail, mainly with lots of pictures and words	Contains a back button, a large number of images, text, or *WebView* widgets
Input interface	The interface is a login interface, registration interface, etc.	Contains widgets such as *EditText*, *SearchView*, and *CheckBox*
Others	Do not belong to the above five categories of interface	No obvious characteristics

Strategy 9. When the "clickable" property of both parent and child widgets in the interface widget tree is "true" and the child widget overrides the parent widget, we crop the parent widget and only generate the action of the child widget.

4.3. Interface State Similarity Calculation. During dynamic construction and traversal of the directed graph, each node corresponds to a different interface state. After each widget action is executed, we need to take different approaches depending on whether the state of the interface has changed. Currently, there are three solutions to determine whether the current interface has changed. (1) Compare the Activity name where the interface is located [22]. (2) Calculate the similarity between the screenshot of the current interface and the screenshot of the interface that has been traversed through the image recognition technology. (3) Extract all widget types, widget quantity, and widget property information on the interface for comparison [23]. Solution 1 is not accurate enough because the same Activity may contain multiple different interfaces. Both solution 2 and solution 3 require a large amount of comparison time and are inefficient. It can be seen that too fine or too thick comparison will result in poor traversal effect. Considering that we focus on the analysis of different network requests in the application and the common widgets among different interfaces (if the widgets corresponds to sending network request event) often correspond to the same request links, we propose to calculate the similarity of two interface states based on the widget path set. Formula (1) is used to calculate the similarity $\mathrm{sim}(s1, s2)$ of the interface state s1 and interface state s2:

$$\mathrm{sim}(s1, s2) = \begin{cases} 0, & \mathrm{activity}(s1) \neq \mathrm{activity}(s2), \\ \dfrac{|R(s1) \cap R(s2)|}{|R(s1) \cup R(s2)|}, & \mathrm{activity}(s1) = \mathrm{activity}(s2). \end{cases} \tag{1}$$

Formula (1) indicates that if $s1$ and $s2$ belong to different Activities, they correspond to different interface states and the similarity of them is zero. When $s1$ and $s2$ belong to the same Activity, the similarity of the corresponding interface state can be calculated based on the widget path set. The $R(s1)$ and $R(s2)$ in the $|R(s1) \cap R(s2)|/|R(s1) \cup R(s2)|$ represent all the paths from all root nodes to leaf nodes of the interface widget tree $s1$ and $s2$, respectively. $|R(s1)| \cap |R(s2)|$ represents the number of identical paths in these paths. $|R(s1)| \cup |R(s2)|$ represents the number of all the paths from the root node to the leaf node. The two paths are the same if and only if the total number of widgets, the widget type, and the widget text on the same hierarchy of both paths are consistent.

This method is only used for the pairwise calculation of the similarity of interfaces belonging to the same Activity, effectively reducing the comparison time. When the similarity is greater than a certain threshold value, it indicates that the interface state is an old state. When the similarity is less than a certain threshold value, it indicates that the interface state is a new state. By setting similarity threshold, the scale of traversal model can be controlled effectively. When the similarity threshold value is higher, the interface state is divided more finely and the traversal coverage rate is higher. This means that the traversal directed graph constructed is large and cumbersome, and it takes lots of time to traverse.

5. System Design and Implementation

5.1. System Architecture. The architecture of Android application SSL security vulnerability automatic detection system based on dynamic UI traversal is shown in Figure 1, which mainly includes three modules: preprocessing, UI traversal, and attack test. According to the three characteristics including application networking permission, packing, and SSL-related interface call, the preprocessing module will screen out the suspicious application set from the test Android application dataset to reduce the size. The UI traversal module distributes the (Android Package) apk files of the suspect application to the idle devices (or emulators) through our scheduling logic. At the same time, our UI traversal technology drives the application to install, run, and uninstall on the device. The whole UI traversal process includes widgets extraction, traversal algorithm design, and

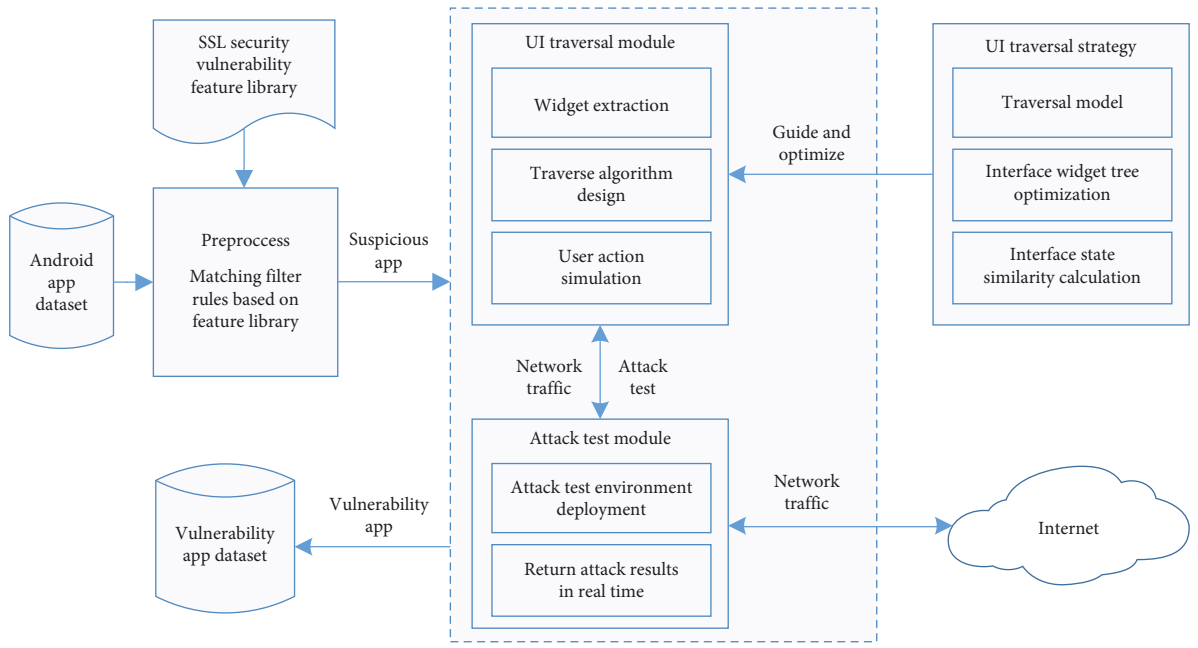

Figure 1: Architecture of Android application SSL security vulnerability automatic detection system.

user action simulation. Among them, UI traversal strategy is used to guide and optimize the process of UI traversal. The attack test module attacks the application running on the device through the man-in-the-middle attack agent tool and returns the attack results in real time. The implementation details of each module are described below.

5.2. Preprocess. The preprocessing module uses static code analysis and feature matching technology to filter the Android application dataset preliminarily and screens out the "suspicious application" set. The whole filtering process consists of building SSL security vulnerability feature library and matching filtering rules. This process is designed to initially filter some applications and determine whether the application belongs to one of the four SSL vulnerability types. In fact, this preprocessing is not always necessary and will not affect the subsequent detection efficiency and progress, because SSLDetecter can also directly conduct dynamic analysis of applications.

5.2.1. SSL Security Vulnerability Feature Library. Based on existing research and combining with specific vulnerability code blocks in Android applications, we divide SSL security vulnerabilities of Android applications into four types: *trusting all certificates*, *trusting all host names*, *calling insecure SSL APIs*, and *ignoring SSL errors*. SSL security vulnerability characteristics are formally described as

$$X = (x_1, x_2, x_3, x_4), \tag{2}$$

where X represents the SSL-related function summary information of the Android application, x_1 represents the function name summary information, x_2 represents the summary information of the interface implemented by the function, x_3 represents the summary information of a class that the function inherits, and x_4 represents the summary information of the function body. The above formal representation is used to describe four known SSL security vulnerabilities to form the vulnerability feature library. For example, the formal description of the characteristics of trusting all hosts vulnerability is shown in Figure 2.

5.2.2. Matching Filter Rules. Based on Androguard [24], feature extraction is carried out to match SSL security vulnerability feature library. Firstly, decompress the test apk file and parse the decompressed Androidmanifest.xml file through the AXMLPrinter2.jar package. Then extract all permissions applied by the application and judge whether there are *network* permissions. For applications with network permissions, we further analyze the smali class files which are obtained by decompiling the dex files of the app. We extract the specific information of SSL-related functions in each smali file according to the functional syntax structure of the smali file. The information includes information of the class inherited by the function and the implemented interface. Meanwhile, the function body information is extracted based on the widget flow graph provided by Androguard at the instruction level. Finally, the extracted information above is uniformly expressed in the form required by the SSL security vulnerability feature library as the characteristics of the application. For the application that can extract the features, we match the feature information with the vulnerability features in the vulnerability feature library one by one. When the matching is successful, it indicates that the application is suspicious and will be stored in the dynamic test application database; otherwise, it will be stored in the white list database. For the applications which cannot be decompiled (such as packed apps), we directly put them into dynamic test application database.

```
Trust_Manager_Signature = (Method,Interface,Class,Method_Body)
Method = {'access_flags':'public','return':'void','name':'checkServerTrusted',
    'params':['java.security.cert.X509Certificate[]','java.lang.String']}
Interface = ['Ljava/net/ssl/TrustManager;','Ljavax/net/ssl/X509TrustManager;']
Class = []
Method_body = [return-true,return-void]
```

Figure 2: The formal description of the characteristics of trusting all hosts vulnerability.

5.3. UI Traversal Module. Appium is a good testing framework for mobile applications. Our UI traversal module drives the application to run automatically by extending the traversal logic of Appium. The module mainly consists of three parts: widget extraction, traversal algorithm design, and user action simulation. Appium bottom layer implements the similar functions of the native test tool Uiautomator that comes with the official Android SDK. Based on the Appium framework, it is very convenient to obtain widget information on the interface and simulate user action, but the framework does not put forward the traversal model and optimization strategy of the widget tree on the interface. It cannot realize the requirements for the test applications to continuously traversal to trigger each functional logic. SSLDetecter extends the Appium client traversal logic on its framework to automatically detect SSL security vulnerabilities of Android applications.

5.3.1. Widget Extraction. Appium encapsulates the Accessibility Service, and all widget information in the current interface can be obtained by calling the *getPageSource*() method. All widget information is marked by the extensible markup language (XML). The widget type is treated as a label, and the widget property information is described in the form of <Key, Value > pair. Firstly, the SAX interpreter is used to parse the XML file to obtain the property information of each widget. It mainly includes widget type (class name), clickable or not, location information (bounds), content information (content-desc), etc., and the dependency and hierarchical relationship among widgets to build the interface widget tree. Then the interface type is identified according to the interface characteristics, and the widget tree optimization strategies are carried out. Finally, generate the action sequence of the interface.

5.3.2. Traverse Algorithm Design. According to our GUI traversal model, a heuristic depth-first traversal algorithm is proposed. Algorithm 1 describes the detailed process of the algorithm. Take the apk file of the test application as input and install the application to run. Line 1 initializes the directed graph *GUIGraph* and the current GUI node *cur* to null. Lines 2–4 judge the interface type of the first interface entered after application's startup, optimize the widget tree, generate the initial node, and assign it to the current node cur, and add the cur node to the node set of *GUIGraph*. Lines 5–7 loop to fetch the task execution from the task list of the current node *cur* until the traversal condition is reached: the task list is empty or SSL security vulnerability has been detected. Lines 8–20 deal with the situation when the task execution causes the interface to jump. Line 9 calculates the similarity between the interface *next* after the task has been executed and the interface that has been traversed. Lines 10–12 indicate that when the state of *next* is an old node that has been traversed, then the old node is assigned to the current node *cur*. At the same time, the information about the current task that causes the jump to the old node is added to the *GUIGraph* jump list. Lines 13–15 determine that when the old node is not the initial node initial and the task is empty, the return operation is performed to continue traversing the task list of the previous node. Lines 17–20 describe that when the *next* node is in a new state, a new GUI node is generated, which is assigned to the current node *cur* and processed to generate a task list. And add the new node to the *GUIGraph* node set and add information about the current task that causes the jump to the new node to the *GUIGraph* jump list. When the interface state does not change and the task list of the current node is not empty, continue to execute the remaining tasks in the task list and loop until the condition of traversal termination is reached.

5.3.3. User Action Simulation. There are two types of events in Android apps: button events and touch events. Button events refer to the home button, volume key, return key, and power button on mobile devices. These events are triggered by the event code of the key. The Appium framework client provides a pressKeyCode() method to simulate button events. Touch events refer to the events that simulate the operation of users on the mobile phone screen, such as click events, input events, and sliding events. For touch event emulation, you first need to locate the widget that you want to operate on and then perform the corresponding action according to the widget type. Appium client can locate widgets in various ways through the *UIAutomator* tools, such as widget ID, widget name, widget class name, widget XPath path, widget CSS properties, and so on. There are also rich APIs for action emulation for different types of widgets, such as button execution operation of the API AndroidElement.click().

Table 2 describes the action classification and corresponding API information based on the traversal model. The wait event in the table is a kind of delay action added after the execution of a specific click action, considering the situation of interface loading and plug-in downloading during the application running. Combined actions are mainly divided into one or more input boxes and one-click action combination or multiple-click action combination, mainly refer to the application login and registration, search, and check box events. To validate the input box during the application run to pass data validation, we need to build a series of valid information in advance. This mainly includes authentication information and search information, wherein the authentication information includes a user name, an e-mail address, a mobile phone number, a zip code, a password, etc., and the search information includes food, books, movies, clothes, geography, etc. They are stored in the detection system in an *xml* file format. When an input action is encountered, an

```
Input: apk file of the test app
(1)  Initialize GUIGraph, cur, initial⟵∅
(2)  After the test apk starts, the interface type judgment and widget tree optimization are carried out to generate the initial node
(3)  cur = initial;//assign the initial node to the current node
(4)  GUIGraph.Nodes.add(cur) //add the initial node to the list of nodes in the GUIGraph
(5)  While cur.curTasks.hasNextTask( )and not meet the stop condition do
(6)    curTask = cur.curTasks.nextTask( ); //Take the task from the task list
(7)      curTask.getEvent( ).runEvent( ); //Simulate event action
(8)    If the state of cur changed then//Interface state changes
(9)      sim = calculateSimilarity(new Similarity(next, GUIGraph))
//calculate the interface state similarity, sim is a Map structure and the keys are similar nodes, the values are the results of similarity
(10)     If sim.getValue( ) ≥ similarity threshold then//indicate a jump to a node that has been traversed
(11)       cur = sim.getKey( ); //the old node is set to the current node and perform the old node task
(12)         GUIGraph.transferList.add(curTask, cur); //add jump information
(13)       If cur.curTasks.size ( ) is 0 and cur ≠ initial then
(14)         cur.addTask(curNode, new BackEvent( )); //add back action
(15)       End If
(16)     Else
(17)       cur = next; //generate new GUI nodes
(18)       cur.curTasks.add(generateTask( )); //generate the task list for the new node
(19)       GUIGraph.Nodes.add(cur); //Add node information
(20)       GUIGraph.transferList.add(curTask, cur); //add jump information
(21)     End If
(22)   End If
(23) End While
```

Algorithm 1: Heuristic depth-first traversal algorithm.

Table 2: Action classification and corresponding API.

	Action category	The specific action	Corresponding API
Action	Simple action	The keys	AndroidDriver.pressKeyCode()
		Click	AndroidElement.click()
		Input	AndroidElement.sendKeys()
		Sliding	AndroidDriver.swipe()
		Wait	AndroidDriver.manage().timeouts().implicitlyWait()
	Combined action	Login register search	AndroidElement.sendKeys() AndroidElement.click()
		Checkbox	AndroidElement.click() AndroidElement.click()

appropriate data input is selected from the input library according to the prompt information of the input box.

5.4. MITM Attack Test Module. Man-in-the-middle attack tests of an application that is running automatically requires identity spoofing between the test device and the server, intercepting network packets that communicate between the test device and the server and forwarding them. The middleman agent tools such as Fiddler, MITMMRoxy, and Burp Suite implement the above functions very well. We implement the attack tests based on the *Burp Suite* agent. The implementation details of the MITM Attack Test Module and the process of interacting with the traversal strategy are shown in Figure 3.

5.4.1. Attack Test Environment Deployment. The man-in-the-middle attack test is implemented based on *Burp Suite*. It is one of the popular web application penetration test tools, which can effectively intercept, tamper with, and encapsulate the network requests of the applications. First, we install the Burp Suite client proxy software on the computer and then configure the static proxy IP of each Android phone to be the IP address of the computer where Burp Suite is located. The ports are incremented from 8080, and the listening address is set under the proxy function menu of the Burp Suite tool. The IP address and port settings in the tool should be the same as those in the test smartphone. Each test Android smartphone is connected with the computer through USB and interacts with SSLDetecter system through (Android Debug Bridge) ADB commands. The Android smartphones and the computer need to be in the same (local area network) LAN. At this time, the entire attack test environment is initially set up.

5.4.2. Proxy the Network Traffic of the Application. This step is primarily viewed with the Proxy tool in the toolkit. Proxy is a Proxy server that intercepts HTTP/S. As a middleman

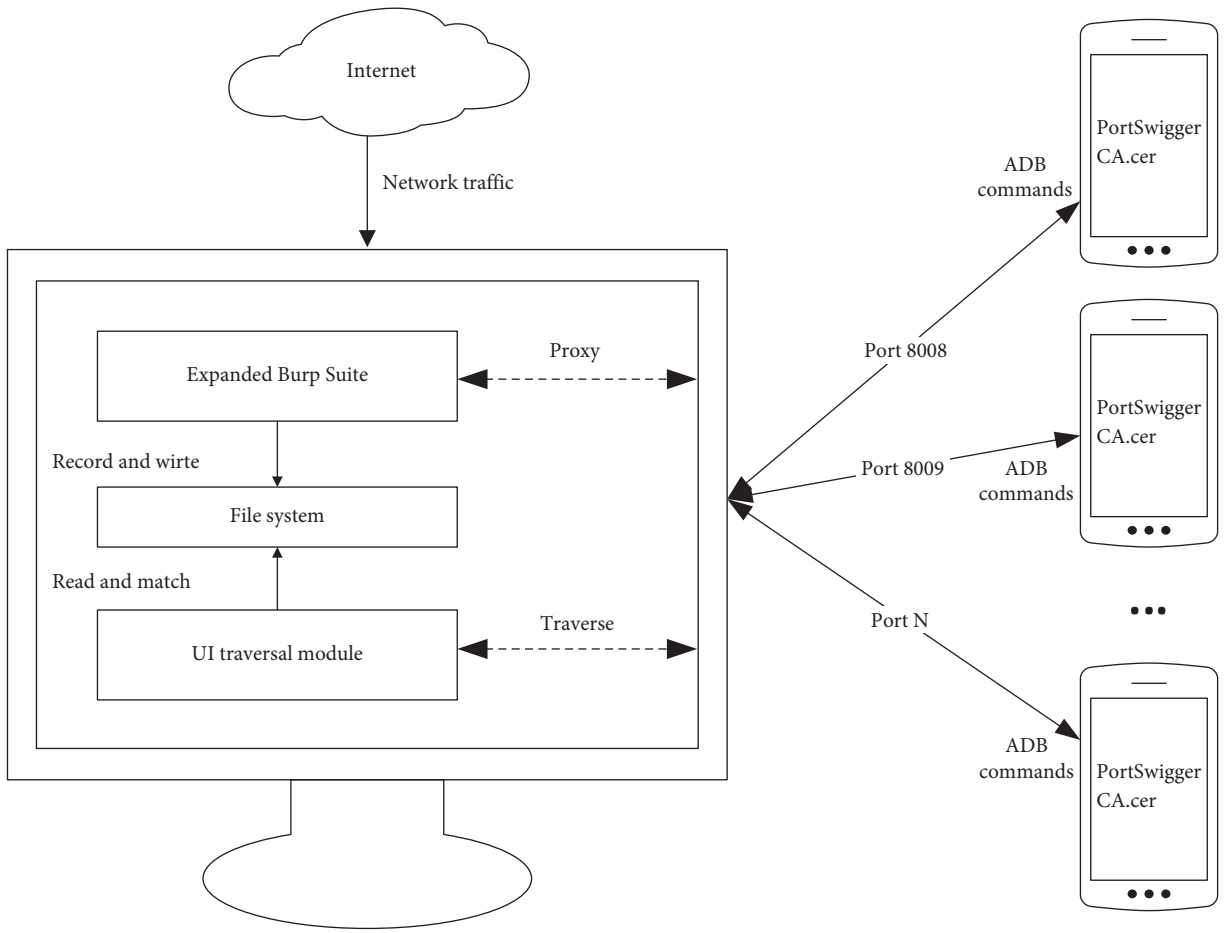

Figure 3: The implementation of the MITM Attack Test Module.

between the client and the server, you can intercept, view, and modify the raw data flow in both directions. The problem that the HTTPS connection cannot be grabbed may occur during the process of capturing the packet. We need to install the Burp Suite certificate *PortSwigger CA.cer* on the mobile phone and authorize it as a trusted credential. In order to verify whether the application has an SSL security vulnerability, it is necessary to determine whether it can resist the man-in-the-middle attack. So we need to grab HTTPS network requests from the target application.

Our UI traversal strategy automatically drives the traversal of the target application to perform its functions after the target application is deployed to intercept and crawl HTTPS network requests. When the application does not perform certificate and hostname verification for the self-signed certificate, the contents of the HTTPS request will be seen in the "HTTP history" window of the Burp Suite tool. The agent can obtain the results returned from the server side too.

5.4.3. Return Attack Results and Integration with the UI Traversal Module. To return the results of the Burp Suite to our SSLDetecter system in real time and to guide the UI strategy whether it needs to continue traversing, the Burp Suite's *Extender* module needs to be extended to complete the following functions:

(1) Successfully determine and filter whether the request is HTTPS

(2) Determine whether the HTTPS request is successful

(3) Save the record and inform the UI Traversal Module

Based on the extension API provided by Burp Suite, we have designed the *SSLlistener.jar* plug-in. The SSLlistener.jar plug-in implements the IBurpExtender and IProxyListener interfaces and calls the registerProxyListener() function to register the agent listener. Burp Suite triggers this agent listener when it processes a network request or network response. In the agent listener processProxyMessage() function, the getHttpService() function is called to retrieve specific information about the intercepted network request or response, such as message type, host name, protocol, and port.

During server authentication, the "middle man" (the extended Burp Suite in our SSLDetecter) intercepts the server certificate and sends the forged certificate to the client. Due to the SSL security vulnerability on the client side, the "middle man" is mistaken for the server without valid validation of the forged certificate. The "middle man" successfully obtains confidential information such as negotiation key and encryption message in the communication process and can even tamper with the transmitted data. The status code 200 indicates that the

server has successfully processed the request. Typically, this means that the server is providing the requested content. So, by implementing the filter conditions including the network message protocol is HTTPS request and the network request status code is 200, the MITM attack test module judges that the man-in-the-middle attack is successful.

Then the function writeToFile() is called to write this record which contains the information of the port, the smartphone serial number, the package name of the target application, and so on to the specific file. At the same time, the file listener thread opened by the UI traversal module captures an event of a specific port file modification. The UI traversal module reads the latest modified content of the file synchronously and matches the host name and the application package name running on the test smartphone of the corresponding port with the obtained information. This will determine which application on the test smartphone's HTTPS request is successfully executed by a MITM attack. In addition, if the network request is sent from the application's own code instead of the third-party library, it can indicate that the application has an SSL security vulnerability. The UI traversal module will record the results and stop exploring the application.

6. Evaluation

Here, we mainly describe the specific experimental process of evaluating our SSLDetecter system. [25] gives us great inspiration on how to describe the whole experimental process. Our goal is to evaluate the effectiveness of our system and compare it with existing systems to draw conclusions.

6.1. Research Questions. This section introduces the research questions to be answered in our study.

RQ_1: in our traversal strategy, what is threshold of the interface state similarity to determine whether two interfaces are the same?

RQ_2: what is the coverage rate of our system to automatically traverse Android apps and does it perform better than Google's official automated testing tool, Monkey [26]?

RQ_3: compared with the existing tool SMV-HUNTER, how effective is our system to detect SSL vulnerabilities in Android applications?

RQ_4: which markets and categories of applications might be more vulnerable to SSL security vulnerabilities?

6.2. Objects of the Experiment. The objects of our experiment are the real-world Android applications. No matter what category the application belongs to or what functions it has (our static analysis can preliminarily screen out the applications without networking functions), it usually can be tested by our SSLDetecter system. The applications include two aspects: one part is used to evaluate the effect of automatic traversal of system. A relatively large number of applications are used to evaluate the effectiveness of SSL vulnerability detection. We have crawled a huge number of applications from the mainstream application markets in China and randomly selected thousands of applications for analysis. These applications come from multiple application markets and multiple application categories, that is, we use real-world applications for analysis, rather than our own development. These applications can represent real Android applications.

We only consider those applications that have a user interface, and very few that do not have a front-end interface are outside the scope of our experiment.

6.3. Metrics Definition. Here, we define the metrics that we use to answer our research questions.

(1) Traversal Coverage (activity coverage) reflects the degree to which our system automates the execution of Android applications. Activity represents the visual interface of Android application. We record the number of activities traversed by the test application and then compare it with the total number of activities in the application. Although there are more accurate coverage calculation methods [27, 28], the core goal of our system is to use automated traversal to complete the dynamic detection of applied SSL security vulnerabilities. Therefore, we choose a relatively simple calculation method to express, as shown in the following formula:

$$\mathrm{TC}\% = \frac{\text{number of activities traversed}}{\text{total number of activities in the app}} * 100. \tag{3}$$

(2) Interface State Similarity is used to indicate how similar the two interface states are (Section 4.3).

(3) Average Analysis Time reflects the time cost of the system analyzing the target applications. The shorter the analysis time, the higher the efficiency of the system, and it is more conducive to large-scale application analysis. The time cost of static analysis and the time cost of dynamic analysis can be calculated separately. It is shown as

$$\mathrm{AAT} = \frac{\text{total time to anlayze the apps}}{\text{total number of apps analyzed}}. \tag{4}$$

(4) Detection Rate refers to the percentage of positive results in the total number of the experiment. For the same target data set, the more the positive results are detected (i.e., the number of applications that do have SSL vulnerabilities detected in our experiments), the more accurate and effective the system will be to detect the SSL vulnerabilities in Android applications. It is shown in the following formula:

$$\mathrm{DR}\% = \frac{\text{number of apps detected}}{\text{total number of apps}} * 100. \tag{5}$$

6.4. Experimental Procedure. In order to answer the research questions raised, we mainly conduct the following 4 experimental processes. These steps mainly include collecting applications, prescreening applications through static analysis, automating traversal of applications and conducting MITM attacks, and collecting and analyzing of experimental data.

In the *collecting applications* step, we use the crawler technology to collect thousands of real-world Android applications from mainstream application markets in China (such as Xiaomi, Baidu market, and OPPO). These applications vary in size and type, which can better reflect the application distribution in the application markets. Due to the large number of applications, it is not necessary to analyze all of them in a time-consuming manner. We randomly select applications for SSL vulnerability detection. In addition, in order to analyze the traversal coverage of our system, we randomly selected applications to evaluate the traversal coverage of the system in combination with manual analysis.

In the *prescreening applications* step, we screen the randomly selected real-world Android applications in accordance with the static code analysis techniques described in Section 5.2. After the successful screening, not only the suspicious application is obtained but also the information of one or more of the four SSL vulnerability categories that may exist in the application. For applications that cannot be statically analyzed, we proceed directly to the subsequent dynamic traversal analysis.

In the *automating traversal of applications and conducting MITM attacks* step, we use the automated traversal techniques described in Section 5.3 to analyze the applications filtered by the previous step. Our system is deployed in Linux/Unix environments (such as Ubuntu and Mac OS).The experimental software and hardware environment information is shown in Table 3. Our system has connected two Android device simulators for testing and analysis.

While driving applications traversal execution, we conduct MITM attacks synchronously. We deploy the MITM attack test module as described in Section 5.4. The MITM attack test module communicates with the UI traversal module through a specific file. According to the success of the attack, it can be judged whether the application does have an SSL security vulnerability.

In the *collecting and analysing of experimental data* step, we collect the information about the applications after testing the MITM attack and record the number of applications remaining after each analysis and the number of applications successfully attacked by MITM. The information is collected not only during the operation of the SSLDetecter system but also under the comparison system SMV-HUNTER. Furthermore, we also record the total number of Activities minus third-party libraries of the randomly selected applications, as well as the number of Activities that SSLDetecter and Monkey traversed during the same time.

6.5. Results and Analysis

6.5.1. Confirm the Interface State Similarity Threshold. The setting of the interface state similarity threshold affects the number of GUI nodes in the GUI traversal model. To balance the efficiency and accuracy and test the influence of different similarity thresholds on the number of GUI nodes and *Activity* components traversed, we randomly select 6 popular applications of different categories from the application markets for testing. The specific information of the six applications is shown in Table 4, including the application name, the size of the application, the collection of Activity components extracted from the *AndroidManifest.xml* file, and the total number of activities after removing the third-party library.

In Figure 4, when the similarity threshold is gradually increased from 0.5 to 0.9, the total number of GUI nodes traversed in the six applications tends to increase. It can be seen from the four applications of "Douguo," "Straight flush," "Tencent News," and "JMEI" that when the threshold value changes from 0.5 to 0.7, the interface state changes slowly and the number of GUI nodes formed is not very different. After the threshold value of 0.7, the interface state is further divided, and the number of GUI nodes increased obviously. Figure 5 shows the variation trend of the total number of *Activities* traversed in the six apps when the similarity threshold gradually increases from 0.5 to 0.9. As can be seen from the figure, the change in the number of *Activities* is not as obvious as the change in the number of GUI nodes and tends to balance in the end.

After further analysis of the two applications of "CareOS" and "Zhihu Daily" which have little change in the number of GUI nodes, the main reasons are found as follows. The total number of widgets on each interface for the "CareOS" application is small, and the traversal saturation has been reached when the threshold is set to 0.5. The "Zhihu Daily" application belongs to a single-function information application, and each category of soft text list is mainly displayed by the *ListView* widget. Since the child widgets of the same structure in the interface widget tree are effectively cropped, the number of nodes traversed varies little with the similarity threshold. At the same time, it is found that the successful login of the application has a certain influence on the number of generated GUI nodes. For example, when the similarity threshold is set to 0.9, if the login is successful, the traversed GUI node is as high as 103; if the login fails, the number of GUI nodes traversed is 70.

Combined with the results shown in Figures 4 and 5, the threshold value of the interface state similarity is set to 0.8 in our SSLDetecter system from the aspects of accuracy and efficiency. This is the answer to the first research question RQ_1.

6.5.2. Average Traversal Coverage of SSLDetecter. We compare the SSLDetecter system with Google's official automated testing tool Monkey. Monkey drives the automation of applications based on a sequence of random events, while our system is based on application widgets. We set Monkey as the default configuration and set the interval between two random test events to 200 milliseconds. In the experiment, we have found that the test results of Monkey are different each time, and finally, we take 5 tests to average. However, when the network state remains unchanged, the

Table 3: Experimental hardware and software environment information.

Hardware environment	CPU	2.7 GHz Intel core i5
	RAM	8 GB
Software environment	Operating system	OS X Yosemite 10.10.5 64 bit
	Android system	Android 4.4 and above
	Appium	Appium 1.5.0
	Burp Suite	Version 1.7.10

Table 4: Basic information of the chosen 6 Android apps.

App name	App size (M)	Total number of Activies	Total number of activities without third-party library
CareOS	15.4	57	47
Straight flush	28.1	39	20
Tencent News	24.9	153	138
JMEI	38.5	226	208
Douguo	8.0	160	156
Zhihu daily	5.3	24	20

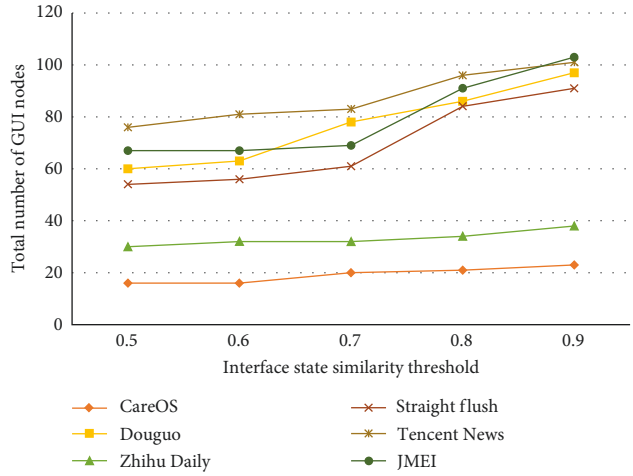

Figure 4: The relationship between the threshold and the number of GUI nodes traversed.

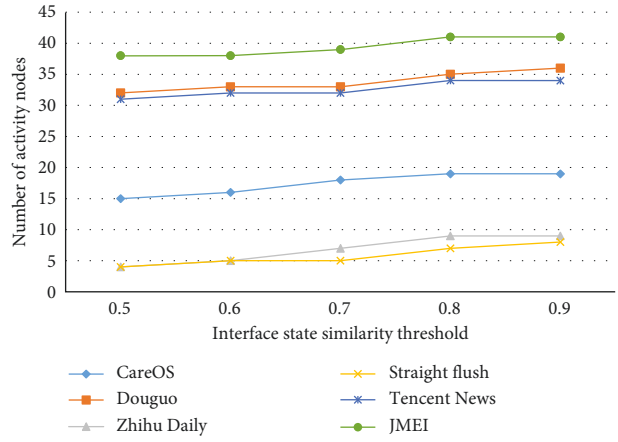

Figure 5: The relationship between the threshold and the number of Activity nodes traversed.

situation in which our SSLDetecter runs multiple times is basically the same. In addition to the six applications listed in Section 6.5.1, we have added four applications for a total of 10 applications for traversal coverage assessment. Each target application tests for 4 minutes on both SSLDetecter and Monkey. The results are shown in Table 5.

The results show that the average Activity Coverage of Monkey is 21.7%. And the Activity Coverage rate of our

Table 5: Activity coverage results of SSLDetecter and Monkey.

App package name	Total number of activities without third-party library	SSLDetecter		Monkey	
		Total number of activities traversed	Activity coverage (%)	Total number of activities traversed	Activity coverage (%)
com.cappu.careoslauncher	47	19	40.4	7	14.9
com.hexin.plat.android	20	7	35.0	6	30.0
com.tencent.news	138	34	24.6	10	7.2
com.jm.android.jumei	208	41	19.7	19	9.1
com.douguo.recipe	156	35	22.4	14	9.0
com.zhihu.daily.android	20	9	45.0	6	30.0
com.wondertek.paper	12	7	58.3	5	41.7
com.chaozh.iReaderFree	120	21	17.5	11	9.2
ru.zdevs.zarchiver.pro	18	10	55.6	9	50.0
com.boly.wxmultopen	32	11	34.4	5	15.6
AVG (%)		35.3		21.7	

SSLDetecter ranges from 17.5% to 58.3%, with an average coverage rate of 35.3%. The Activity Coverage of our system is much higher than that of Monkey. In practice, SSLDetecter traverses more application UIs than Monkey. This is the answer to the second research question RQ_2.

In general, as the number of Activities in the application increases, the Activity Coverage gradually decreases in both SSLDetecter and Monkey. The main reason is that the number of activities in the application is positively correlated with the functional complexity of the application. The more complex the function is, the more complex the GUI node jump will be, which will lead to the reduction of coverage. There are many reasons why Monkey's traversal coverage is lower than our system. Firstly, Monkey is easy to click on the outside area of the application, causing exceptions such as jumping out of the application and being unable to execute. While operations of our system is to click on the application's widgets, all operations are controllable, alleviating a lot of jump failures. Secondly, SSLDetecter can automatically complete some simple user input events based on the UI type and the preconfigured input library. Monkey does not have the support of simple user input, and the research [28] has shown that automatic inputs can help increase traversal coverage. In addition, sometimes, on an Activity of an application, there are many similar widgets that can display a lot of similar content. Monkey tends to spend too much time in this Activity because it does not have traversal strategies, while our SSLDetecter system has corresponding traversal strategies that can handle these specific situations very well. Table 6 summarizes the different characteristics of our SSLDetecter system and Monkey.

A study [29] shows that if we allow users to manually exhaust the traversal of the GUI interface of the application, the activity coverage achieved by the artificial traversal method is 30.08% on average. The experimental results prove that the coverage rate of our automatic traversal algorithm is also better than that of manual traversal. Furthermore, our method is applicable to packed applications, and time is within acceptable limits without being too lengthy.

Overall, our SSLDetecter system has much higher traversal coverage than the Monkey. Our traversal strategy is optimized for MITM attack scenarios, with higher coverage, and is more suitable for driving application autoexecution and completing dynamic verification of SSL security vulnerabilities.

6.5.3. Comparison between SSLDetecter and SMV-HUNTER. In order to verify the feasibility and practicability of our SSLDetecter system, we use crawler technology to collect popular applications of various categories in the 9 major application markets, with a total number of 2456. The number distribution of each category in 2456 applications is shown in Figure 6.

Our SSLDetecter system and the existing SMV-HUNTER system are, respectively, used to test 2456 applications for comparison. As can be seen from Figure 7, compared with the SMV-HUNTER system, the SSLDetecter system improves the detection efficiency by about 38%. The total time of static analysis of SMV-HUNTER is 44.8 hours. It statically analyzes 2,456 applications, with an average static analysis time of 65.7 seconds per application. After static analysis, 350 applications are left for dynamic analysis. And the total time of dynamic analysis using two simulators is 54.2 hours. The average time of dynamic analysis is 558 seconds.

While for our SSLDetecter system, the total time of static analysis is 33.7 hours for 2456 applications, with an average static analysis time of 49.4 seconds per application. After static analysis, 773 applications are left for dynamic analysis. The total time of dynamic analysis using two simulators is 72.1 hours and average time of dynamic analysis is 336 seconds. In summary, the average analysis time for each application of our SSLDetecter system is 385.4 seconds, while SMV-HUNTER is 623.7 seconds.

As is shown in Figures 8 and 9, SSLDetecter detects more applications with SSL vulnerabilities, and the average detection rate increases by 6.39 percentage points than SMV-HUNTER. It should be noted that the SMV-HUNTER currently only supports the detection of only the vulnerability type of *trusting all hosts*, and the number of such vulnerabilities detected is 267. Our SSLDetecter system detects a total of 424 applications with SSL security vulnerabilities (not just trusting all host vulnerabilities).

Table 6: SSLDetecter vs. Monkey.

Features	SSLDetecter	Monkey
Higher traversal coverage	✓	×
Including static analysis	✓	×
Automated simple input	✓	×
Judge UI types	✓	×
Events based on the widgets	✓	×
Stability of traversal effect	✓	×

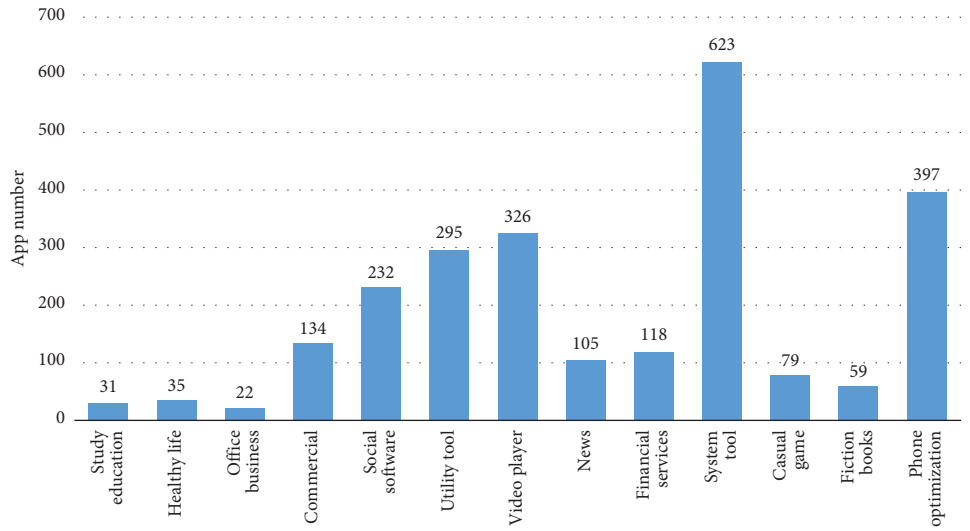

Figure 6: A histogram of the number distribution of each category in the apk dataset.

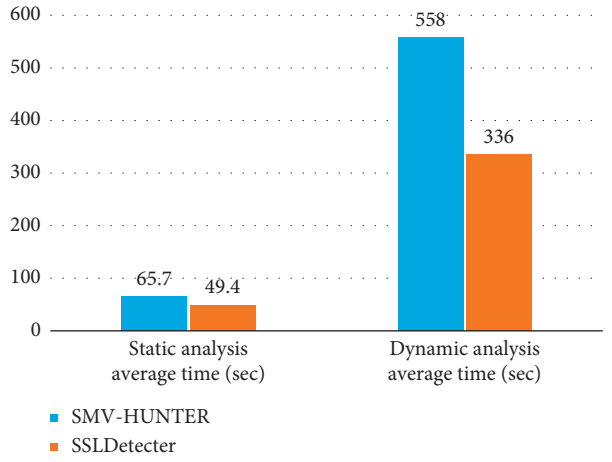

Figure 7: Comparison of detection time between SMV-HUNTER and SSLDetecter.

To further analyze the results in detail, the test results of SSLDetecter are divided into three categories: trusting all host-type vulnerabilities, other types of SSL vulnerabilities, and SSL security vulnerabilities of packed applications. Figure 8 describes the results of SSLDetecter system in detail. For the vulnerability of trusting all hosts, SSLDetecter detected 258 apps which are consistent with the SMV-HUNTER detection results. And the other 39 applications that are not detected by the SMV-HUNTER system due to the failure to acquire the target components. Further analysis of the 9 applications not detected by SSLDetecter shows that these 9 test applications communicate with other software (such as the system's built-in browser, camera, WeChat, or QQ) during the running process, which caused the test applications to jump out of the interface more than three times and thus interrupted the traversal test.

In addition, the details of each step of the two detection systems are discussed below. Table 7 describes the

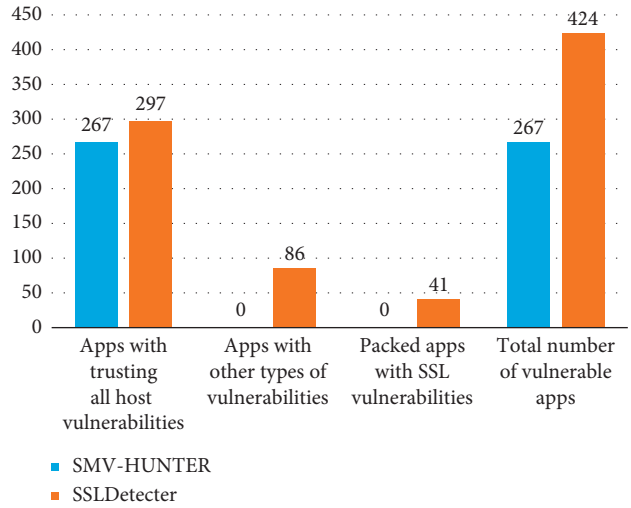

Figure 8: The distribution of vulnerable applications detected by SMV-HUNTER and SSLDetecter.

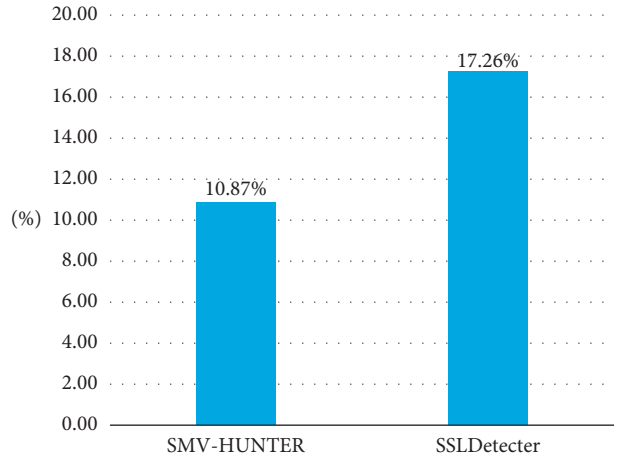

Figure 9: Detection rates of SMV-HUNTER and SSLDetecter.

number of applications after each step of SMV-HUNTER analysis. SMV-HUNTER spends a lot of time in dynamic analysis. The reason is that for the collection of vulnerability target components built by static analysis, SMV-HUNTER needs to exhaust every component in the start-up set and triggers events for every clickable widget and input widget on the interface. We analyze the 671 applications with the target component set, and the number of applications with more than 50 components to be started account for 25% of the total number of dynamic tests. Each of these applications is tested for about 30 minutes. In the application package named COM.-IMMOMO.MOMO, the number of target components that may have SSL security vulnerability is as high as 540, which takes several hours to test.

Compared with SMV-HUNTER, SSLDetecter system can detect more applications with SSL vulnerabilities, which is mainly attributed to the fact that our system does not rely on static analysis to generate a set of target components. The application number results after each step of SSLDetecter are shown in Table 8. SSLDetecter does not build a set of target Activity components based on static analysis, which can effectively solve the detection problem of applications against static analysis (such as packed applications).

Based on these experimental results, we answered the research question RQ_3 as follows: our SSLDetecter system has improved the average detection rate by 6.39 percentage points and detection efficiency by 38% compared with SMV-HUNTER system, making it more suitable for SSL security vulnerability detection.

6.5.4. Further Analysis of the Distribution of Applications with SSL Vulnerabilities. The 424 applications with SSL security vulnerabilities are statistically analyzed from their categories and application markets. The results are shown in Figures 10 and 11. It can be seen from Figure 10 that SSL security vulnerabilities mainly exist in three categories: commercial, financial services and utility tools, whose applications are closely related to users' privacy. For example, the weather

Table 7: Number of applications after analysis of each step of SMV-HUNTER.

System	Number of successful decompiled applications	Number of applications with SSL misuse APIs obtained through static analysis	Number of applications that can build a collection of vulnerability target components	Number of applications with input	Number of applications where SSL security vulnerabilities are dynamically verified
SMV-HUNTER	2127	1006	671	220	267

Table 8: Number of applications after analysis of each step of SSLDetecter.

System	Preliminary filter				Dynamically verify the presence of SSL security vulnerabilities
	Application with network permissions	Applications that may have vulnerabilities of trusting all hosts	Applications that may have other types of SSL vulnerabilities	Packed apps	Total number
SSLDetecter	1708	1006	210	329	424

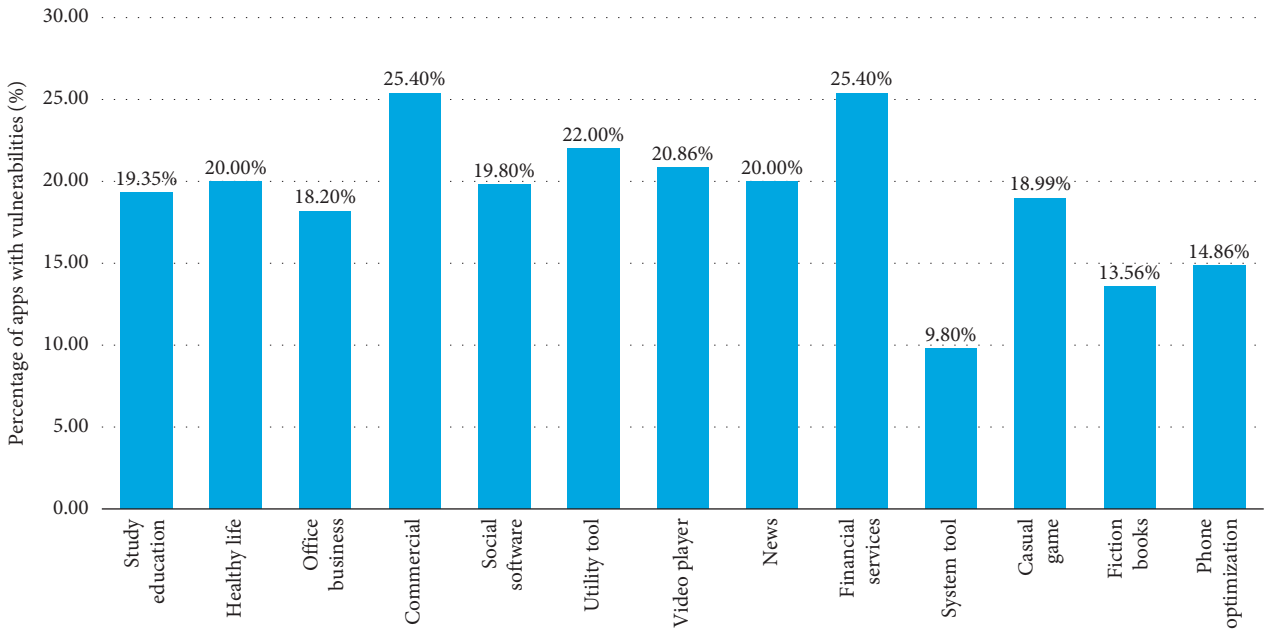

Figure 10: Percentage of apps with vulnerabilities for each category.

forecast application in the utility tool needs to obtain the user's geographical location information, the WeChat application in the social software needs to obtain the user's mobile phone number, commercial and financial services involve the user's bank account, Alipay account, and other confidential information. When users use insecure SSL protocol for network communication, their confidential information is at risk of being stolen, so it is necessary to carry out strict security audit for this type of applications. In the larger markets such as Myapp, 360 mobile assistant and HiMarket, the application security can be guaranteed to some extent due to the certain review mechanism before the application is put on the shelf. But there are still a few applications with SSL security vulnerabilities. Therefore, it is of great significance to propose an effective SSL vulnerability detection scheme for various markets with different levels of participation in self-audit. There are a relatively large number of applications with SSL security vulnerabilities in the markets named Anzhi, Mumayi and 25PP. Applications in the commercial and financial services categories are more vulnerable to SSL security vulnerabilities. This is the answer to the last research question RQ_4.

6.6. Threats to Validity. Here are the threats that may affect the validity of our system.

6.6.1. Internal Validity. In our experiment, we have to manually analyze some screenshot samples of applications to formulate the UI classification rules of Android applications.

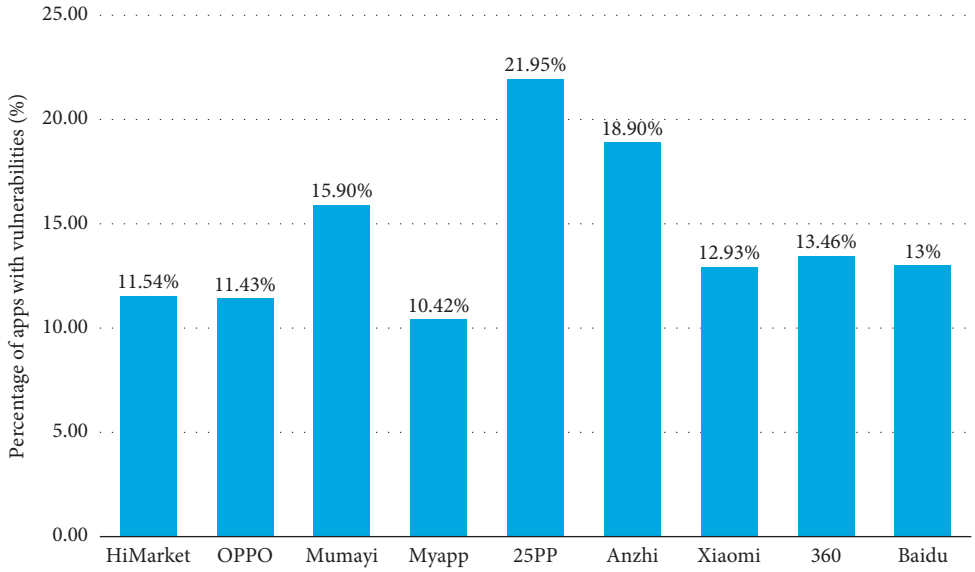

Figure 11: Percentage of apps with vulnerabilities for each market.

Therefore, the distributive relationship between objects and subjects may be a threat to the internal validity. This may lead to incomplete classification rules and failure of interface widget tree optimization strategy in some cases. However, the imperfection of this optimization strategy does not affect our system to effectively analyze Android application SSL security vulnerabilities. Because our system is mainly to trigger the network communication behaviour of the application, it is more time-consuming but still accurate without optimization. To mitigate this threat, we randomly select the interfaces of multiple categories of Android applications for analysis, and develop classification rules that fully take into account a variety of situations.

Another threat is that the characteristics of SSL vulnerability type are obtained by static analysis. Static analysis can initially help us determine that the application may have SSL security vulnerabilities, but is not always accurate. The successful implementation of MITM attack only verifies that the application does have SSL security vulnerability, but does not verify the specific type of SSL vulnerability. However this does not affect the accuracy of our SSLDetecter system for detecting SSL security vulnerabilities.

In addition, our traversal coverage calculation metrics are not sufficiently granular. The coverage calculated by this formula may not be a very accurate reflection of the coverage of our system compared with other systems. A better approach is to refine to a smaller level of widgets in the Activity interface. However, the error in the calculation of coverage does not directly affect the accuracy of detecting SSL vulnerabilities, which is the ultimate goal of our system.

6.6.2. External Validity. A possible threat to the external validity is the choice of the testing Android applications. Specific applications that have been artificially selected do not provide a good indication of the situation. To alleviate this threat, we randomly selected a large number of real-world application samples for evaluation. These samples cover a wide variety of categories and are widely distributed in size. The more comprehensive the object applications, the more truly it can reflect the detection effect of our system on the real world application.

7. Conclusions and Future Work

We propose a method of automatic detection of SSL security vulnerabilities in Android applications. The application is dynamically driven to run automatically through UI traversal technology. And the man-in-the-middle attack test is conducted in real time during the application running process to determine whether it can resist the MITM attack. In order to trigger as many different logic network request events as possible and return the attack results in real time, we propose a new traversal strategy by optimizing the widget tree according to the interface type and calculating the interface state similarity based on the widget path set. Based on the network port monitoring technology, we extend the function of the man-in-the-middle agent to achieve the purpose of rapid and accurate detection. We design and implement the Android application SSL security vulnerability automatic detection system SSLDetecter and verified the effectiveness and availability of the system through experiments. Compared with the existing system SMV-HUNTER, the time efficiency of our system increases by about 38% and the average detection rate increases by 6.39 percentage points, with more types of vulnerabilities detected. Our method can effectively carry out security analysis for packed applications which can resist static analysis to some extent.

In the future, more attack test scenarios can be further added to dynamically verify SSL vulnerability types rather than just judging the existence of SSL vulnerabilities based on the success of the MITM attack. In addition, we extract

interface widget information relying on the Android system tool UIAtuomator and the open source tool Appium. Some widget types, such as HTML5 developed widgets or user-defined widgets, may not be recognized for analysis. We plan to improve the method to obtain more interface information more accurately and improve the effect of traversal. Furthermore, our widget tree traversal strategies are based on manual analysis and may not be comprehensive. More judgments can be added to improve the optimization strategies.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under grants 2016YFB0800402 and 2016QY01W0202, National Natural Science Foundation of China under grants U1836204, 61572221, 61433006, U1401258, 61572222, and 61502185, major projects of the National Social Science Foundation under grant 16ZDA092, and Guangxi High level innovation Team in Higher Education Institutions Innovation Team of ASEAN Digital Cloud Big Data Security and Mining Technology.

References

[1] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source attribution of cryptographic API misuse in android applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 133–146, Incheon, South Korea, June 2018.

[2] M. Stevens, A. Sotirov, J. Appelbaum et al., "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *Proceedings of the Annual International Cryptology Conference*, pp. 55–69, Santa Barbara, CA, USA, August 2009.

[3] R. Biddle, P. C. Van Oorschot, A. S. Patrick, J. Sobey, and T. Whalen, "Browser interfaces and extended validation SSL certificates: an empirical study," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pp. 19–30, Chicago, IL, USA, November 2009.

[4] http://appium.io/.

[5] J. Wang, "The prediction of serial number in OpenSSL's X. 509 certificate," *Security and Communication Networks*, vol. 2019, Article ID 6013846., 11 pages, 2019.

[6] X. Wei and M. Wolf, "A survey on HTTPS implementation by Android apps: issues and countermeasures," *Applied Computing and Informatics*, vol. 13, no. 2, pp. 101–117, 2017.

[7] S. Fahl, M. Harbach, T. Muders et al., "Why eve and mallory love android: an analysis of android SSL (in) security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 50–61, Raleigh, NC, USA, October 2012.

[8] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 49–60, Berlin, Germany, November 2013.

[9] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and S. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 38–49, Raleigh, NC, USA, October 2012.

[10] L. Onwuzurike and E. De Cristofaro, "Danger is my middle name: experimenting with SSL vulnerabilities in Android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 84–90, New York, NY, USA, June 2015.

[11] M. Conti, N. Dragoni, and S. Gottardo, "Mithys: mind the hand you shake-protecting mobile devices from ssl usage vulnerabilities," in *Proceedings of the International Workshop on Security and Trust Management*, pp. 65–81, Egham, UK, September2013.

[12] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson, "An analysis of the privacy and security risks of android VPN permission-enabled apps," in *Proceedings of 16th ACM Conference on Internet Measurement Conference*, pp. 349–364, Santa Monica, CA, USA, November 2016.

[13] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-hunter: large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, August 2014.

[14] F. Gagnon, F. MA, M. A. Fortier, S. Desloges, J. Ouellet, and C. Boileau, "AndroSSL: a platform to test android applications connection security," in *Proceedings of the International Symposium on Foundations and Practice of Security*, pp. 294–302, Québec City, Canada, October 2015.

[15] E. Munivel and A. Kannammal, "New authentication scheme to secure against the phishing attack in the mobile Cloud computing," *Security and Communication Networks*, vol. 2019, Article ID 5141395, 11 pages, 2019.

[16] L. Li, D. Li, T. F. Bissyande et al., "Understanding android app piggybacking: a systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[17] D. Maier, M. Protsenko, and T. Müller, "A game of droid and mouse: the threat of split-personality malware on Android," *Computers & Security*, vol. 54, pp. 2–15, 2015.

[18] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[19] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "AppSpear: automating the hidden-code extraction and reassembling of packed android malware," *Journal of Systems and Software*, vol. 140, pp. 3–16, 2018.

[20] M. Protsenko and T. Müller, "Protecting android apps against reverse engineering by the use of the native code," in *Trust, Privacy and Security in Digital Business*, pp. 99–110, Springer, Berlin, Germany, 2015.

[21] J. Li, "Research on automatic detection of SSL security vulnerability in network communications of android applications," Master's thesis, Huazhong University of Science and Technology, Wuhan, China, 2017.

[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: automated model-based

testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[23] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 238–249, Singapore, September 2016.

[24] https://github.com/androguard/androguard.

[25] C. Wohlin, P. Runeson, M. Höst et al., *Experimentation in Software Engineering*, Springer Science & Business Media, Berlin, Germany, 2012.

[26] https://developer.android.com/studio/test/monkey.html.

[27] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, "A general framework for comparing automatic testing techniques of Android mobile apps," *Journal of Systems and Software*, vol. 125, pp. 322–343, 2017.

[28] D. Amalfitano, V. Riccio, N. Amatucci et al., "Combining automated GUI exploration of android apps with capture and replay through machine learning," *Information and Software Technology*, vol. 105, pp. 95–116, 2019.

[29] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641–660, 2013.