The Journal of Systems and Software 167 (2020) 110609

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Identifying vulnerabilities of SSL/TLS certificate verification in Android apps with static and dynamic analysis

Yingjie Wang[a], Guangquan Xu[b,c], Xing Liu[a], Weixuan Mao[d], Chengxiang Si[d], Witold Pedrycz[f], Wei Wang[a,e,*]

[a] Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, China
[b] Big Data School, Qingdao Huanghai University, China
[c] Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, 300350 Tianjin, China
[d] National Computer Network Emergency Response Technical Team / Coordination Center of China, China
[e] Division of Computer, Electrical and Mathematical Sciences & Engineering (CEMSE), King Abdullah University of Science and Technology (KAUST), Thuwal 23955-6900, Saudi Arabia
[f] Department of Electrical and Computer Engineering, University of Alberta, Canada

ARTICLE INFO

Article history:
Received 15 September 2019
Revised 23 March 2020
Accepted 17 April 2020
Available online 22 April 2020

Keywords:
Android security
Dynamic analysis
MITM
SSL/TLS
Vulnerability detection
Static analysis

ABSTRACT

Many Android developers fail to properly implement SSL/TLS during the development of an app, which may result in Man-In-The-Middle (MITM) attacks or phishing attacks. In this work, we design and implement a tool called DCDroid to detect these vulnerabilities with the combination of static and dynamic analysis. In static analysis, we focus on four types of vulnerable schema and locate the potential vulnerable code snippets in apps. In dynamic analysis, we prioritize the triggering of User Interface (UI) components based on the results obtained with static analysis to confirm the misuse of SSL/TLS. With DCDroid we analyze 2213 apps from Google Play and 360app. The experimental results show that 457 (20.65%) apps contain potential vulnerable code. We run apps with DCDroid on two Android smart phones and confirm that 245 (11.07%) of 2213 apps are truly vulnerable to MITM and phishing attacks. We propose several strategies to reduce the number of crashes and shorten the execution time in dynamic analysis. Comparing with our previous work, DCDroid decreases 57.18% of the number of apps' crash and 32.47% of the execution time on average. It also outperforms other three tools, namely, AndroBugs, kingkong and appscan, in terms of detection accuracy.

1. Introduction

Smartphones are now widely used in people's daily life. Android has become the most popular mobile operating systems (OS), accounting for around 74.13% of the smart phone's market in the world (Mobile Operating System, 2020). According to Google's statistics, there are over 2.9 million Android apps that are downloaded for over hundreds of billions times from Google Play as of December 2019 (Google Play Store, 2020). These apps cover a range from life, entertainment to finance or business. In order to secure the transmission of sensitive data for avoiding data leakage or attacks, many apps use HTTPS (HTTP over Security Socket Layer (SSL)/Transport Layer Security (TLS)) protocol to transmit sensitive data (Google, 2020). Unfortunately, improper implementation of SSL/TLS certificates can lead to Man-In-The-Middle (MITM) attacks (Clark and van Oorschot, 2013) and phishing attacks (He et al., 2015). In the process of MITM attack or phishing attack, attackers impersonate the server to intercept and even modify app traffic to obtain sensitive data. In general, an attacker is not able to decrypt network traffic. However, if the client blindly trusts any certificate without checking the signatures, or does not verify the host name, or ignores the verification error prompts, the attacker can impersonate the server to gain the trust of the client using a fake certificate, and then decrypt the traffic to obtain sensitive data during the attack.

Existing efforts have been made on the detection of malicious apps. Our previous work detected malicious apps (Wang et al., 2018a; Su et al., 2019; Wang et al., 2019a; Xie et al., 2019), analyzed privacy leakage (Liu et al., 2018b; 2019) and detected (Wang and Battiti, 2006; Wang et al., 2014a; 2010; 2018c; Xu et al., 2020; Wang et al., 2006; 2015; Feng et al., 2009; Wang et al., 2008; 2020a; Guan et al., 2009; Wang et al., 2009b; 2004a; 2009a) or prevented (Xu et al., 2019; Li et al., 2019b; Wang et al., 2004b)

* Corresponding author.
E-mail addresses: yingjie_w@bjtu.edu.cn (Y. Wang), losin@tju.edu.cn (G. Xu), xingliu@bjtu.edu.cn (X. Liu), maoweixuan@cert.org.cn (W. Mao), pedrycz@ee.ualberta.ca (W. Pedrycz), wangwei1@bjtu.edu.cn (W. Wang).

https://doi.org/10.1016/j.jss.2020.110609
0164-1212/

intrusions with different methods. There also exist related work (Fahl et al., 2012; Brubaker et al., 2014; Sounthiraraj et al., 2014; Liu et al., 2018c) on Android MITM attacks caused by improper implementation of SSL/TLS in Android's apps. However, these methods require manual analysis to confirm vulnerabilities. Most of the related work aimed at the detection of malicious apps, rather than the detection of vulnerabilities. They often focus on a specific kind of Android apps, and the vulnerabilities detected are not comprehensive enough. In addition, some existing work started Activity directly in the process of detection, which may lead to apps' crash.

In order to solve these problems, in our previous work (Wang et al., 2019c), we propose an automatic method to detect apps with SSL/TLS certificate verification vulnerabilities. It contains the following steps: definition of the vulnerable code, detecting the vulnerability of Smali code and running app dynamically under the MITM attack with fake certificates. Finally, by analyzing the traffic between an app and the server to confirm the vulnerability (that is, an app is confirmed to have this kind of vulnerability when it is successfully attacked). However, we find that some apps crashed during the detect process, and some apps were run too long time. In addition, a number of false positives exist with our previous method. Through comprehensive analysis, we summarize the following reasons for the problems:

- Some Activities of an app cannot start directly. They must be run in a certain order from one Activity to another.
- Some apps have many similar UI elements, and these elements are different while the code logic is the same (such as some tabs).
- Some HTTPS connections of an app are mixed with others because some apps are run in the background (such as some system services).

In order to deal with these problems and detect the vulnerabilities stably, fast and accurately, we re-design our tool called DCDroid (Detecting vulnerable Certificates in Android apps) that contains static analysis, dynamic analysis and traffic analysis. In static detection, we define vulnerable code, disassemble an app to get Smali code and search the code to locate the vulnerable point. We then get the entry point by analyzing the invocation relationship of the method. In dynamic detection, the apps are run under the guide of static analysis. An activity with vulnerable code does not start directly so that the tool is more stable in the detection. Besides, duplication is reduced in execution phase with some strategies. Next we set up proxy servers to carry out MITM attacks. We develop an app to capture traffic on the smart phone so that we can get the pure traffic of apps and reduce the false positives. With these measures, we have achieved higher accuracy, stability and execution efficiency.

We make the following contributions:

- We develop an automated tool called DCDroid to detect SSL/TLS vulnerabilities with the combination of static and dynamic analysis. With DCDroid, we analyze 2213 apps and find that 457 apps are vulnerable through static analysis. After dynamic analysis we find that 245 apps are truly vulnerable. There are over 10% of apps that have vulnerabilities of SSL/TLS. We analyze the categories and version evolution of vulnerable apps and provide our suggestions to developers based on the detection results.
- We start executing the Activity of the vulnerable code from the Main Activity of an app instead of starting it directly. By dynamic execution, DCDroid is more stable than previous work with few crashes. Compared with starting Activity directly, the number of crashes decreased by 3.39 (57.18%) times per app on average.
- We reduce some meaningless execution during the detection processes. We detect vulnerable but similar views with some strategies, and the running time for each app decreases by 88 seconds (32.47%).
- We develop an app to capture traffic on Android smart phones to mark the relationships between traffic and apps. We find that the traffic is more reliable in the experiments and we identify 3 false positives with DCDroid.

The remainder of this paper is organized as follows. In Section 2, we introduce the background. We introduce the research statement and main challenges in Section 3. In Section 4, we present DCDroid based on the static analysis and the dynamic analysis. We describe the data sets and give our experimental results in Section 5. In Section 6, we discuss the limitations of DCDroid. We introduce related work in Section 7. Finally we conclude this paper in Section 8.

2. Background

In this section, we first introduce the applications of SSL/TLS on Android, then the Android UI, and finally the network and MITM of Android.

2.1. SSL/TLS and Android

SSL and its successor TLS protect the message from MITM attacking by encrypting network messages. To achieve this goal, it is important to obtain certificates containing public keys from the server. According to RFC 5280 (Internet X.509, 2020) documents, the client must verify the certificate to ensure that the certificate received is the server's certificate being connected to. Correct verification includes the following aspects:

- Each certificate in the certificate chain has not expired;
- Certificates or the root certificate in the certificate chain is signed by Certification Authority (CA) of clients;
- The domain name in the certificate matches with the domain name of the server being connected to.

Android OS provides a built-in digital certificate verification method, which is not vulnerable. However, it also allows developers to implement their own certificate verification method (Android Developer Trainingś, 2020; Elenkov, 2014; GnuTLS Transport Layer Security Library, 2020). The reasons that developers rewrite certificate verification methods include: using self-signed certificates, servers' root certificate is not in Android's CA list, correcting the unsafe implementation of some third-party libraries (Georgiev et al., 2012) and so on. However, in the process of implementation, vulnerable certificate verification methods are often introduced for various reasons, including (Fahl et al., 2012; Liu et al., 2018c):

- Trust all certificates with the *X509TrustManager* interface;
- Domain name is not checked by *HostnameVerifier*;
- Accept any domain name using the *setHostnameVerifier* (By using *ALLOW_ALL_HOSTNAME_VERIFIER*) method;
- Call *proceed()* method directly in *onReceivedSslError()* method to ignore certificate verification errors when a certificate verification error occurs in WebView component.

2.2. Android UI

Activity (Activities, 2020) is a visual interface used by Android to interact with users. An app may consist one or more Activities. The Activity used by the app is defined in Android's AndroidManifest.xml file. In particular, the Activity entered at the start of the app is called Main Activity. Activity manages Views with Windows. A View refers to editable components (such as text boxes), clickable components (such as buttons) and static components (such as labels). Service has no interface, and it will be executed in background. For example, Service can get data from the network or perform some computational tasks while users are dealing with other tasks. Intent is an object that holds the content of a message. It describes the operation that Activity wants to perform, and contains the data needed to start next Activity. It is used to jump to another Activity from this Activity.

We regard the interface as a directed graph, in which the nodes represent Activity or Service, the edges represent intent. Running all activities means the traversal of the graph starting from Main Activity nodes. By incorporating the UI into a graph, the automation algorithm of UI can be implemented more conveniently.

2.3. Android network and MITM

Each app has a unique process ID called PID. PID and IP are saved in a file during the network connection. Besides, Android OS provides an interface called VPNService, from which URL and IP can be obtained. The corresponding relationship between PID and apps' name can be obtained from Android OS, too.

In a MITM attack, the attacker is in the middle of client and server's communication. The attacker can intercept the client's message and send the intercepted message to the server. It can also intercept or modify the server's message and impersonate the server to communicate with the client. Before communicating, it can send a certificate containing its host name to the client. If the client does not verify the certificate or verify the certificate without checking the host name of the certificate (because the certificate of the middle-man may also be signed by Certificate Authority (CA)), the middle-man can constantly intercept, eavesdrop on and even modify the message.

3. Problem statement

In this section, we introduce the main challenges in this work.

3.1. How to define potential vulnerable code and trigger them

Because it is time-consuming to run apps dynamically, we first need to determine which apps contain vulnerable SSL/TLS implementation. We eliminate some apps by static detection and provide guidance for dynamic detection. We need to pre-define vulnerable code. If the selected SSL/TLS vulnerable code is not representative, it will result in false negatives. The definitions of vulnerable code are limited in previous work, and the coverage of vulnerable apps is not comprehensive enough. Therefore, we need to analyze the typical vulnerable apps and extract the common features of all the vulnerable codes as the basis of static detection. On the one hand, the challenges are how to define the detection rules by analyzing vulnerable codes. On the other hand, the code we find may not be executed. The code may be test code only and is not really invoked in the process of running, or the code may be executed through system callbacks and will never be executed. In order to determine whether the code is actually executed, we need to trace back the vulnerable code to find which Activity executes it. However, if we start Activity that we find directly, the program may crash with high probability. Therefore, we need to find a path from the Main Activity to the entry Activity, and then execute it sequentially. Finally, it can be confirmed whether there is a real vulnerability through the MITM attack tool.

3.2. How to simulate human operations

To simulate human operations, DCDroid needs to understand the UI elements on the current screen and provides the necessary operation. For example, text boxes need to input content and radio boxes need to check. It then selects the special UI elements to click according to the results of static analysis. Existing tools are not suitable for our UI automation, such as MonkeyRunner (MonkeyRunner, 2020) whose execution has no special purpose and mainly relies on random clicks. Therefore, it is difficult to trigger vulnerable code. Another automation framework Appium (appium, 2020) can use specific scripts to run UI elements precisely. However, it has no commonality and needs to be customized for each app. Some other automation tools, such as FlowDroid (Arzt et al., 2014) and DroidScope (Yan and Yin, 2012), can track method call relationship. However, they cannot trigger dynamic vulnerabilities. Dynodroid (Machiry et al., 2013) focuses on processing automatic input. In contrast, Smart Droid (Zheng et al., 2012) and Brahmastra (Bhoraskar et al., 2014) cannot deal with Web UI.

DCDroid that we developed is an automatic tool for UI elements based on AndroidViewClient (AndroidViewClient, 2020). It can get all UI elements on the screen and execute click events on a specified UI element, etc. It can also run UI elements with potential vulnerable code first.

3.3. How to run efficiently

If all the UI elements associated with vulnerable code are executed, the running time is very long. We find that similar UI elements tend to have the same implementation logic. The Activity always has many similar elements for which we can select a part of them to execute. However, some elements are similar in size but are not a collection of the same implementation, such as various tabs. For example, Fig. 3 shows an app of delicious food. The four different tabs (green box) above it represent different contents, while the sub-menus (red box) of each category tab represent the same contents. The difficulty is how to select elements by appropriate methods so that the execution speed can be accelerated without impacting the accuracy of detection results.

3.4. How to run effectively

While much existing work focused on how to run apps stably, there is no effective tool to identify the traffic generated by apps. As some apps are run in background and some apps start another app in the process of their running, the traffic collected is not always generated by current active apps. Although DCDroid will return to the app we are running when starting another app, some other app's traffic will be mixed inevitably. Therefore, the traffic obtained by MITM attack tools may not be generated by current app. For instance, the traffic may be generated by other apps which are running in the background. How to precisely extract the traffic generated by each app from the mixed traffic is a challenge.

4. Our method

In this section, we first introduce the framework of our method, then describe the static detection process, and finally introduce the dynamic detection process.

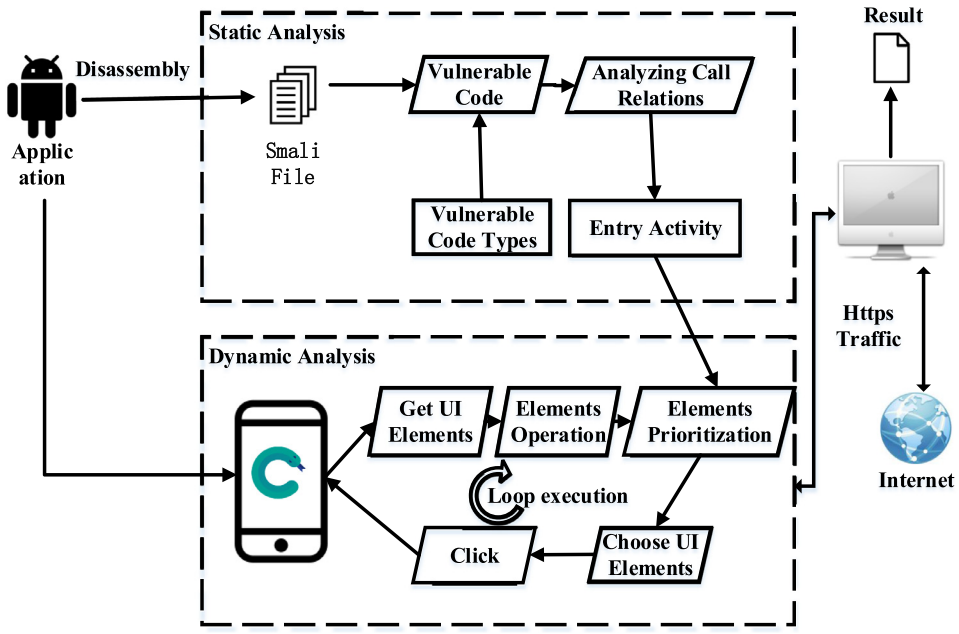


Fig. 1. System overview.

Table 1
The number and the type of vulnerabilities from the test samples.

Type of Vulnerablity	X509TrustManager	HostNameVerifier	WebViewClient sslError	X509HostnameVerifier
numbers	83	64	58	67

4.1. System overview

An overview of DCDroid is presented in Fig. 1. Given an app, we first conduct static analysis. We disassemble the app to get the Smali file and then locate the vulnerable points according to the characteristics of the vulnerable code. By analyzing the method call relationships, we get the vulnerable entry Activity. We then start dynamic detection, install the app into the smart phone with the ADB management tool and start to run the app to trigger the potential vulnerable code. We intercept traffic with MITM attack tools and use VPNService to capture traffic on smart phones. Finally, we confirm those true vulnerable apps by comparing the traffic between the smart phone and the attack tool.

4.2. Static analysis

4.2.1. Disassembling apps

Android apps can be decompiled into Java code or disassembled into Smali code directly. We choose to disassemble an app into Smali code because we only need to analyze the call relationship of the code without knowing its design. Smali code can be disassembled faster and is less affected by confusion technology. It can be done with apktool (android apktool, 2020). Androguard (Androguard, 2020) can analyze its call relationships.

4.2.2. Vulnerable code analysis

Based on existing related work (He et al., 2015; Fahl et al., 2012; Sounthiraraj et al., 2014) and disassembling 100 typical vulnerable apps manually, we propose 4 types of vulnerable code. The number of each type among 100 apps is shown in Table 1. A typical vulnerable code is shown in Fig. 2. As shown in the figure, the vulnerable method returns *void* directly without any check.

- X509TrustManager: We check if the code extends the *X509TrustManager* class. If it happens, we check the *checkClientTrusted* and *checkServerTrusted* methods to see if the method has only one instruction which is *return-void*. If so, we consider the method is vulnerable.

```
.method public checkClientTrusted([Ljava/security/cert/X509Certificate;Ljava/lang/String;)V
    .locals 0
    .param p1, "chain"    # [Ljava/security/cert/X509Certificate;
    .param p2, "authType"    # Ljava/lang/String;
    .annotation system Ldalvik/annotation/Throws;
        value = {
            Ljava/security/cert/CertificateException;
        }
    .end annotation

    .prologue
    .line 157
    return-void
.end method
```

Fig. 2. An example of vulnerable code.

- HostNameVerifier: We check whether the *HostnameVerifier* interface is implemented in the code. If it exists, we check the *verify* method. If the method has only two instructions, and the first instruction begins with *const* and the second instruction is *return*, we then consider the method vulnerable.
- WebViewClient sslError: We check whether the code extends the *WebViewClient* class. If that happens, we check *onReceivedSslError* method. If this method has only two instructions, and the first instruction starts with *invoke-virtual* and ends with *Landroid/webkit/SslErrorHandler;->pro ceed()V*, and the second instruction is *return void*, we then consider this method vulnerable.
- X509HostnameVerifier: We check whether there is an instruction named *sget-object* in the class which extends *X509TrustManager* class. If that happens, we check

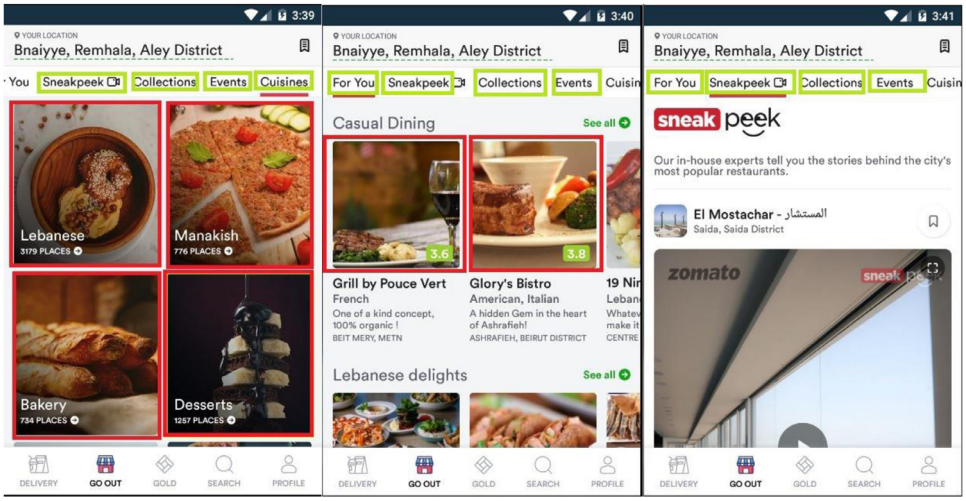


Fig. 3. The similar and unsimilar elements of an app.

if it is ended with *ALLOW_ALL_HOSTNAME_VERIFIER Lorg/apache/http/conn/ssl/X509Hostname Verifier*. If it happens, we check whether the next instruction is *->setHostname Verifier (Lorg/apache/ http/n/ssl/X509Hostname Verifier);V*. If it exists, we then consider the method vulnerable.

4.2.3. Call relation analysis

We use Algorithm 1 to analyze the method call relationship of an app. We analyze the call relationship of the method with the Method Call Graph (MCG) to determine the entry point (including Activity, Service) where the vulnerable method is finally executed. By recording these entry points, we give priority to these entry points in the dynamic detection phase so that they can be executed.

We start from vulnerable methods found in static analysis, traverse their methods (these methods are called their parents), and then traverse their parents until the method has not been called by other methods. We then jump to the constructor of the class where the method belongs to and continue traversing until we reach a constructor that has never been called by other app code. These constructors are therefore called only by system code and are the entry points of an app.

Algorithm 1 Find Final Caller of Vulnerable Method.

Require: *MCG* : *Method Call Graph*,*VM* : *Vulnerable Method*
Ensure: *Result* : *Set of Entry Point Methods*

```
function FindFinalCaller(MCG, VM)
    if method_callers of VM not null then
        for each method_caller in method_callers do
            FindFinalCaller(MCG,method_caller)
        end for
    else
        for each method in class(method_callers) do
            if method is class' constructor and method is not in Result then
                Result.append(method)
                FindFinalCaller (MCG, method)
            else method is in Result
                return
            end if
        end for
    end if
end function
```

4.2.4. Get entry activity

The entry points will be associated with Activity and Service. The final call points of vulnerable methods have been known with Algorithm 1. We also find the entry Activity associated with the vulnerable code. Then we analyze the call relationship of the activity and build the Activity Call Graphs (ACG). The node in the ACGs represents Activity, the leaf represents a vulnerable Activity, and the edge represents an Intent of the Activity. The detail of construction is shown in Algorithm 2. We consider click events mainly in the algorithm. Some complex events (such as swipe and long touch) are ignored because they are unlikely to trigger HTTPS connections. Then we can get the execution path from the Main Activity to the entry Activity. During the dynamic analysis, we run the Activities contained in Activity path one by one under the guidance of ACGs.

Algorithm 2 Build Activity Call Graph(ACG).

Input: *AndroidManifest.xml potientalVulnerableViews*
Output: *ActivityCallGraph(ACG)*

```
function BuildACG(potientalVulnerableViews, AndroidManifest.xml)
    ACG=ϕ
    for each views in potientalVulnerableViews do
        EntryActivity=findActivityByViewID(views.getId)
        ActivitySet=initActivity(AndroidManifest.xml)
        ACG=Fun(EntryActivity,AvtivitySet,ACG)
    end for
    return ACG
end function
function Fun(EntryActivity, AtivitySet, ACG)
    for each activity in ActivitySet do
        if activity == MainActivity then
            return ACG
        end if
        if activity jump to EntryActivity By method.Event(such as Button.click) then
            ACG ∪ (activity → EntryActivity)
            fun(activity,ActivitySet,ACG)
        end if
    end for
end function
```

4.3. Dynamic analysis

4.3.1. UI Automation

UI automation is the core component of DCDroid. It makes an app run to the direction that vulnerable code can be executed and avoids meaningless execution. There are three tasks for UI automation components: obtain UI elements and operate them, reduce UI elements and determine priorities, run app and manage UI status.

When the app enters an Activity, it needs to get every element of the Activity and extract the attributes of the element, such as the text of the button and the input form of the text box. With the information obtained, the system creates appropriate events to operate elements so that Activity can jump from one to another normally. For example, select events are created for check boxes and input events are created for text boxes. To achieve this goal, we use the AndroidViewClient to manage component. It can get the UI elements, create appropriate events for the UI elements and execute the dynamic operation of a specific app.

4.3.2. Acceleration

In order to accelerate the operation, we select only a part of UI elements to execute from similar elements. Through our analysis, we find that it is appropriate to select four to execute for similar elements. Take Fig. 3 as an example, on the one hand, it can avoid meaningless execution of duplicate elements. On the other hand, it can ensure that similar UI elements with different code logic are executed, too (Such elements are usually tab options with no more than 4). When acquiring UI elements, we add up to four similar UI elements at most and simply delete the extra elements. Algorithm 3 describes our method. If a component inherits from the same parent component and has the same attributes (such as the same size, color, etc.), then we consider that they are the same. With this strategy, we can speed up our dynamic detection process.

Algorithm 3 Similar Views Find.

```
Require: Views
Ensure: Set of Vulnerable Views
function SimilarViewsFinder(Views)
    for each view in Views do
        viewSize=view.getSize();
        if viewSize in viewWithSameParentAndSize then
            viewWithSameParentAndSize[viewSize].append(view);
        else
            for each exitView in viewWithSameParentAndSize do
                if difference(exitView, view) < threshold then
                    viewWithSameParentAndSize[exitView].append(view);
                else
                    viewWithSameParentAndSize[view].append(view);
                end if
            end for
        end if
    end for
    for key in viewWithSameParentAndSize.keys do
        if len(viewWithSameParentAndSize[key]) < 4 then
            potientalVulnerableViews[key].append(viewWithSameParentAndSize[key]);
        else
            potientalVulnerableViews[key].append(viewWithSameParentAndSize[key][1.4]);
        end if
    end for
    return potientalVulnerableViews
end function
```

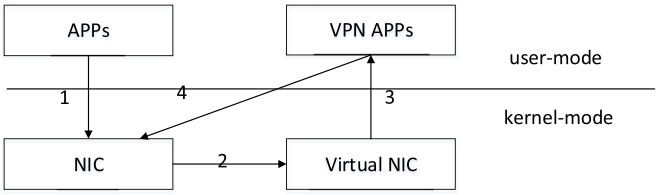


Fig. 4. The work flow of VPNService.

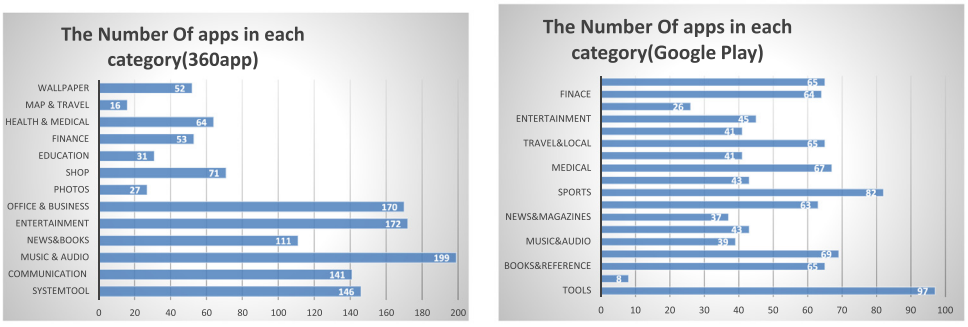
We use AndroidViewClient to manage the state of the Activity. It provides the API to obtain the information about the current Activity and the way to operate the window elements. The Activities are executed one by one according to the ACG information provided with the static detection phase. Specifically, for each execution path, we execute each Activity's method.Event (e.g., *button.Click*). If we enter a new Activity, we will continue to execute until the entry Activity. If a crash happened during the execution or an execution path is completed, we will exit the app and start Main Activity again. If the execution ends normally, we release the corresponding execution path. If the execution ends abnormally, we retry once and release it.

4.3.3. Set proxy

In order to execute a SSL/TLS MITM attack, all traffic between Android clients and servers must be intercepted. Fidder (Fidder, 2020) is a widely-used tool in this area. However, in our experiment, not only do we need to intercept HTTPS traffic, but we also need conduct some processing for traffic. Therefore Fidder is not suitable for us. We use mitmproxy (Mitmproxy, 2020) as a MITM attack tool. Mitmproxy is a proxy tool. It forwards requests and executes MITM attack like a normal proxy. It generates its own fake certificate and sends it to the client to start an attack. We do not add mitmproxy certificates to the host's trust certificate list. Therefore, those HTTPS traffic that could be established successful are generated by vulnerable apps. The biggest advantage of mitmproxy is that it can manage intercepted requests by using Python script. With this feature, we can easily analyze intercepted requests, such as the number, type and content of requests. It is helpful to analyze the experimental results to determine the app with certificate verification vulnerabilities.

4.3.4. Traffic analysis

In the MITM attack tool, we can only get all the traffic intercepted. However, we are not sure which app generates the traffic. In other words, we cannot judge whether the app is vulnerable according to the intercepted traffic only. Therefore, it is necessary to identify which app generates the traffic. We use Android's VPNService (VPNService, 2020) interface to capture network packets on the client side. Its work flow is shown in Fig. 4. First, an app sends the data to Network Interface Card (NIC) with socket (step 1). Then, NIC sends all data packet to the Virtual NIC (step 2). Next, VPN opens the /dev/tun and reads data from it, and then the packet can be saved or changed (step 3). Finally, VPN sends data to NIC. Sockets used by VPN apps must be explicitly bound to NIC to avoid infinite loop of data packets (step 4). The method is to read /proc/net/tcp and /proc/net/tcp6 files to get the IP of PID and its URLs. The UsageStatsManager class (UsageStatsManager, 2020) can get the currently running app's PID. The PackageManager class (PackageManager, 2020) can get the corresponding relationship between PID and app. We thus can get the corresponding relationship between each HTTPS traffic and the app. By comparing the



(a) 360app (b) Google Play

Fig. 5. The number of apps in each category.

Table 2
Analysis of data.

			360app		Goole Play		Total
		Count	Percentge	Count	Percentge	Count	Percentge
static	Disassembly failure	13	1.04%	17	1.77%	30	1.36%
	Potential vulnerable apps	281	22.43%	176	18.33%	457	20.65%
	Free from such vulnerabilities	959	76.53%	767	79.90%	1726	77.99%
	Tatal apps	1253	100%	960	100%	2213	100%
dynamic	Vulnerability confirmed	151	55.87%	94	53.41%	245	53.61%
	Vulnerability free	130	44.13%	82	46.59%	212	46.39%
	Total apps	281	100%	176	100%	457	100%

HTTPS traffic obtained by smart phones and MITM attack tools, the vulnerable app can be confirmed. We develop an Android traffic capture tool to achieve this function.

5. Experiments

We use a Windows 10 computer as the running environment, an Ubuntu 12.0 as the attack environment, and two Android 6 smart phones as the client environment.

5.1. Dataset

The dataset in the experiments comes from two app markets. One is 360app, a popular app market in China. We downloaded 1253 popular apps using crawlers in December 2018. These apps belong to 13 subcategories of the “software” category. Another market is Google Play. We downloaded 960 popular apps available in June 2016. The apps from Google Play belong to 18 subcategories of the “software” category. The number of apps under each subcategory is shown in Fig. 5. For each app, we get its apk file, size, developer information and description. In particular, we removed apps larger than 100M in size, because most of these apps are complex games which can cause frequent crashes in dynamic analysis.

5.2. Static analysis

We conduct static analysis on both two data sets. In the static analysis, we use apktool to disassemble apps into Smali file. Some apps cannot be disassembled successfully. In our experiments, 30 apps cannot be disassembled. It took 65 s to analyse per app on average. The results of static detection are summarized in Table 2, that shows that 30 (1.36%) of 2213 apps from 360app and Google Play cannot be disassembled. There are 457 (20.65%) apps that have potentially vulnerable code and these apps are considered to have potential certificate verification vulnerabilities. They need further dynamic detection to confirm whether they are really vulnerable. We compared the static detection results with AndroBugs (AndroBugs, 2020), kingkong (kingkong, 2020) and appscan (appscan, 2020). The results are shown in Table 3 and Fig. 6. AndroBugs is slightly better than DCDroid in terms of detection accuracy in static detection. However, it generates a big number of false positives without dynamic detection. As for kingkong and appscan, DCDroid is better in terms of detection accuracy in static detection. Besides, they cannot detect HostNameVerifier vulnerability. Both of these two tools also contain a number of false positives. As a result, DCDroid is not the best in static detection phase. However, the major advantage of DCDroid is that we can run the app dynamically and remove false positives.

There are 1726 apps that do not have the vulnerabilities we defined. The app size is much larger than itself after disassembling. Therefore, in order to save space, we delete the Smali file after finishing static analysis.

5.3. Dynamic analysis

In dynamic analysis, we use AndroidViewClient to operate two Android smart phones and run apps. On average, each app spends 183 seconds. In the process of running, considering the network speed and other reasons, DCDroid waits 2 seconds for each window to finish loading. If DCDroid does not wait for loading, the detection can be finished faster. However, it’s probably easier to crash.

The results of dynamic detection are shown in Table 1. It is seen that 245 apps from 360app and Google Play are identified as having certificate validation vulnerabilities, accounting for 53.61% of potential vulnerable codes and 11.07% of all apps. This indicates that 11.07% of apps in our dataset have certificate validation vulnerabilities. It is worth noting that 8 apps cannot be run dynamically and crashed due to version and other reasons. Because the number is pretty small, we simply consider that they have no certificate verification vulnerabilities. The number of vulnerable apps in each category is shown in Table 1 (including Google’s app and

Table 3
The vulnerabilities detected with different tools.

	X509TrustManager	HostNameVerifier	WebViewClient sslError	X509HostnameVerifier	Vulnerable apps
ourtool	163	159	114	161	457
AndroBugs	218	164	149	88	506
kingkong	162	–	167	153	386
appscan	171	–	133	115	373

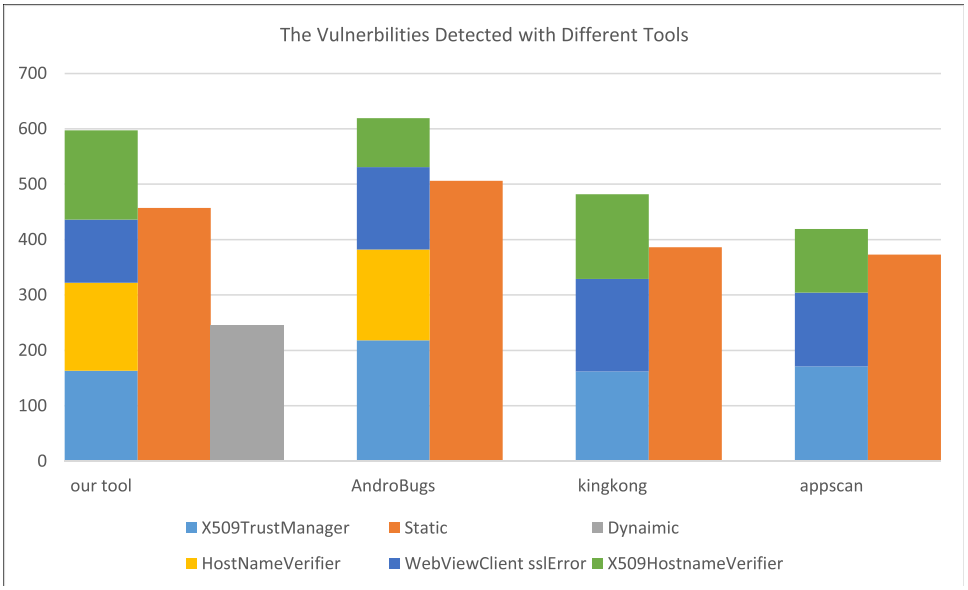


Fig. 6. The vulnerbilities detected with different tools.

360app's app). It is seen from the Table that the percentage of certificate validation vulnerabilities in 360app is 12.05% and in Google Play is 9.79%. There are more vulnerable apps in 360app than those in Google Play.

In dynamic analysis, we take some strategies to reduce the number of app crashes and speed up the app's execution. Comparing with our previous work, we have reduced the number of app crashes and make app executed faster. The number of crashes is 4.18 per app on average in our previous work while 1.79 now which has reduced 3.39 times. We analyze 457 apps and find it costs 271 seconds per app on average in our previous work while it only costs 183 seconds now with the strategy. The running time is reduced by 88 seconds per app on average. In addition, it can ensure that the app stop after a period of time. According to our traffic processing method, we have effectively reduced 3 false positive cases and made our detection results more reliable.

5.4. Vulnerable apps

Because the app's version of 360app is relatively new and easy to find all versions, we mainly analyze the vulnerable apps in 360app.

We analyze the vulnerabilities of different kinds of apps in 360app. The percentage of each category is shown in Fig. 7. Among them, News&Books, finance and Health&Medical categories take the biggest percentage. By analyzing these apps, we find that some apps have code vulnerabilities. Besides, many of these apps are vulnerable because they invoke the third-party SDKs that have vulnerabilities, such as pushSDK and the old version of weiboSDK (the new version has been fixed). Another notable finding is that apps developed by the same organization often have similar vulnerabilities, such as SohuNews and SohuVideo which are all vulnerable and developed by the same organization.

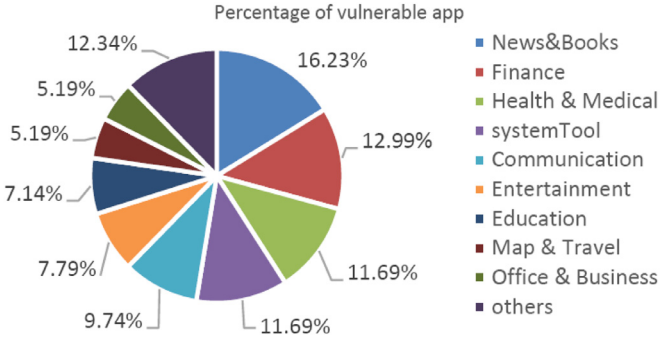


Fig. 7. Percentage of vulnerable apps.

We randomly select 30 apps with certificate verification vulnerabilities, and analyze the evolution of 156 historical versions of them. For each application with different versions, we first try to obtain it from application markets. If some of these applications cannot be found, we will try to obtain them from the provider's official website. If they still cannot be found, we consider they are missing. The results are shown in Fig. 8. The vertical axis is version number (we only select major version updated) and the horizontal axis is the app. The red dots represent vulnerable apps while the green dots represent non-vulnerable ones. The white dots indicate that the related versions do not exist or cannot be found. We find that most of the apps with low versions tend to have vulnerabilities when a higher version exists. In addition, we find that the apps with lower versions have fewer or possibly no vulnerability comparing with high versions. Through manual analysis, we speculate that one reason is that the low version is released earlier and it may not use SSL/TLS at all, such as #1, #7 app, or early version may be simple and not easy to invoke vulnerabilities. As the

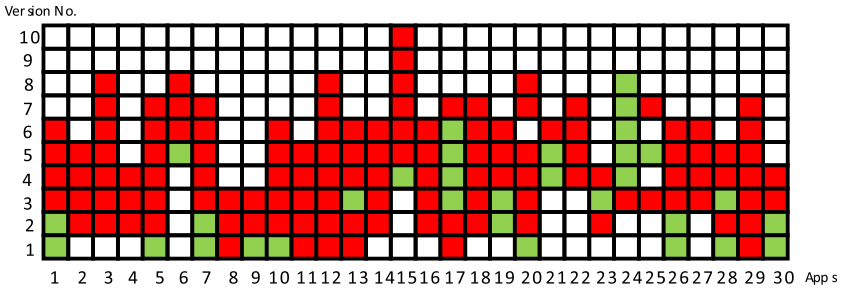


Fig. 8. Statistics of vulnerable app version.

complexity of the code increases, vulnerabilities are more likely to occur. The use of the third-party libraries may also increase such vulnerabilities. Based on our analysis, if vulnerability is invoked, the probability of fixing the vulnerability is very small in the later version. For example, only #24 app has completely fixed this vulnerability without invoking new vulnerabilities.

6. Discussion and analysis

DCDroid implements automatic detection of digital certificate verification vulnerabilities with static detection and dynamic detection under the guidance of static detection. Although it has been demonstrated as effective, there are still limitations.

In static detection, we check the vulnerable code, such as the method that only has a simple instruction-return. However, some code may have complex implementation of the method and finally still does not conduct the verification. We cannot check this type of vulnerable code, which may lead to false negatives. Comparing with other tools like AndroBugs, kingkongand and appscan, DCDroid does not show a better result in the static detection phase. As far as we know, the reason is that our detection rules are different. However, DCDroid's advantage is that we can reduce false positives with dynamic analysis.

In dynamic detection, in order to speed up the execution of dynamic operation, we delete some similar UI components. Although we prove that for most cases this operation will not change the detection results, we still cannot estimate the number of false negatives caused by the deletion. Besides, some dynamic analysis which needs specific trigger conditions (such as login) may lead to false negatives.

The static analysis can be used to detect other vulnerabilities if the detection rules are defined. The dynamic analysis can be used to run apps dynamically in other detection. Of course, it need specific verification methods like setting proxy in detecting SSL/TLS vulnerabilities. Developers should be very careful during the development and should follow the specifications. In addition, developers should check the security carefully when referencing the third-party library. We use Android 6 as the test platform. However, the tools we use in the experiments are applicable to all Android versions.

7. Related work

Information security has been widely studied. Privacy, such as identity privacy (Li and Shen, 2013; Li et al., 2018b) or search privacy (Li et al., 2019a), has also been a research topic. Li and Shen (2013) proposed an effective rank-based attack model and algorithms for rank anonymization and hypergraph reconstruction, in the aim to privacy preserving for hypergraph-based data publishing. Li et al. (2019a) showed various privacy problems in forward and backward security model and proposed efficient algorithms to solve the problem for pattern privacy leakage. Outliers can be regarded as anomalies or even intrusions. There exist related work focusing on machine learning methods to detect outliers or anomalies in streaming data. Zhang (2018) reviewed state-of-the-art techniques for extracting representation and discovering knowledge from streaming and temporal data where outliers or anomalies exist. In Particular, they proposed novel models such as KDE-Track (Qahtan et al., 2015) and StrAP (Zhang et al., 2008; 2014) to effectively detect outliers.

Smart contracts can be regarded as applications that are run on Blockchain. We have designed ContractWard (Wang et al., 2020b) to detect vulnerabilities in smart contracts. While these work lies in the broad range of information security, the most related work includes security and privacy in Android ecosystems, and Android SSL/TLS vulnerability analysis.

7.1. Security & privacy in android ecosystems

Improper implementation of SSL/TLS leads to security problems such as users' privacy leakage. In our previous work (Wang et al., 2018b), we analyzed the malicious behavior in Android apps with machine learning methods. We constructed some common classifiers to detect malapps. Our method achieved the accuracy of 99.39% in the detection of malapps and achieved the best accuracy of 82.93% in the categorization of benign apps. We also detected malicious software with Convolutional Neural Network (CNN) (Wang et al., 2019b). The accuracy with our models was improved by 5% compared with SVM. We also analyzed the privacy leakage of third-party libraries (He et al., 2019). We evaluated 150 popular apps and collected 1909 privacy information related call chains. We found that the third-party libraries access to privacy information accounted for the largest proportion. We also proposed an automated community detection method (Su et al., 2018) for Android malapps by building a relation graph based on their static features. Our method outperformed the traditional clustering methods and achieved the best performance with rand statistic of 94.93% and accuracy of 79.53%. Earlier, we studied the permission risk (Wang et al., 2014b) of Android apps. We built a malapp detectors on risky permissions and the detection rates are 94.62% with a false positive rate as 0.6%. We also detected the malware after extracting the different features of apps and Android platforms (Wang et al., 2017). Privacy leakage is an important issue in Android ecosystem. We studied the privacy leakage of sensors in Android apps (Liu et al., 2018a). We developed a tool called "SDFDroid" to identify the used sensors types and generate the sensor data propagation graphs in each app. We found that some third-party libraries registered all the types of sensors recklessly. Li et al. detected Inter-Component privacy leaks in Android apps (Li et al., 2015) and studied malicious code grafting systematically (Li et al., 2017). Dong et al. (2018) developed a tool called "FraudDroid" to detect AD fraud automatically for Android Apps and found 335 cases in 12,000 apps. Li et al. (2018a) proposed a novel solution for the detection of Android malware based on a machine learn-

ing algorithm with high detection accuracy. The method is able to effectively and efficiently identify the malware apps in a scalable and dynamical way. While these work focused on the detection of malapps or on the analysis of privacy leakage in apps, we are motivated to identify vulnerable apps.

There are also some tools to dynamically analyze apps. MonkeyRunner (MonkeyRunner, 2020) is a tool that can run Android apps while its execution has no special purpose and relies on random clicks. As a result, it is difficult to trigger vulnerable code. Appium (appium, 2020) uses specific scripts to run UI elements precisely. However, it has no commonality and needs to be customized for each app. FlowDroid (Arzt et al., 2014) can generate call graphs, locate target methods and analyze the path of information flow. However, it cannot trigger vulnerabilities dynamically. DroidScope (Yan and Yin, 2012) is an Android analysis platform that performs the traditional virtualization-based malware analysis. Dynodroid (Machiry et al., 2013) is an input generation system that focuses on automatic input only. Smart Droid (Zheng et al., 2012) triggered a certain behavior through automated UI interactions. Bhoraskar et al. (2014) used static analysis to construct a page transition graph and discover execution paths to invoke the third-party code. However, these tools or methods cannot deal with Android UI or web UI.

7.2. Android SSL/TLS vulnerability analysis

In 2012, Fahl et al. (2012) first raised the problem of Android SSL/TLS vulnerabilities and developed a static analysis tool called MalloDroid to detect their vulnerabilities. They found that 8% of apps had this problem. The authors further manually analyzed 100 of these apps and found that 41 of them had this problem. The weakness of their method is that it requires manual analysis to confirm the vulnerabilities. Brubaker et al. (2014) studied certificate validation logic. They mainly carried out large-scale certificate validation tests on the server and found many vulnerabilities in browsers and SSL/TLS libraries. Sounthiraraj et al. (2014) developed a tool called SMV-HUNTER that introduced automated analysis of Android apps on this problem. However, they only analyzed the original Android apps and did not cover the hybrid web apps. Liu et al. (2018c) aimed at SSL/TLS error-handling vulnerability in hybrid web apps. They only considered one type of vulnerable code. All these methods aimed at a specific kind of Android apps. The vulnerabilities detected are not comprehensive enough. In addition, they start Activity directly in the process of detection, which may lead to application crash. Gao et al. (2018) studied vulnerability evolution on Android apps and found that the vulnerabilities' fix cycle of the risk is very long.

8. Conclusion

In this work, we propose an effective method and develop a tool called "DCDroid" to identify the vulnerabilities in the implementation of SSL/TLS digital certificate verification in Android system. We analyze 960 apps from Google Play and 1253 apps from 360app with DCDroid. Extensive experimental results show that with initial static analysis, 457 (20.65%) apps contain potential security risks in the implementation of SSL/TLS and with further dynamic analysis 245 (11.07%) out of 2213 apps are finally identified as vulnerable to MITM and phishing attacks. DCDroid also analyzes the characteristics of vulnerable apps, including their categories and version evolution. Comprehensive experimental results and analysis demonstrates the effectiveness and efficiency of DCDroid in the detection of SSL/TLS certificate verification vulnerabilities in Android apps.

In the future work, we are designing mechanisms to automatically patch the vulnerabilities found with DCDroid.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Yingjie Wang: Methodology, Software, Validation, Writing - original draft, Visualization. **Guangquan Xu:** Methodology, Formal analysis. **Xing Liu:** Methodology, Software, Formal analysis. **Weixuan Mao:** Methodology, Validation, Visualization. **Chengxiang Si:** Formal analysis, Data curation. **Witold Pedrycz:** Methodology, Validation. **Wei Wang:** Conceptualization, Methodology, Formal analysis, Investigation, Writing - review & editing, Writing - original draft, Project administration, Funding acquisition, Supervision.

Acknowledgements

The work reported in this paper was supported in part by Natural Science Foundation of China, under Grant U1736114, and in part by National Key R&D Program of China, under grant 2017YFB0802805.

References

Activities, 2020. https://developer.android.com/guide/components/activities/intro-activities.
AndroBugs, 2020. https://github.com/AndroBugs/AndroBugs_Framework.
Android Developer Training ś Security with HTTPS and SSL, 2020. https://developer.android.com/training/articles/security-ssl.html.
Androguard, 2020. http://code.google.com/p/androguard/.
AndroidViewClient, 2020. https://github.com/dtmilano/AndroidViewClient.
Android apktool, 2020. http://code.google.com/p/android-apktool/.
Appium, 2020. https://github.com/appium/appium.
Appscan, 2020. http://appscan.360.cn/.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, pp. 259–269. doi:10.1145/2594291.2594299.
Bhoraskar, R., Han, S., Jeon, J., Azim, T., Chen, S., Jung, J., Nath, S., Wang, R., Wetherall, D., 2014. Brahmastra: driving apps to test the security of third-party components. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20–22, 2014, pp. 1021–1036.
Brubaker, C., Jana, S., Ray, B., Khurshid, S., Shmatikov, V., 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014, pp. 114–129. doi:10.1109/SP.2014.15.
Clark, J., van Oorschot, P.C., 2013. Sok: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013, pp. 511–525. doi:10.1109/SP.2013.41.
Dong, F., Wang, H., Li, L., Guo, Y., Bissyandé, T.F., Liu, T., Xu, G., Klein, J., 2018. Frauddroid: automated ad fraud detection for android apps. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04–09, 2018, pp. 257–268. doi:10.1145/3236024.3236045.
Elenkov, N., 2014. Android Security Internals: An In-Depth Guide to Android's Security Architecture.
Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., Freisleben, B., 2012. Why eve and mallory love android: an analysis of android SSL (in)security. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16–18, 2012, pp. 50–61. doi:10.1145/2382196.2382205.
Feng, L., Wang, W., Zhu, L.-n., Zhang, Y., 2009. Predicting intrusion goal using dynamic Bayesian network with transfer probability estimation. J. Netw. Comput. Appl. 32 (3), 721–732.
Fidder, 2020. https://www.telerik.com.
Gao, J., Li, L., Kong, P., Bissyandé, T.F., Klein, J., 2018. Poster: on vulnerability evolution in android apps. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, pp. 276–277.
Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V., 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In: The ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16–18, 2012, pp. 38–49. doi:10.1145/2382196.2382204.

Google, H. a., 2020. https://developer.android.com/training/articles/security-ssl.html.
Guan, X., Wang, W., Zhang, X., 2009. Fast intrusion detection based on a non-negative matrix factorization model. J. Netw. Comput. Appl. 32 (1), 31–44. doi:10.1016/j.jnca.2008.04.006.
GnuTLS Transport Layer Security Library, 2020. https://developer.android.com/training/articles/security-ssl.html.
He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V.N., Yang, R., Zhang, Z., 2015. Vetting SSL usage in applications with SSLINT. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015, pp. 519–534. doi:10.1109/SP.2015.38.
He, Y., Yang, X., Hu, B., Wang, W., 2019. Dynamic privacy leakage analysis of android third-party libraries. J. Inf. Sec. Appl. 46, 259–270. doi:10.1016/j.jisa.2019.03.014.
Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, 2020. https://tools.ietf.org/html/rfc5280.
kingkong, 2020. https://service.security.tencent.com/kingkong.
Li, J., Huang, Y., Wei, Y., Lv, S., Liu, Z., Dong, C., Lou, W., 2019. Searchable symmetric encryption with forward search privacy. IEEE Trans. Depend. Secure Comput. doi:10.1109/TDSC.2019.2894411.
Li, J., Sun, L., Yan, Q., Li, Z., Srisa-an, W., Ye, H., 2018. Significant permission identification for machine-learning-based android malware detection. IEEE Trans. Ind. Inform. 14 (7), 3216–3225. doi:10.1109/TII.2017.2789219.
Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.D., 2015. IccTA: detecting inter-component privacy leaks in android apps. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1, pp. 280–291. doi:10.1109/ICSE.2015.48.
Li, L., Li, D., Bissyandé, T.F., Klein, J., Traon, Y.L., Lo, D., Cavallaro, L., 2017. Understanding android app piggybacking: asystematic study of malicious code grafting. IEEE Trans. Inf. Forensics Secur. 12 (6), 1269–1284. doi:10.1109/TIFS.2017.2656460.
Li, L., Liu, J., Cheng, L., Qiu, S., Wang, W., Zhang, X., Zhang, Z., 2018. Creditcoin: a privacy-preserving blockchain-based incentive announcement network for communications of smart vehicles. IEEE Trans. Intell. Transp. Syst. 19 (7), 2204–2220.
Li, L., Xu, G., Jiao, L., Li, X., Wang, H., Hu, J., Xian, H., Lian, W., gao, h., 2019. A secure random key distribution scheme against node replication attacks in industrial wireless sensor systems. IEEE Trans. Ind. Inf. doi:10.1109/TII.2019.2927296.
Li, Y., Shen, H., 2013. On identity disclosure control for hypergraph-based data publishing. IEEE Trans. Inf. Forensics Secur. 8 (8), 1384–1396.
Liu, X., Liu, J., Wang, W., He, Y., Zhang, X., 2018. Discovering and understanding android sensor usage behaviors with data flow analysis. World Wide Web 21 (1), 105–126. doi:10.1007/s11280-017-0446-0.
Liu, X., Liu, J., Wang, W., Zhu, S., 2018. Android single sign-on security: issues, taxonomy and directions. Fut. Gener. Comp. Syst. 89, 402–420. doi:10.1016/j.future.2018.06.049.
Liu, X., Liu, J., Zhu, S., Wang, W., Zhang, X., 2019. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. IEEE Trans. Mob. Comput. doi:10.1109/TMC.2019.2903186.
Liu, Y., Zuo, C., Zhang, Z., Guo, S., Xu, X.-S., 2018. An automatically vetting mechanism for SSL error-handling vulnerability in android hybrid web apps. World Wide Web 21 (1), 127–150. doi:10.1007/s11280-017-0458-9.
Machiry, A., Tahiliani, R., Naik, M., 2013. Dynodroid: an input generation system for android apps. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013, pp. 224–234. doi:10.1145/2491411.2491450.
Mitmproxy, 2020. https://docs.mitmproxy.org/stable/.
Mobile Operating System Market Share Worldwide Dec 2018. Dec 2020,., 2020. http://gs.statcounter.com/os-market-share/mobile/worldwide.
MonkeyRunner, 2020. https://developer.android.com/studio/test/monkeyrunner.
Number of available applications in the Google Play Store from December 2009. to December 2020,., 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
PackageManager, 2020. https://developer.android.com/reference/android/content/pm/PackageManager.
Qahtan, A.A., Alharbi, B., Wang, S., Zhang, X., 2015. A PCA-based change detection framework for multidimensional data streams: change detection in multidimensional data streams. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10–13, 2015, pp. 935–944.
Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L., 2014. Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014.
Su, D., Liu, J., Wang, W., Wang, X., Du, X., Guizani, M., 2018. Discovering communities of malapps on android-based mobile cyber-physical systems. Ad Hoc Netw. 80, 104–115. doi:10.1016/j.adhoc.2018.07.015.
Su, D., Liu, J., Wang, X., Wang, W., 2019. Detecting android locker-ransomware on chinese social networks. IEEE Access 7, 20381–20393. doi:10.1109/ACCESS.2018.2888568.
UsageStatsManager, 2020. https://developer.android.com/reference/android/app/usage/UsageStatsManager.
VPNService, 2020. https://developer.android.com/reference/android/net/VpnService.
Wang, W., Battiti, R., 2006. Identifying intrusions in computer networks with principal component analysis. In: Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, April 20–22 2006, Austria, pp. 270–279. doi:10.1109/ARES.2006.73.
Wang, W., Gao, Z., Zhao, M., Li, Y., Liu, J., Zhang, X., 2018. Droidensemble: detecting android malicious applications with ensemble of string and structural static features. IEEE Access 6, 31798–31807. doi:10.1109/ACCESS.2018.2835654.
Wang, W., Guan, X., Zhang, X., 2004. A novel intrusion detection method based on principle component analysis in computer security. In: Advances in Neural Networks - ISNN 2004, International Symposium on Neural Networks, Dalian, China, August 19–21, 2004, Proceedings, Part II, pp. 657–662. doi:10.1007/978-3-540-28648-6_105.
Wang, W., Guan, X., Zhang, X., 2004. Profiling program and user behaviors for anomaly intrusion detection based on non-negative matrix factorization. In: 2004 43rd IEEE Conference on Decision and Control, Vol. 1, pp. 99–104.
Wang, W., Guan, X., Zhang, X., 2008. Processing of massive audit data streams for real-time anomaly intrusion detection. Comput. Commun. 31 (1), 58–72. doi:10.1016/j.comcom.2007.10.010.
Wang, W., Guan, X., Zhang, X., Yang, L., 2006. Profiling program behavior for anomaly intrusion detection based on the transition and frequency property of computer audit data. Comput. Secur. 25 (7), 539–550. doi:10.1016/j.cose.2006.05.005.
Wang, W., Guyet, T., Quiniou, R., Cordier, M.-O., Masseglia, F., Zhang, X., 2014. Autonomic intrusion detection: adaptively detecting anomalies over unlabeled audit data streams in computer networks. Knowl.-Based Syst. 70, 103–117. doi:10.1016/j.knosys.2014.06.018.
Wang, W., He, Y., Liu, J., Gombault, S., 2015. Constructing important features from massive network traffic for lightweight intrusion detection. IET Inf. Secur. 9 (6), 374–379.
Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X., 2018. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Fut. Gener. Comp. Syst. 78, 987–994. doi:10.1016/j.future.2017.01.019.
Wang, W., Liu, J., Pitsilis, G., Zhang, X., 2018. Abstracting massive data for lightweight intrusion detection in computer networks. Inf. Sci. 433–434, 417–430. doi:10.1016/j.ins.2016.10.023.
Wang, W., Shang, Y., He, Y., Li, Y., Liu, J., 2020. Botmark: automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors. Inf. Sci. 511, 284–296.
Wang, W., Song, J., Xu, G., Li, Y., Wang, H., Su, C., 2020. Contractward: automated vulnerability detection models for ethereum smart contracts. IEEE Trans. Netw. Sci. Eng. 1–12. doi:10.1109/TNSE.2020.2968505.
Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X., 2014. Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. 9 (11), 1869–1882. doi:10.1109/TIFS.2014.2353996.
Wang, W., Zhang, X., Gombault, S., 2009. Constructing attribute weights from computer audit data for effective intrusion detection. J. Syst. Softw. 82 (12), 1974–1981.
Wang, W., Zhang, X., Gombault, S., Knapskog, S.J., 2009. Attribute normalization in network intrusion detection. In: The 10th International Symposium on Pervasive Systems, Algorithms, and Networks, ISPAN 2009, Kaohsiung, Taiwan, December 14–16, 2009, pp. 448–453. doi:10.1109/I-SPAN.2009.49.
Wang, W., Zhang, X., Pitsilis, G., 2010. Abstracting audit data for lightweight intrusion detection. In: Information Systems Security - 6th International Conference, ICISS 2010, Gandhinagar, India, December 17–19, 2010. Proceedings, pp. 201–215. doi:10.1007/978-3-642-17714-9_15.
Wang, W., Zhao, M., Gao, Z., Xu, G., Xian, H., Li, Y., Zhang, X., 2019. Constructing features for detecting android malicious applications: issues, taxonomy and directions. IEEE Access 7, 67602–67631. doi:10.1109/ACCESS.2019.2918139.
Wang, W., Zhao, M., Wang, J., 2019. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient Intell. Hum. Comput. 10 (8), 3035–3043. doi:10.1007/s12652-018-0803-6.
Wang, X., Wang, W., He, Y., Liu, J., Han, Z., Zhang, X., 2017. Characterizing android apps' behavior for effective detection of malapps at large scale. Fut. Gener. Comp. Syst. 75, 30–45. doi:10.1016/j.future.2017.04.041.
Wang, Y., Liu, X., Mao, W., Wang, W., 2019. Dcdroid: automated detection of SSL/TLS certificate verification vulnerabilities in android apps. In: Proceedings of the ACM Turing Celebration Conference - China, ACM TUR-C 2019, Chengdu, China, May 17–19, 2019, pp. 137:1–137:9. doi:10.1145/3321408.3326665.
Xie, N., Wang, X., Wang, W., Liu, J., 2019. Fingerprinting android malware families. Frontiers Comput. Sci. 13 (3), 637–646.
Xu, G., Guo, B., Su, C., Zheng, X., Liang, K., Wong, D.S., Wang, H., 2020. Am i eclipsed? a smart detector of eclipse attacks for ethereum. Comput. Secur. 88, 101604. doi:10.1016/j.cose.2019.101604.
Xu, G., Wang, W., Jiao, L., Li, X., Liang, K., Zheng, X., Lian, W., Xian, H., Gao, H., 2019. Soprotector: safeguard privacy for native SO files in evolving mobile iot applications. IEEE Internet Things J. doi:10.1109/JIOT.2019.2944006.
Yan, L.-K., Yin, H., 2012. Droidscope: seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8–10, 2012, pp. 569–584.
Zhang, X., 2018. Mining streaming and temporal data: from representation to knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13–19, 2018, Stockholm, Sweden, pp. 5744–5748.
Zhang, X., Furtlehner, C., Germain-Renaud, C., Sebag, M., 2014. Data stream clustering with affinity propagation. IEEE Trans. Knowl. Data Eng. 26 (7), 1644–1656. doi:10.1109/TKDE.2013.146.

Zhang, X., Furtlehner, C., Sebag, M., 2008. Data streaming with affinity propagation. In: Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15–19, 2008, Proceedings, Part II, pp. 628–643.

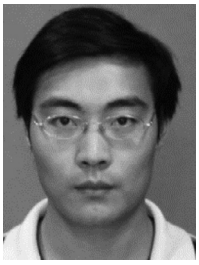
Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W., 2012. Smartdroid: an automatic system for revealing UI-based trigger conditions in android applications. In: SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA, pp. 93–104. doi:10.1145/2381934.2381950.

Yingjie Wang is currently a M.S. student of the School of Computer and Information Technology, Beijing Jiaotong University, China. He received his B.S. degree from Southwest JiaoTong University, China, in 2017. His main research interests lie in mobile security.

Guanquan Xu received the Ph.D. degree from Tianjin University, in 2008. He is currently a Full Professor with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, China. His research interests include cyber security and trust management. He is a member of CCF.

Xing Liu received the Ph.D. degree from School of Computer and Information Technology, Beijing Jiaotong University, China, in 2019. He received his B.S. degree from Beijing Jiaotong University in 2012. He visited the department of Computer Science and Engineering, The Pennsylvania State University, from October 2015 to October 2016. His main research interests lie in mobile security and privacy.

Weixuan Mao received the B.S. degree in computer science and engineering and the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2009 and 2017, respectively. He is currently a Security Engineer with the National Computer Network Emergency Response Technical Team/Coordination Center of China (CNCERT/CC). His research interests include intrusion/malware detection, data driven security analysis, and program analysis.



Chengxiang Si received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2012. He has been an Engineer with the National Computer Network Emergency Response Technical Team/Coordination Center of China, since 2012. His research interests include networks and information security.

Witold Pedrycz received the M.Sc., Ph.D., and the D.Sci. degrees from the Silesian University of Technology, Gliwice, Poland. He is a Professor and the Canada Research Chair of Computational Intelligence with the Department of Electrical and Computer Engineering, Faculty of Engineering, King Abdulaziz University, Jeddah, Saudi Arabia. He is also with the Systems Research Institute of the Polish Academy of Sciences, Warsaw, Poland. His current research interests include computational intelligence, fuzzy modeling and granular computing, and software engineering. He has published numerous papers in the above areas. He has also authored 15 research monographs covering various aspects of computational intelligence, data mining, and software engineering. Dr. Pedrycz was a recipient of the prestigious Norbert Wiener Award from the IEEE Systems, Man, and Cybernetics Society in 2007, the IEEE Canada Computer Engineering Medal 2008, the Cajastur Prize for Soft Computing from the European Centre for Soft Computing for "pioneering and multifaceted contributions to granular computing" in 2009, the Killam Prize in 2013, and the Fuzzy Pioneer Award 2013 from the IEEE Computational Intelligence Society. He was elected as a Foreign Member of the Polish Academy of Sciences in 2009 and a fellow of the Royal Society of Canada in 2012. He has been a member of numerous program committees of the IEEE conferences in the area of fuzzy sets and neurocomputing.

Wei Wang is currently a full professor in the Department of Information Security, Beijing Jiaotong University, China. He is currently also affiliated with Computer, Electrical and Mathematical Sciences and Engineering Division, King Abdullah University of Science and Technology Saudi Arabia. He earned his Ph.D. degree in control science and engineering from Xi'an Jiaotong University, in 2006. He was a postdoctoral researcher in University of Trento, Italy, from 2005 to 2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, from 2007 to 2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009 to 2011. He has authored or coauthored over 90 peer reviewed papers in various journals and international conferences. He is an Editorial Board member of Computers & Security and a Young AE of Frontiers of Computer Science. His main research interests include mobile, computer and network security.