DCDroid: Automated Detection of SSL/TLS Certificate Verification Vulnerabilities in Android Apps

Yingjie Wang  
yingjie_w@bjtu.edu.cn  
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University

Weixuan Mao  
maoweixuan@cert.org.cn  
National Computer Network Emergency Response Technical Team / Coordination Center of China

Xing Liu  
xingliu@bjtu.edu.cn  
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University

Wei Wang  
wangwei1@bjtu.edu.cn  
1Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University  
2Division of Computer, Electrical and Mathematical Sciences & Engineering (CEMSE) King Abdullah University of Science and Technology (KAUST)Thuwal 23955–6900, Saudi Arabia

ABSTRACT

Current Android applications (apps) often use Security Socket Layer (SSL)/Transport Layer Security (TLS) protocols to transmit users’ information, as the implementation of SSL/TLS secures the transmission of sensitive information. However, for various reasons, Android developers fail to properly implement SSL/TLS during the development of an app, resulting in security risks. The improper implementations include trusting all certificates, trusting all domain names, or ignoring certificate verification errors. These improper implementations may result in Man-In-The-Middle (MITM) attacks or phishing attacks. In this work, we are motivated to detect vulnerabilities in implementation of SSL/TLS in Android apps by designing and implementing a tool called DCDroid (Detecting SSL/TLS Certificate verification vulnerabilities in Android apps) with the combination of static analysis and dynamic analysis. We focus on four types of vulnerable schema and locate the potential vulnerable code snippets in apps with static analysis. In dynamic analysis, we prioritize the triggering of User Interface (UI) components based on the results with static analysis to confirm the misuse of SSL/TLS. The dynamic analysis benefits from the static analysis and removes false positives. With DCDroid we analyze 960 apps from Google Play and 1253 apps from 360app. The experimental results show that 457 (20.65%) apps contain potential security risks in the implementation of SSL/TLS. Guided by the static analysis, we further confirm that 248 (11.21%) out of 2213 apps are truly vulnerable to MITM and phishing attacks. By analyzing the categories, ranks and version evolution of these detected vulnerable apps, we find that apps of News&Books are more likely to introduce SSL/TLS risks. We also find that the fix cycle of the risk is very long. We provide suggestions on SSL/TLS certificate verification to Android developers in order to deal with the SSL/TLS certificate verification vulnerabilities.

CCS CONCEPTS

• Security and privacy → Mobile and wireless security;  • Networks → Mobile networks.

KEYWORDS

Android security, MITM, SSL/TLS, static analysis, dynamic analysis

ACM Reference Format:


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 85% of the smart phone’s market in the world[7]. According to Google’s statistics, there are over 2.6 million Android apps that are downloaded for over hundreds of billions times from Google Play as of March 2019[11]. 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. Unfortunately, improper implementation of SSL/TLS certificates can lead to Man-In-The-Middle (MITM) attacks[17] and phishing attacks[20]. 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, but 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 pose as 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. In our previous work, we detected malicious apps [28][29][30][31][32][33][34] for analyzed privacy leakage [21][23][24][25] with different methods. There exist related work on Android MITM attacks caused by improper implementation of SSL/TLS in Android’s apps. Fahl et al. [18] first raised this issue and developed a static analysis tool called MalloDroid to detect vulnerabilities. They found that 8% of apps had this problem. The authors chose 100 of these apps to manually analyze and found that 41 of them did have problems. The weakness of the approach is that it requires manual analysis to confirm vulnerabilities. Georgiev et al. [27] developed a tool called SMV-HUNTER that introduced automated analysis of Android apps on this issue. However, they only analyzed the original Android apps and did not cover the hybrid web apps. Yang et al. [35] aimed at SSL/TLS error-handling vulnerability in hybrid web apps but didn’t consider other vulnerable code. All of these approaches aimed at a specific kind of Android apps, and the vulnerabilities detected are not comprehensive enough. In addition, they started Activity directly in the process of detection, which may lead to application crash.

In order to automatically, comprehensively and efficiently detect the SSL/TLS certificate verification vulnerabilities in Android apps, in this work, we developed a new tool called DCDroid combining static detection and dynamic detection. In static detection phase, we define a comprehensive type of vulnerable code. We disassembled an app to get Smali code and search the code to locate the SSL/TLS vulnerable code, we then get the entry point by analyzing the invocation relationship of the method. We use the results of static analysis on the app to guide the dynamic detection, in which we give priority to triggering User Interface(UI) components that call vulnerable code in the execution process. Next we set up proxy servers to carry out MITM attacks. We consider an app truly has this kind of vulnerabilities when an app is attacked successfully. As we detect native Android apps and hybrid Android apps at the same time, DCDroid has a wider coverage of vulnerable code than previous work. Thus, DCDroid is able to detect more certificate verification vulnerabilities. In contrast to [27][20], we did not directly start Activity with vulnerable code in APP so that DCDroid is more stable in the detection. In summary, we make the following contributions:

- We developed an automated tool called DCDroid to detect SSL/TLS vulnerabilities with combination of static and dynamic analysis in real time. We start executing the Activity of the vulnerable code from the entrance of APP instead of starting it directly. By dynamic execution, DCDroid is more stable than previous work with few crashes and additionally improves the detection accuracy.
- We systematically studied the vulnerabilities of Android in the process of implementing SSL/TLS. By analyzing a number of apps, we summarize the vulnerable codes of implementing SSL/TLS in Android app. With DCDroid, more vulnerabilities can be found in the static detection phase, and the coverage of detection can be improved. As a result, we found 20.65% of the vulnerable code in the static detection phase, which is higher than the previous work.
- We tested 2213 apps and found that 457 apps were vulnerable through static testing. After dynamic analysis we found that 248 apps were truly vulnerable. More than 10% of apps have vulnerabilities of SSL/TLS. We thus provide suggestions to developers based on the test results.
- We analyzed the categories, ranks and version evolution of vulnerable apps. To the best of our knowledge, this is the first SSL/TLS security detection for app version evolution. We found that more vulnerable code may be introduced when app functions tend to be more complex. In addition, the heavy use of third-party libraries may lead to similar vulnerabilities in various versions of vulnerable apps, which requires developers to pay more attention on the detection of vulnerabilities of apps.

The remainder of the 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 including 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. Finally we conclude this paper in Section 7.

2 BACKGROUND

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

2.1 SSL/TLS & Android

SSL and its successor TLS protect the message from MITM attack by encrypting network messages. To achieve this goal, it is important to obtain certificate containing public keys from the server. According to RFC 5280[8] 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;
Automated Detection of SSL/TLS Certificate Verification Vulnerabilities

Certificate 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, but also allows developers to implement their own certificate verification method. 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[19] and so on. However, in the process of implementation, vulnerable certificate verification methods are often introduced for various reasons, including[18],[35]:

- 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.

2.2 Android UI

Activity[1] 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. Specially, the Activity entered at the start of the app is called the 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 the background. For example, Service can get data from the network or do some computational task 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 Activity. It is used to jump to another Activity from this Activity.

We regard the interface as a directed graph, the node of the graph is Activity or Service, the edge of the graph is intent, the main activity is the root of the graph. Running all activities means the traversal of the graph. By abstracting the UI into a graph, the automation algorithm of UI can be implemented more conveniently.
2.3 MITM

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 pose as 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 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 Define Potential Vulnerable Code

Because it is time-consuming to run apps dynamically, we first need to determine which apps are potentially vulnerable of SSL/TLS. We eliminate some apps by static detection and provide guidance for dynamic detection. We need to define vulnerable code reasonably. If the selected SSL/TLS vulnerable code is not representative, it will cause more false negatives. There are limited definitions of vulnerable code in the 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 vulnerable codes as the basis of static detection. The challenges are how to define the detection rules by analyzing vulnerable codes.

3.2 Trigger Vulnerable Code

We need to find activities that vulnerable code eventually executes through appropriate methods, and find a path from entry activity to target activity.

It is not difficult to find vulnerable code, but it is not easy to determine whether the vulnerable code is actually executed, because the code may be test code only and the app 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 must trace back through the vulnerable code to find the Activity which executed code. However, if we start Activity directly, the program may crash more easily. Therefore, we find a path from the entrance activity to the target activity, then execute it sequentially, and finally confirm whether there is a real vulnerability through the MITM attack tool.

3.3 Fast-running

If all the UI elements associated with vulnerable code are executed, it will take a lot of time, because similar UI elements tend to have similar implementation logic, sometimes the same Activity will have many similar elements, for these elements we can select a part of them to execute. But even if some elements satisfy such conditions, they are not a collection of similar elements, such as various tabs. For example, the Figure 1 shows a file management app. The three different options (green box) above it are different contents, but the sub-menu (red box) of category option is the same content. 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 Simulate Human Operations

To simulate automated testing, we first need to understand the UI elements on the current screen and provide the necessary elements operation, such as text boxes need to input content, radio boxes need to check. And then we select the interface elements with high priority to click according to the results of static analysis. Existing tools are not suitable for our UI automation, such as monkeyrunner[10], whose execution has no purpose and relies on random clicks, so it is difficult to trigger vulnerable code. Appium[5], another automation framework, can use specific scripts to run UI elements precisely, but it has no commonality and needs to be customized for each test APP. Some other automation tools, such as FlowDroid[15] and DroidScope[22], can track method call relationship, but can’t trigger dynamic vulnerabilities. Dynodroid[26] focuses on processing automatic input, Smart Droid[36], Brahmastra[16] can’t deal with Web UI.

We have developed an automatic running tool for UI elements based on AndroidViewClient[4]. With our tool, we can get all UI elements on the screen, run click events on a specified UI element, etc. We can also run UI elements with potential vulnerable code first.

![Figure 3: State Management.](image)

4 DCDROID

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

4.1 System Overview

An overview of DCDroid is presented in Figure 2. Given an app, we first conduct static analysis. We disassemble the app to get the Smali file, then locate the vulnerable points according to the characteristics of the vulnerable code. By analyzing the method call relationship, we get the vulnerable entry Activity. We combine the vulnerable entry activity with the static string to determine the priority of the UI elements. Then we do dynamic detection, we
Algorithm 1 Find Final Caller Of Vulnerable Method

Input: MCG : Method Call Graph, VM : Vulnerable Method
Output: Result : Set of Entry Point Methods
1: function FindFinalCaller(MCG, VM)
2: if method_callers of VM not null then
3:   for each method_caller in method_callers //continue recursion do
4:     FindFinalCaller(MCG, method_caller)
5:   end for
6: else
7:   for each method in class(method_callers) //add constructors of vulnerable class do
8:     if method is class’ constructor and method is not in Result then
9:       Result.append(method)
10:   end if
11: end if
12: return
13: end function

install app to the mobile phone with the ADB management tool, and then we start to dynamically execute app to trigger potentially vulnerable code. We intercept traffic on MITM attack tools and use VPNService to capture traffic on mobile phones. Finally, we confirm those real vulnerable apps by comparing traffic between phone and the attack tool.

4.2 Static Analysis

4.2.1 Disassembling Apps.

Android apps can be decompiled into Java code or directly disassembled into Smali code. We choose to disassemble it 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 it is less affected by confusion technology. It can be done using apktool[3], and Androguard[2] can finish the analysis of its call relationship easily. We can also easily get String.xml files by disassembling them.

4.2.2 Vulnerable Code Analysis.

From [18, 20, 27] and manually disassembling 100 typical vulnerable APPs, we propose 4 types of vulnerable code:

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

**HostnameVerifier:** We check whether there is an instruction named sget-object in the class which extends X509TrustManager. If that happens, we check if it is ended with ALLOW_ALL_HOSTNAME_VERIFIER Lorg/apache/http/conn/ssl/X509HostnameVerifier, and if that happens, we check whether the next instruction is _-> setHostnameVerifier (Lorg/apache/http/n/ssl/X509HostnameVerifier); V, if it exists, we consider the method vulnerable.

**WebViewClient sslError:** We check whether the code extends the WebViewClient class, and 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;->proceed()V, and the second instruction is return void, we consider this method vulnerable.

**X509HostnameVerifier**:

We use the Algorithm1 to analyse the method call relationship of app. We analyze the call relationship of the method with the method call graph (MCG), so as 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 as to ensure that vulnerable entry points are executed first.

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

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 through Algorithm1. By analyzing their association with these entry Activity, we can trigger them first in dynamic analysis.

4.3 Dynamic Analysis

4.3.1 Device Management.

We use the Android phone for testing because the emulator is more likely to crash. We open multiple devices at the same time, use ADB tools to manage each device, and monitor the state of the device. When the device is ready, we install potentially vulnerable apps to the device for UI automation. If the app runs unexpectedly, we will uninstall apps directly. A simple device management such as Algorithm2.

4.3.2 UI Automation.

UI automation is the core component of the system. It makes app run to the direction that vulnerable code is 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 an app starts Activity, it needs to get every element on the Activity, extracts the attributes of the element, such as the text of the button and the input form of the text box. Using 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 radio boxes and check boxes, and input events are created for text boxes. To achieve this goal, we use the AndroidViewClient component to manage, which can get the UI elements, create appropriate events for the UI elements, and execute the dynamic operation of a specific app.

In order to speed up 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 Figure 2 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 also be executed (Such elements are usually tab options no more than 4). When acquiring UI elements, we add up to four similar UI elements at most, and we simply delete the extra elements. With this strategy, we can speed up our dynamic detection efficiency. We prioritize elements associated with vulnerable code from static detection analysis, and prioritize execution in the same activity. For example, if there are multiple clicks events in the same activity, we will give priority to those with potentially vulnerable code.

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. We record the execution status of each activity. If the potential vulnerable code in the current activity has been fully executed, we jump to the next activity. Otherwise, when the activity status changes, we still return to the activity to continue execution. When the app crashes, the tool returns to the main activity to re-execute and when all activities are executed, it exits the current app. The state management for a single activity is shown in Figure 3.

Algorithm 3 AnalyseUrl

**Input:** apps

**Output:** related url

1: url, ip=getUrlAndIpFromNetworkCard();
2: app, uid= getUrlAndIpUidFromOS();
3: ip, uid=getIpAndUidFromFile();
4: url, app=combine(url, ip, uid, app)

4.3.3 Set Proxy.

In order to execute an SSL/TLS MITM attack, all traffic between Android clients and servers must be intercepted. Fiddler[6] is a widely-used tool in this area, but in our experiment, not only do we need to intercept HTTPS traffic, but we also need do some
Table 1: Static analysis data

<table>
<thead>
<tr>
<th></th>
<th>360app</th>
<th>Google Play</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Percentge</td>
<td>Count</td>
</tr>
<tr>
<td>Disassembly failure</td>
<td>13</td>
<td>1.04%</td>
<td>17</td>
</tr>
<tr>
<td>Potential vulnerable apps</td>
<td>281</td>
<td>22.43%</td>
<td>176</td>
</tr>
<tr>
<td>Free from such vulnerabilities</td>
<td>959</td>
<td>76.53%</td>
<td>767</td>
</tr>
<tr>
<td>Total Apps</td>
<td>1253</td>
<td>100%</td>
<td>960</td>
</tr>
</tbody>
</table>

Table 2: Dynamic analysis of data

<table>
<thead>
<tr>
<th></th>
<th>360app</th>
<th>Google Play</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Percentge</td>
<td>Count</td>
</tr>
<tr>
<td>Vulnerability confirmed</td>
<td>154</td>
<td>54.80%</td>
<td>94</td>
</tr>
<tr>
<td>Vulnerability free</td>
<td>127</td>
<td>45.20%</td>
<td>82</td>
</tr>
<tr>
<td>Potential vulnerable apps</td>
<td>281</td>
<td>100%</td>
<td>176</td>
</tr>
</tbody>
</table>

4.3.4 Traffic Analysis.

In the MITM attack tool, we can only get all the traffic intercepted, but we can not judge whether the APP is vulnerable according to the intercepted traffic only. Therefore, it is necessary to know which APP generates traffic. We use Android’s VPNService[14] interface to capture packages on the client side. The concrete method is to read /proc/net/tcp and /proc/net/tcp6 files to get the IP of uid and its links, use UsageStatsManager class[13] to get the currently running APP’s uid, and use PackageManager class[12] to get the corresponding relationship between uid and APP, so we can get the corresponding relationship between each HTTPS traffic and app. By comparing the HTTPS traffic obtained by mobile phones with the traffic of MITM attack tools, we can know the vulnerable APP. We developed an Android phone capture tool to achieve this function with Algorithm3. Finally, we confirm the real vulnerable app by comparing the HTTPS traffic between the phone and the MITM attack tool.

5 EXPERIMENTS

We use a Windows10 computer as the running environment, an Ubuntu12.0 as the attack environment, and two Android6 mobile phones as the test environment.

5.1 Dataset

The dataset in the experiments comes from two app markets, one is 360app, one of the most popular app market in China. We downloaded 1,253 popular apps using crawlers in Dec. 2018. These apps belong to 13 subcategories of the “software” category. Another is Google Play. We downloaded 960 popular apps available in Jun. 2016. The apps from Google Play belong to 18 subcategories of the “software” category. The number of apps under each subcategory is shown in Figure 4. For each app, we get its apk file, size, developer and description. Specially, we have deleted apps larger than 100M in size, because these apps can cause frequent crashes in execution in dynamic analysis.

5.2 Static Analysis

We do static analysis on both two data sets. In the process of static analysis, we use apktool to disassemble app as Smali file. Some apps can not be disassembled successfully. In our experiment, 30 apps can not be disassembled. The results of static detection are summarized in Table 1, which shows that 30 (1.36%) of 2213 apps from 360app and Google Play can not be disassembled. There are 457 (20.65%) apps 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.

Besides, 1726 apps do not have the vulnerabilities that we defined. The volume of the app is much larger than itself after disassembling, so in order to save hardware space, we delete the Smali file after finishing static analysis.

5.3 Dynamic Analysis

In the process of dynamic analysis, we use AndroidViewClient tool to operate two Android mobile phones and run apps. On average, each app spends 183 seconds. In the process of running, considering the network speed and other reasons, we wait 3 seconds for each window to complete loading. If we don’t use latency, we can finish it faster, but it’s probably easier to crash.
The percentage of each category is shown in Figure 5. Among them, the News&Books, Finance, and Health&Medical categories take the greatest percentage. By analyzing these apps, we find that some apps do have code vulnerabilities, besides, many of these apps are vulnerable because of invoking third-party SDKs which have vulnerabilities such as pushSDK and the old version of weiboSDK (the greatest percentage. By analyzing these apps, we find that some apps do have code vulnerabilities, besides, many of these apps are vulnerable because of invoking third-party SDKs which 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, they are all vulnerable and are developed by same organization.

We analyze the ranks of vulnerable apps in 360 app. The results are shown in Table 3. We find that popular apps are more likely to introduce certificate validation vulnerabilities. In our sample, we found that more than 60% of the apps with vulnerabilities were in the top 600, and only less than 40% ranked 600-1253. After our analysis, we find that there are two reasons. Popular apps often have more complex functions, so more vulnerable third-party libraries are invoked. Lower ranked apps use fewer HTTPS connections or even do not use HTTPS at all, they are easier to be attacked.

We randomly select 30 apps with certificate verification vulnerabilities, and analyze the evolution of 156 historical versions of them. The results are shown in Figure 6, the vertical axis is version number (we only select major version updated) and the horizontal axis is app, red dot is vulnerable app while green dot is not. We find that most of the low versions of apps tend to have vulnerabilities when a new version has, and the lower versions have fewer or possibly none. Through manual analysis, 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 complexity of the code increases, vulnerabilities are more likely to occur, and the use of third-party libraries may also increase such vulnerabilities. From 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, including static detection and dynamic detection under the guidance of static detection. There are still limitations.

In the static detection phase, we check the vulnerable code, such as 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 negative.

In the dynamic detection phase, in order to speed up the exe-
cution of dynamic operation, we delete some similar UI compo-
nents, although we prove that for most cases, this operation will
not change the detection results. However, we cannot estimate the
number of false negatives caused by the deletion.

Our work is also limited by the support of hardware. In the experi-
ments only two devices are used in the testing process. However,
DCDroid is easy to migrate to multi-device conditions by using
multiple threads to run on multiple devices at the same time. In
addition, the proxy can use different ports.

7 CONCLUSION
In this work, we developed an automated tool called DCDroid,
to detect the vulnerabilities in the implementation of SSL/TLS digi-
tal certificate verification in Android system. DCDroid analyzed
960 apps from Google Play and 1253 apps from 360app. Extensive
Experimental results show that with initial static analysis 457
(20.65%) apps contain potential security risks in the implementa-
tion of SSL/TLS and with further dynamic analysis 248 (11.21%) out of
2213 apps are finally detected as vulnerable to MITM and phish-
ing attacks. We also use DCDroid to analyze the characteristics
of vulnerable apps, including their categories, ranks and version
evolution. DCDroid demonstrates its effectiveness and efficiency 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 by DCDroid.

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
intro-activities
VpnService
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel,
Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Medani. 2014. Flow-
Droid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint
global-market-share-held-by-smartphone-operating-systems/
content/pm/PackageManager
app/usage/UsageStatsManager
[15] Ranvi Bhoraskar, Seungyoup Han, Jinsung Jeon, Tanvirul Azim, Shuo Chen,
Jaeyeon Jung, Sunan Nath, Wang Rui, and David Wetherall. 2014. Brahmsa: Driv-
ing Apps to Test the Security of Third-Party Components. In Usenix Security
Symposium.
ing Past Challenges and Evaluating Certificate Trust Model Enhancements. In
Security & Privacy.
[17] Sascha Fahl, Marian Harbach, Thomas Maders, Bernd Freidemann, and Mathew
Smith. 2012. Why eve and mallory love android.an analysis of android ssl
(use)security. In Acm Conference on Computer & Communications Security.
[18] Martin Georgiev, Subodhi Iyengar, Suman Jana, Rishtul Anshulbha, Boneh Dan,
and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating
SSL certificates in non-browser software. In Acm Conference on Computer &
Communications Security.
[19] Hongsong He, Vaibhav Rastogi, Yinzhao Cao, Chen Yan, V. N. Venkataraman,
Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL Usage in Applications with
SLLINT. (2015).
leakage analysis of Android third-party libraries. Journal of Information Security
and Applications 46 (2019), 259 – 270. https://doi.org/10.1016/j.jisa.2019.03.014
the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In
Usenix Conference on Security Symposium.
Discovering and understanding android sensor usage behaviors with data flow
017-0446-0
security: Issues, taxonomy and directions. Future Generation Comp. Syst. 89
Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android
org/10.1109/TMC.2019.2903186
generation system for Android apps. In Joint Meeting on Foundations of Software
Engineering.
[26] David Soumitharaj, Justin Salus, Garrett Greenwood, Zhiqiang Lin, and Latifur
Khan. 2014. SMV HUNTER: Large Scale, Automated Detection of SSL/TLS Man-
in-the-Middle Vulnerabilities in Android Apps. In Network & Distributed System
Security Symposium.
[27] Dan Su, Jiqiang Liu, Wei Wang, Xiaoayong Wang, Xiangdong Du, and Mohsen
Guizani. 2018. Discovering communities of malapps on Android-based mobile
org/10.1016/j.adhoc.2017.05.015
Locker-Ransomware on Chinese Social Networks. IEEE Access 7 (2019), 20891-
20939. https://doi.org/10.1109/ACCESS.2018.2885868
[29] Wei Wang, Zhenzhun Gao, Meichen Zhao, Yulong Li, Jiqiang Liu, and Xiangliang
Ensemble of String and Structural Static Features. IEEE Access 6 (2018), 31798–
31807. https://doi.org/10.1109/ACCESS.2018.2835654
Detecting Android malicious apps and categorizing benign apps with ensemble of
org/10.1016/j.future.2017.01.019
[31] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang
Zhang. 2014. Exploring Permission-Induced Risk in Android Applications for
Malicious Application Detection. IEEE Trans. Information Forensics and Security
detection with a hybrid model based on deep autoencoder and convolutional
neural network. Journal of Ambient Intelligence and Humanized Computing (28
[33] Xing Wang, Wei Wang, Yongzhong He, Jiqiang Liu, Zhen Han, and Xiangliang
Zhang. 2017. Characterizing Android apps’ behavior for effective detection of
doi.org/10.1016/j.future.2017.04.041
[34] Lui Yang, Chaoshun Zuo, Zonghua Zhang, Shangqing Guo, and Xinshun Xu. 2017.
An automatically vetting mechanism for SSL error-handling vulnerabilities in
android hybrid Web Apps. World Wide Web Internet & Web Information Systems
[35] Cong Zheng, Shixiong Zhu, Shaofu Dai, Guefei Gu, Xiaorui Gong, Xinhui Han,
and Wei Zou. 2012. SmartDroid an automatic system for revealing UI-based
trigger conditions in android applications. (2012).