Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications

Yunhan Jack Jia, Qi Alfred Chen, Yikai Lin, Chao Kong, Z. Morley Mao
University of Michigan
{jackjia, alfchen, yklin, chaokong, zmao}@umich.edu

Abstract—**Open ports are typically used by server software to serve remote clients, and the usage historically leads to remote exploitation due to insufficient protection. Smartphone operating systems inherit the open port support, but since they are significantly different from traditional server machines in performance and availability guarantees, little is known about how smartphone applications use open ports and what the security implications are. In this paper, we perform the first systematic study of open port usage on mobile platform and their security implications. To achieve this goal, we design and implement OPAnalyzer, a static analysis tool which can effectively identify and characterize vulnerable open port usage in Android applications.**

Using OPAnalyzer, we perform extensive usage and vulnerability analysis on a dataset with over 100K Android applications. OPAnalyzer successfully classifies 99% of the mobile usage of open ports into 5 distinct families, and from the output, we are able to identify several mobile-specific usage scenarios such as data sharing in physical proximity. In our subsequent vulnerability analysis, we find that nearly half of the usage is unprotected and can be directly exploited remotely. From the identified vulnerable usage, we discover 410 vulnerable applications with 956 potential exploits in total. We manually confirmed the vulnerabilities for 57 applications, including popular ones with 10 to 50 million downloads on the official market, and also an app that is pre-installed on some device models. These vulnerabilities can be exploited to cause highly-severe damage such as remotely stealing contacts, photos, and even security credentials, and also performing sensitive actions such as malware installation and malicious code execution. We have reported these vulnerabilities and already got acknowledged by the application developers for some of them. We also propose countermeasures and improved practices for each usage scenario.

1. Introduction

An open port (or a listening port) is a communication endpoint for accepting incoming connections in computer networking model, typically used by server applications to handle requests from remote clients. However, these ports can also be connected by malicious clients if not carefully protected, exposing potential vulnerability in the server software to remote exploitation. Such inherent weakness has always accompanied the usage of open ports throughout the history of network services, opening doors for large numbers of severe Internet attacks such as TCP SYN flooding attacks [27], the Conficker worm [14], and more recently the Heartbleed bug [9]. To mitigate the problem in these traditional usage scenarios, firewalls and user authentication mechanisms are usually adopted.

In the recent evolution to the mobile era, smartphone operating systems inherit the support for open port. But for smartphone applications (apps), traditional open port use cases such as hosting network services no longer apply. One major reason is that compared to stationary server machines with wired network connectivity, the mobility nature of smartphones makes it difficult to maintain a stable IP address. Moreover, the IPs assigned to mobile devices are often behind a NAT (network address translation) preventing incoming network connections. Also, continuously receiving network traffic can easily drain the battery of a mobile device, leading to a form of denial-of-service (DoS) attack [49]. Due to these inherent differences, our current understanding about smartphone usage of open ports are rather limited.

With the immense popularity of smartphones, any potential smartphone open port usage may directly expose end users to severe damage. Several such examples have already been reported recently, called “Wormhole” apps [12], where open ports in popular Android apps allow an attacker to remotely collect location data, insert contacts, and even install app without authorization, and over 100M devices are affected. While these exploits are alarming, it is still unclear whether these vulnerabilities are exposed by popular use cases of open ports in the smartphone ecosystem, or just by poor implementation practices.

In this work, we perform the first systematic study of open port usage and the security implications on mobile platform. To achieve this goal, we design and implement a tool called *OPAnalyzer*, which can effectively identify and characterize vulnerable open port usage in Android apps. To use OPAnalyzer, we first formalize open port app design pattern in the language of program analysis, which in high level specifies what sensitive functions are triggered from open ports, and how they are triggered. With these definitions, OPAnalyzer first uses static taint analysis to track the information flow from the remote input entry point, and identifies the sensitive functionalities that can potentially be triggered. After this step, a set of usage paths of the open port are generated, which will lead to remote exploits if not well protected. To help prioritize human inspection, OPAnalyzer examines the security checks along the usage paths guarding the sensitive functionality. If the execution of a given path is found to have no constraints or contains only *weak* checks, a potential remote exploit is directly revealed (§4.4). OPAnalyzer also dynamically tests whether the vulnerable port is open by default, and labels the weak paths as *highly insecure* if the corresponding port opens automatically at app launching time. For high precision, our design leverages the Amandroid approach [46], which supports flow-, context-sensitive data-flow analysis.

To ensure high effectiveness, we overcome several engineering challenges in the tool implementation. First, our analysis needs accurate identification of the permission-protected APIs, but the API to permission mappings provided by the most recent work, PScout [21], are incomplete for our purpose since it does not consider the prerequisites of the API usage. To address this limitation, we improve PScout to automatically fix some common missing cases (§4.3). Second, we find that Java reflection is commonly used to handle remote input from open ports, which is not resolved by many static taint analysis tools such as Amandroid. To ensure the call graph completeness, we add an extra analysis to locate the target class or method, which successfully resolves over 86% Java reflection use cases in our app dataset (§4.4). Third, we find that many apps actually implement open port usage in native code, which cannot be captured by Java-layer static analysis alone. Therefore, our tool also includes native code support based on binary analysis techniques, which is commonly excluded in nearly all existing static analysis tools on Android apps due to high engineering efforts [20], [33], [34], [46](§4.2).

Using OPAnalyzer, we perform an open port usage analysis on 24K popular Android apps from Google Play, and successfully classify 99% of the usage paths into 5 categories: data sharing, proxy, remote execution, VoIP call, and PhoneGap (§5.2). We also find that significantly different from traditional usage, ports in some categories were mostly intended only for clients in physical proximity of the smartphone, or even on the same device.

Among these open port usage families, many are found to directly enable a number of serious remote exploits if not well protected. More specifically, we use OPAnalyzer to examine the security checks along the identified usage paths, and find that they generally lack sufficient protection: for the most popular usage, data sharing, over half of the paths can be easily triggered by any remote attacker, and in some usage categories such as proxy, over 80% of the paths are not protected. From OPAnalyzer output, we uncover 410 vulnerable applications with 956 potential exploits in total, and manually confirm 57 vulnerable apps that have not been previously reported, including popular ones on the market and even a pre-installed app on some device models. These newly-discovered exploits can lead to a large number of severe security and privacy breaches. for example remotely stealing sensitive data such contacts, photos, and even security credentials and performing malicious actions such as executing arbitrary code and installing malware remotely (§6). To get an initial estimate on the impact of these vulnerabilities in the wild, we performed a port scanning in our campus network, and immediately found a number of mobile devices in 2 minutes which were potentially using these vulnerable apps. we have reported these vulnerabilities to the relevant parties through vulnerability tracking systems including CVE [5] and CERT [16], and some of them have been acknowledged (*e.g.,* CVE-2016-5227, VR-176). We encourage readers to view several short attack video demos at **https://sites.google.com/site/openportsec/** [11].

Leveraging the insights from these analysis, we further categorize the vulnerable apps based on their intentions of open port, and discuss defense strategies depending on the unique characteristics in each category (§7). Specifically, for the physical proximity usage, which does not have any effective and usable protection yet, we propose a transparent socket-level solution that allows users to conveniently verify a connection from a device nearby and can be easily adopted by app developers.

We summarize the key contributions of this paper:

- We formalize open port app design pattern, and develop OPAnalyzer to systematically characterize open port usage in Android apps and detect exposed vulnerability. To ensure high accuracy, we tackle several challenges, *e.g.,* improving the API to permission mapping completeness, resolving Java reflection, and enabling native code analysis.
- Using our tool, we perform the first systematic study of open port usage and their security implications on mobile platform. We are able to classify 99% of the identified usage into 5 distinct usage families, and discover some mobile-specific scenarios. We find that nearly half of these usage paths have no protection implemented, which can directly be triggered by remote attackers to leak sensitive information and perform high-privileged actions.
- We perform an in-depth analysis on the vulnerable open port usage, and construct real exploits to validate the threats. From the results, we manually confirmed 57 new vulnerable apps containing popular ones on the market and also a pre-installed app on some device models, which can be used to remotely steal sensitive user data such as photos, security credentials, and perform malicious actions such as executing arbitrary code and installing malware. We also suggest countermeasures and improved practices to mitigate these problems in each intended open port usage scenario.

2. Background and Threat Model

In this paper, we broadly define mobile apps with open TCP or UDP ports as *open port apps*. And two types of open port apps are covered by our study. (1) *Mobile service app* provides useful functionality such as sharing files on the handset by opening a file server to be connected by user's desktop. (2) *Malicious open-port apps* intentionally open ports to carry out malicious activities such as receiving commands from remote attackers for data theft or device control. Our study does not focus on malware detection, since it's very hard to distinguish malicious and legitimate open port usage without having a comprehensive understanding of the designed functionality of each app. Instead, we focus on identifying problematic usage (including both malicious and legitimate) that exposes vulnerabilities to attacker and affects the well-being of the user.

Threat model. The threat to an app with open ports comes from the attackers with the ability to reach these ports. In the design of popular smartphone operating systems such as Android, ports are reachable from both the same device, e.g., another app or a script on the web page, and another host in the same network with the victim device. Thus, compared to the majority of previously-reported smartphone app vulnerabilities that only consider the threat from on-device malware [20], [28], [30], [50], [51], open port apps additionally face threats from network attackers, e.g., local network attacks, and web attackers, e.g., malicious scripts, which is much more diverse and also of wider range. More specifically, in this paper we consider the following *three* adversary types:

(1) Malware on the same device. A malicious app, or malware, installed by the smartphone user can use `netstat` command or proc file `/proc/<pid>/net/tcp` to find the listening ports on the same device and send exploitation traffic.

(2) Local network attacker. For victims behind NAT or using private WiFi networks, attackers sharing the same local network can use ARP scanning [4] to find reachable smartphone IP addresses at first, and then launch targeted port scanning to discover vulnerable open ports.

(3) Malicious scripts on the web. When a victim user visits an attacker-controlled website using their mobile device, malicious scripts running in the handset's browser can exploit the vulnerable open ports on the device by sending network requests, which doesn't require any permission.

For each of these three threat models, we have prepared short attack video demos on our website [11] to help readers more concretely understand their practicality.

Scope and assumptions. Our study focuses on TCP ports, which are most commonly used. We did not study UDP ports, but we argue that our methodology can be easily adapted for it. Our tool is expected to handle obfuscated Android apps as long as they can be disassembled. In the current implementation, our tool only fails to analyze very few samples (0.6% of apps in our dataset); for them, even the disassembling process cannot succeed.

3. Design Pattern of Open Port Apps

Figure 1 shows a simple example Android app that opens a port for accepting remote command to push notifications on the user's device. The app first creates a `ServerSocket` to listen on a TCP port. Once a client connects to the port, the app reads the remote input from the socket and serves the request. In this example, the app checks whether the remote input contains the "PUSH" command, and if so, it starts pushing the messages contained in the remote input to device's notification bar.

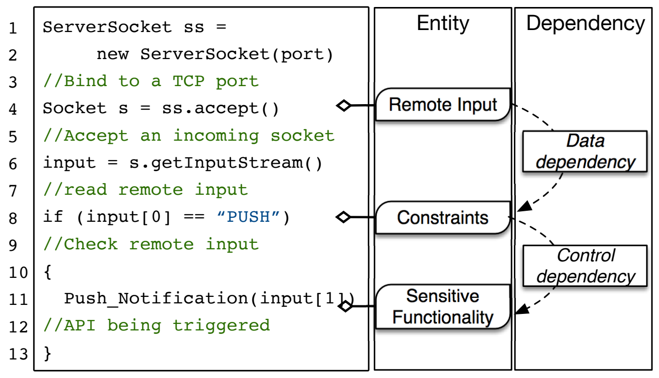

```
ServerSocket ss =
     new ServerSocket(port)
//Bind to a TCP port
Socket s = ss.accept()
//Accept an incoming socket
input = s.getInputStream()
//read remote input
if (input[0] == "PUSH")
//Check remote input
{
  Push_Notification(input[1])
//API being triggered
}
```

Figure 1: Design pattern of open port Android app

We generalize the logic of Android apps with open ports as the design pattern shown on the right side of Figure 1, consisting of three different entities and the dependency relationship among them.

Remote entry point is defined as the content passed to the open port app from the incoming sockets. And it serves as the entry point in our analysis framework (§4.1). Security mechanisms such as authentication token can be used to authorize the remote access to the app.

Sensitive functionality refers to the sensitive API set that can be triggered by remote input. The sensitive API set defined in this work contains (1) APIs protected by the Android permission system e.g., `sendTextMessage()` protected by `SEND_SMS` permission, and (2) APIs not protected by permissions but considered sensitive in the open port context. For example, an app does not require any permission to read its own data cache. However, if the data is written back to the incoming socket and transmitted to the remote client, a potential information leakage is caused, since the app cache may contain sensitive user data. We describe our approach to construct sensitive API set in §4.3.

Constraints refer to the conditional statements along the path between the remote input and the sensitive APIs. It is usually introduced by the protocols that the app uses to communicate with the remote clients. If the constraints on a path are easy to bypass, the sensitive functionality on the path may be exploitable by remote attacker to launch privilege escalation attack. We discuss more on the strength of the constraints in §4.4.

Dependency. We identify the dependency between remote input and sensitive API as the *Program Dependency*, consisting of *control dependency* and *data dependency*. We use it to describe the "*trigger*" relationship between the source and the sink, which is shown to be efficient for us to characterize open port usage and understand their security implications.

In practice, the dependency between the remote entry point and sensitive API set of an app is not easy to characterize. The analysis must handle various program flow jumps in the Android lifecycle, including inter-component communications, Java reflection, and even jumps from native code, to accurately model the open port functionality. Moreover,

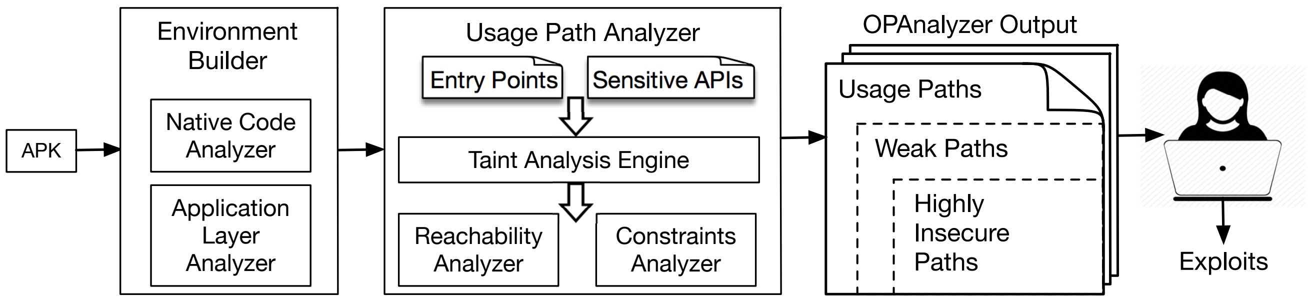


Figure 2: OPAnalyzer approach overview

to identify those vulnerable paths that can be leveraged by remote attacker, the analyzer is required to evaluate whether a given usage path is practically exploitable in terms of the timing window for attacker, and the strengths of the checks performed on the path. We design OPAnalyzer to analyze the usage and security implications of these apps to address these technical challenges.

4. OPAnalyzer Approach

The goal of OPAnalyzer is bi-fold (1) to characterize the open port usage on mobile devices, and (2) to identify vulnerability exposed by the usage. To achieve the goal, we design OPAnalyzer to automatically discover all the sensitive functionalities that can be triggered by remote input and examine the constraints that guard them. We define a ***usage path*** as a program path from the remote entry point to a single sensitive functionality with all the conditional statements along the path annotated. OPAnalyzer performs the analysis on usage path level, so that various functionality of a given open port can be comprehensively examined.

Figure 2 shows the overview of OPAnalyzer's approach. (1) OPAnalyzer takes `apk` files as input, extracts both Dalvik bytecode and native shared objects to build the environment for the app; (2) It calculates the entry points from both the native code and Dalvik bytecode for subsequent static analysis(§4.1). (3) It constructs the Inter-component Data Flow Graph (IDFG) and Data Dependency Graph (DDG) from the entry points based on Amandroid, which are both flow- and context-sensitive; (4) It performs taint analysis to study the dependency between remote input and a precomputed sensitive API set(§4.3), and outputs the paths.(5) Constraints analyzer examines all the checks along the paths that are *control dependent* on the remote input, and annotates the strength of each constraint. (6) Reachability analyzer filters those paths unreachable from the program entry set, and annotates the run-time reachability for each path(§4.4). Usage categorization and vulnerability discovery are then performed on the annotated usage paths.

In the remainder of this section, we provide an overview of the main analysis steps and how we overcome several challenges.

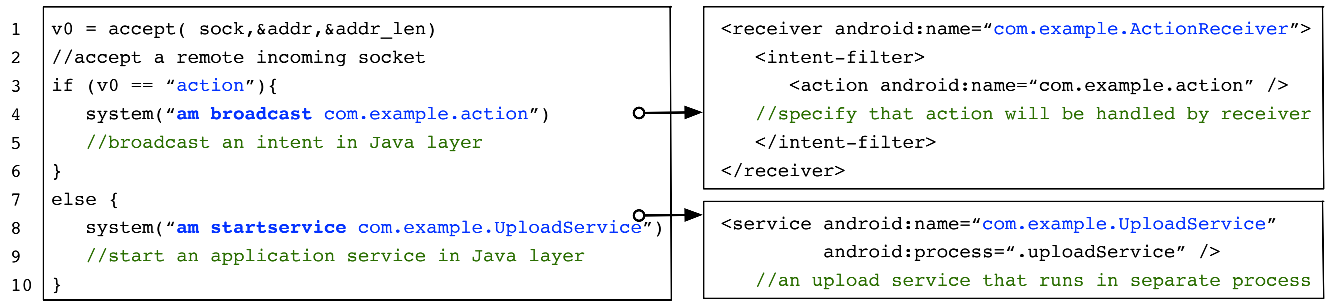
4.1. Entry Point Analysis

Entry point analysis collects the remote entry points from both Dalvik and native code. It integrates `apktool` as its front-end to decompress the apk file. Dalvik bytecodes are decoded to `smali` format and further converted to an IR called `Pilar`. The application layer analyzer extracted those Java classes that accept connections from either `ServerSocket` or `ServerSocketChannel`, which are the only Android framework APIs for apps to open TCP port in Java, as entry points. However, apps can also embed the open port functionality into the native code either for the purpose of disguising their stealth behavior or for performance reasons. Thus we implement a native code analyzer to collect those entry points embedded in native code and capture the control-flow jumps from native code to the Java layer.

4.2. Native Code Analyzer

Figure 3 is a code snippet from a real app showing the open port functionality embedded in native code. The app accepts the incoming socket in native `C` code, and passes the control-flow to application layer to serve the request. Shown on `line 4`, native code broadcasts an `intent` using the `system` function, and the `intent` is captured by the Java layer receiver and triggers the `ActionReceiver` logic. Another type of control-flow jump in this example is shown on `line 8`. The app starts a service defined in the Android manifest from native code, and the `UploadService` starts running in the background. Such cross-layer interactions cannot be captured by existing static analysis approaches, leading to inaccuracy in the security analysis. We design and implement a native code analyzer that captures such control-flow jumps based on inter-procedure taint analysis.

The native code analyzer takes the shared object files extracted from the `apk` as input, and performs taint analysis on the decompiled assembly code. The taint source is the socket accepted from the open port, while the sinks are those function calls that can initiate control-flow jump to application layer, such as `system()`. After locating the source and sink in the assembly code, it analyzes whether there is a path that propagates the taint value to the sink either explicitly or implicitly. The `system()` function calls



```
v0 = accept( sock,&addr,&addr_len)
//accept a remote incoming socket
if (v0 == "action"){
   system("am broadcast com.example.action")
   //broadcast an intent in Java layer
}
else {
   system("am startservice com.example.UploadService")
   //start an application service in Java layer
}
```

```
<receiver android:name="com.example.ActionReceiver">
   <intent-filter>
      <action android:name="com.example.action" />
   //specify that action will be handled by receiver
   </intent-filter>
</receiver>
```

```
<service android:name="com.example.UploadService"
         android:process=".uploadService" />
   //an upload service that runs in separate process
```

Figure 3: Snippet from real app showing control-flow jump from the native code layer (Left) to the application layer (Right).

in Figure 3 are not directly tainted by the `source`, but are control-dependent on the tainted conditional statement, as an example of implicit taint. The taint analysis handles inter-procedure calls, as well as asynchronous `I/O`. To handle many clients simultaneously in native code, app can perform non-blocking `I/O` using Linux's `epoll` facility [8], which provides readiness notification to simulate `I/O` multiplexing. To serve multiple requests, the thread uses `epoll_wait` to get tasks from event queue, and `epoll_ctl` to add new connections waiting to be handled into the event queue. The taint analysis propagates taint values accurately through such asynchronous function calls by keeping track of each event queue. We show how the native code analyzer improves the effectiveness of our tool in §4.5

The native code analyzer is implemented as a plug-in for `IDAPro` written in *Python*. It uses `IDAPro` [10] as the front-end, and performs the inter-procedural data-flow analysis on the CFG of the assembly code. The time it takes for analyzing an app depends on the number of shared object files contained in the app. The mean analyzing time is 80.0 seconds with the interquartile range of 74.9 seconds.

4.3. Sensitive API Selection

OPAnalyzer aims at characterizing all the sensitive functionality triggered by remote input. To define the sensitive API set and categorize their functionality, we need an accurate mapping from Android APIs to the permissions they require. However, constructing such mapping for our use case is non-trivial because our analysis is performed on the component level. We detail the challenges and our solutions below.

PScout [21] provides a static analysis approach to find the mappings between API calls and permissions. However, we find that it suffers from some completeness problems. Taking the `ServerSocket` as an example, which requires Android `Internet` permission. In the mappings generated by PScout, we only find the constructor of the `ServerSocket` mapped to the `Internet` permission, while other sensitive APIs such as `accept()` and `getOutputStream()` are missing. We suspect that it is because Android enforces some permission checks at the class level instead of API level. Although enforcing the permission check at the constructor implies that all the member functions of this class are also protected, the incompleteness of the $< API, permission >$ mappings restricts its usability for program analysis, especially when the analysis is performed on the component level. To address this problem, we design a static analysis tool to automatically add such missing APIs to the mappings. For every class constructor presented in the original PScout output, the tool takes the AOSP source code as input and extracts all the member functions of the class to complete the mappings.

In addition, we find that some APIs are not protected by Android permissions, however, when used together, are also considered sensitive in the mobile service context. For example, an app retrieves the device location and stores it in the application data cache. Once a remote connection comes in, it reads the location data from the cache and sends it to the remote attacker. In this scenario, the incoming socket does not trigger any permission check except `INTERNET`, which we think is granted by default, but the sensitive location data is stolen and leaked asynchronously. To capture such sensitive functionality in the mobile service context, we manually collect all the sources that an app can retrieve data asynchronously that do not require permission, including app cache, database, shared preference, etc. We define a pseudo permission called `DATA_LEAK`. If the data written to the incoming socket is dependent on the data retrieved from these asynchronous sources, we consider it as invoking the `DATA_LEAK` permission check. The pseudo permission together with the associated API pairs are added to the sensitive API set.

4.4. Usage Path Analysis

To identify program dependency, OPAnalyzer performs taint analysis on the DDG rooted from remote entry points, taking the remote input as `source`, and sensitive API set as `sink`. It outputs all the usage paths that the remote input can trigger, together with all the constraints that guard the `sinks`, which are useful for open port usage categorization and vulnerability discovery.

IDFG and DDG. are built for each remote entry point using Amandroid, which include all those Inter-Component Communication (ICC) edges, and are both flow- and context-sensitive. However, the control-flow jumps in-

```
1   socket = ServerSocket.accept()
2   // socket = Source#1()
3   remote_input = socket.getInputStream()
4   flag = remote_input[0]
5   if (flag == "True") {
6         password = getSharedPreferences("password",0)
7         // password = Source#2()
8         outStream = socket.getOutputStream()
9         outStream.write(password)
10        // Sink()
11  }
```

Figure 4: An example for implicit and explicit taint logic.

```
1   Socket socket = ServerSocket.accept();
2   InputStream in = socket.getInputStream();
3   String input = in.readLine();
4   String command = input.split("=")[0];
5   String value = input.split("=")[1];
6   Map map = new HashMap<String,String>();
7   map.put (command,value);
8   if (map.size() > 0){                  //C1
9         if (command == "SEND_SMS") {    //C2
10              SendSMS(value);           //Sink()
11        }
12  }
```

Figure 5: Sample app code showing that sensitive API is protected by constraints

troduced by Java reflection cannot be captured by this approach. Since we find that reflections are heavily used in our dataset, and also are found in those well-known *Wormhole* apps, we add reflection support to the taint analysis engine.

Java reflection is used by programs to examine or modify the runtime behavior of apps running in Java virtual machine. We have seen reflections used in apps for both legitimate purpose (e.g., bypass some API-level restrictions) and malicious purpose (e.g., disguise the entry to malicious code). Handling reflection accurately is impossible for static analysis, and remains challenging even combining runtime analysis [43]. To capture the control-flow jumps of reflection to our best effort, we implement a handler in the IDFG builder similar to the one used in FlowDroid [20], which can link the target class or method invoked by reflection to the calling function, and add the missing edges to the IDFG. Currently, it only handles reflection calls with targets explicitly provided in the same procedure, and will miss those whose targets are not deterministic statically. However, we find that over 86% of reflections in our dataset can be handled by applying this heuristic, and it also helps identify significantly more vulnerable paths as shown in §4.5.

Java layer taint analysis is used to examine the dependency among remote input and sensitive APIs. We define a notion to describe the explicit and implicit dependency relationships among statements. The notion is further used to categorize usage.

Notion 1. The dependency relationship between two statements along the same usage path is either implicit or explicit, and transitivity applies.

$\mathbf{stmt_1} \xrightarrow{\mathbf{E}} \mathbf{stmt_2}$: if $stmt_2$ is explicitly tainted by the return value of $stmt_1$.

$\mathbf{stmt_1} \xrightarrow{\mathbf{I}} \mathbf{stmt_2}$: if $stmt_2$ is implicitly tainted by the return value of $stmt_1$.

$\mathbf{stmt_1} \xrightarrow{\mathbf{I}} \{\mathbf{stmt_2} \xrightarrow{\mathbf{E}} \mathbf{stmt_3}\}$: if $stmt_3$ is explicitly tainted by the return value of $stmt_2$, and both $stmt_2$ and $stmt_3$ are implicitly tainted by the return value of $stmt_1$.

Explicit taint describes the data dependency relationship propagated by assignment expressions, while *implicit taint* reflects the "triggering" relationship, in which the source is presented in the conditional statements. Shown in Figure 4, the first taint source comes from the remote input, and `flag` is explicitly tainted by the `remote_input`, since it is derived from it. All the statements in the `if` block are implicitly tainted by the `flag`, since they are control-dependent on the `if` statement. Specifically, another taint source is identified as the `getSharedPreferences`, which reads data from an asynchronous source. The `password` read from the shared preferences is written to the socket, identified as an *explicit taint* relationship between statements in *line* 5 and *line* 6. Thus, the dependency relationship along this usage path is expressed as:

$$accept() \xrightarrow{I} \{getSharedPreferences() \xrightarrow{E} write()\}$$

To further narrow down the potential vulnerable path list, and provide insights to identify the vulnerability, OPAnalyzer integrates both constraints analysis and reachability analysis to help analyst prioritize paths output by OPAnalyzer to reduce human efforts.

Constraints analyzer examines all the conditional statements along the usage path to which the sensitive API is control dependent on. Shown in Figure 5, the remote input is separated into two substrings and put into a `Map`. The sensitive API `SendSMS` is control dependent on two constraints. Constraint C_1 defined in *line* 8 checks whether the `Map` is empty, while the constraint C_2 in *line* 9 checks whether the `command` passed in from remote input is "SEND_SMS". Both checks are easy to bypass, an arbitrary remote attacker can construct the input string to bypass the checks and trigger the malicious payload to be sent via SMS, which results in a remote privilege redelegation attack [30]. OPAnalyzer annotates a constraint as ***weak*** if it is either (1) comparison with constants or (2) comparison with a predefined set of trivial API (e.g., `Map.size()`, `Set.isEmpty()`). The heuristic reduces human efforts by prioritizing usage paths that are obviously easy to be controlled, but could introduce false negatives(§ 4.5).

Reachability analyzer characterizes the reachability of a path using both static and dynamic approaches. First, the static analysis models the app `Activity` lifecycle, and filters those usage paths unreachable from the program entry set. The dynamic approach identifies those usage paths that start with a port that is open by default at app launching

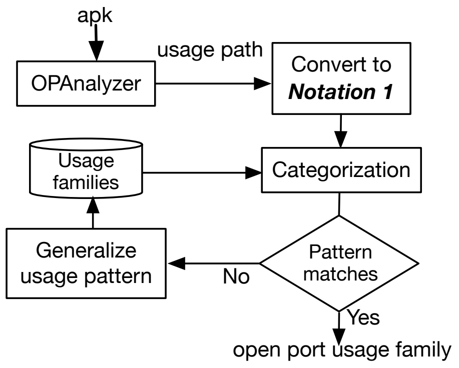


Figure 6: Usage categorization methodology

time. These paths are annotated as ***highly insecure*** since remote attacker has a large timing window to exploit them. Note that ports that are not open by default are still vulnerable to the on-device malware in our threat model, since the malware can monitor the `proc` file and exploit the vulnerable paths as long as it detects the port is open. The dynamic analysis is implemented using function hooking based on Xposed framework [17], and combined with device automation, the analyzer automatically annotates reachability results on the usage paths.

Usage categorization methodology. Shown in Figure 6, OPAnalyzer aggregates usage paths with similar open port usage into one family based on the sensitive APIs they contain and the notion. It extracts usage paths and converts them to $Notion$ 1. The categorization approach matches the notion of new paths with those already identified patterns. If an incoming path cannot be categorized into any of the existing families, we manually generalize its usage to a pattern and add a new family to the existing usage family set. Following this approach, OPAnalyzer categorizes most of the usage paths except a few that cannot be converted to $Notion$ 1(§5.2).

Vulnerability discovery methodology. Considering an attacker in our threat model, to exploit a usage path and trigger the sensitive functionality, he needs to (1) find the right timing when the port is open, (2) bypass all the checks along the path to execute the sensitive API. The usage paths output from OPAnalyzer come with the reachability information, sensitive functionality, and strengths of predicates all annotated, help human analyst efficiently examine these two prerequisites for an attack, and identify vulnerability. Specifically, OPAnalyzer prioritize "weak paths" and "highly insecure" paths for manual inspection. And note that we also selectively examined the other usages paths in the OPAnalyzer output since they may also be vulnerable to remote exploits. Some of the interesting vulnerabilities (e.g., AirDroid exploit) in our case study(§6) are actually identified in those non-weak usage paths.

4.5. Evaluation

We obtain 24,000 apps from the PlayDrone dataset [45] to evaluate our tool, which contains top 1000 popular apps from each of the 24 categories in the Google Play including entertainment, tools, etc. Our evaluation focuses on the vulnerability discovery of OPAnalyzer, which is the most security critical functionality.

Discovery accuracy. To evaluate the false positives (FPs) of the weak path detection, we first define the FPs as "weak" paths that are verified to be *not* exploitable through manual inspection. We examined the weak usage paths output by OPAnalyzer, and constructed remote input to see if the sensitive functionality could be triggered. We call these weak paths that are verified to be exploitable *vulnerable paths*, and the rest of them are considered as FPs. Among the 24,000 apps, 6.8% of them (1632) have open-port functionality, and 133 weak paths are identified to be reachable at app launching time by OPAnalyzer. We manually identified 113 vulnerable paths that can be easily exploited by constructing remote input (**FP rate 15.1%**). The FPs mainly come from paths that contain checks on the runtime property of the app. As an example, a usage path is guarded by the constraint `if (debugMode == True)` will not be triggered when the app is in release mode, but will be output falsely as weak path.

Due to the lack of any publicly accessible study of open port vulnerabilities on mobile platform, we run OPAnalyzer on the recently-reported Wormhole apps from the Chinese app market to evaluate the false negatives (FNs) of OPAnalyzer. To the best of our knowledge, they are the only reported instances that contain confirmed open port exploits, which are usable as ground truth in the FN evaluation. The FNs of the weak path detection are those paths that are not annotated as "weak", but are verified to be exploitable with our manual inspection. Wormhole apps are vulnerable due to their integrations of vulnerable SDKs including Baidu, Qihoo360, AMap, and Tencent [12], and thus we choose the most popular apps in each SDK category. We do not test on Qihoo360 library since the problem only affects an old beta version which can no longer be found in apps on major app markets.

As shown in Table 1, OPAnalyzer discovers usage paths that contain sensitive functionality in all of the three apps, with some of them reported as weak paths. Unfortunately, the report has no detailed path information that can be used as ground truth for evaluation, we do our best to manually analyze the decompiled app to find exploitable paths. For Baidu SDK, OPAnalyzer detects all the exploitable paths that we manually discovered, and even discovers new exploits that have not been reported in the report, such as stealing the WiFi `BSSID`. For AMap, OPAnalyzer reports four usage paths, while none of them are recognized as weak paths. However, three exploits (FNs) from those non-weak paths are identified. We find that one of the conditional statement shared by all 4 usage paths depends on value that is defined beyond the IDFG of the remote entry point. OPAnalyzer thus does not consider the check as "weak" since its value cannot be determined, while it turns out to be constant in the run time. It is due to the limitation of lacking of backward analysis support, so that OPAnalyzer doesn't have enough visibility into how a variable encountered on

Wormhole family	Sample app	Version	Entry points #	Usage path #	Weak path		Not weak path	
					total	exploitable	total	exploitable
Baidu	com.baidu.BaiduMap	8.5.0	1	15	8	8	7	0
Tencent	com.tencent.android.qqdownloader	5.7.0	4	16	1	1	15	1
AMap	com.autonavi.minimap	7.3.4	1	4	0	0	4	3

TABLE 1: OPAnalyzer output for three popular "Wormhole" apps that are reported vulnerable

Feature	# of usage paths captured	Improv.	Featured app/lib
none	636	N/A	WiFi file transfer
+ reflection	804	+26%	OpenVPN
+ API	1472	+131%	AMap
+ native	845	+33%	Tencent XG
+ all	1934	+204%	Baidu wormhole

TABLE 2: Evaluation of accumulative improvement brought by (1) handling explicit Java reflection and (2) adding open-port specific sensitive API (3) capturing native code jump.

the usage path is propagated to this procedure, if it is defined beyond the IDFG of the entry point. This affects the accuracy of the information leakage tracking and constraints analysis, and can be solved by integrating backward slicing technique [20], [48]. We plan to add that to the OPAnalyzer in the future.

False negatives may also be introduced by the native code analyzer due to a limitation inherited from the `IDA-PRO` front end. ARM processor supports multiple interaction sets mixed in one segment in the runtime, while the disassembler can interpret one type at the same time in the static analysis [3]. And when the 16-bit "Thumb" and 32-bit "ARM" instructions coexist within one segment due to optimization, the disassembling result may be incorrect. Additionally, the native code analysis of OPAnalyzer does not cover all possible types of interactions between native code and Java code. Combining native code and Java code analysis is a fundamental challenge of Android app analysis [38], and we leave it as future work to accurately model all the control- and data-flow transitions between native and Java layers.

We also evaluate the improvement on the effectiveness of OPAnalyzer's path discovery brought by three of our engineering efforts; namely (1) adding open-port specific APIs to the sensitive API set, (2) handling explicit Java reflections, and (3) capturing native code jump. Shown in Table 2, integrating these features greatly improves the coverage of OPAnalyzer by 204%, while completing the sensitive API set turns out to be the most effective engineering effort we spent. These improvements reduce false negatives, which is crucial to our system.

Performance. The most compute-intensive step in OPAnalyzer is the taint analysis, which includes building the IDFG and DDG, and running the Dijkstra's algorithm on the DDG to find all the usage paths. We measure the time to perform the taint analysis for the top 1000 popular apps from our dataset. The experiment runs on a machine with Intel Core i5-3470 CPU and 8GB of RAM. For apps with at least one entry point, the median processing time for OPAnalyzer to finish the taint analysis is 61.5 seconds with standard deviation of 127.2.

5. Usage and Vulnerability

With the usage paths output from OPAnalyzer, we systematically study open port usage and their security implications in the top 1000 popular apps from each of the 24 different categories on Google play.

5.1. Popularity and Permission Usage

Among the 24,000 apps, we identified open port functionality in 6.8% (1632), while 50% of these open port apps have more than 500K downloads.

Sensitive permission usage. To understand the most common functionality triggered by remote input, we use the API to permission mappings constructed in §4.3 to get the set of sensitive permissions. Figure 7 shows the top security-sensitive permissions involved in open port usage. Surprisingly, we find that in open port apps, a rich set of highly-sensitive OS-level functions in Android can be invoked remotely, ranging from accessing private data such as contacts and location to performing sensitive actions such as using camera and sending SMS. If not protected sufficiently, this usage can be remotely exploited to cause severe damages such as privacy leakage and privileged code executions, just like the recently reported Wormhole exploits [12]. In addition, we find that the pseudo permission `DATA_LEAK` (defined in §4.3), which indicates that data is read from asynchronous sources such as internal storage or content providers and sent to the remote end, has top popularity in open-port apps. This shows that open port usage commonly has potential risk of exposing internal application data to the remote attacker, which typically involves plenty of sensitive data such as credentials and conversation history [51].

5.2. Usage Family Categorization

Using OPAnalyzer, we categorize usage paths into different usage families defined by code patterns. Table 3 shows the 5 major usage families we identified in our dataset, together with the path categorization results and the types of potentially-exposed vulnerabilities. As shown, 99% of reachable usage paths of open port apps are categorized into one of the five families, which are described in detail as follows.

Data sharing path a usage path through which data read from the device is sent to the remote host. In this category,

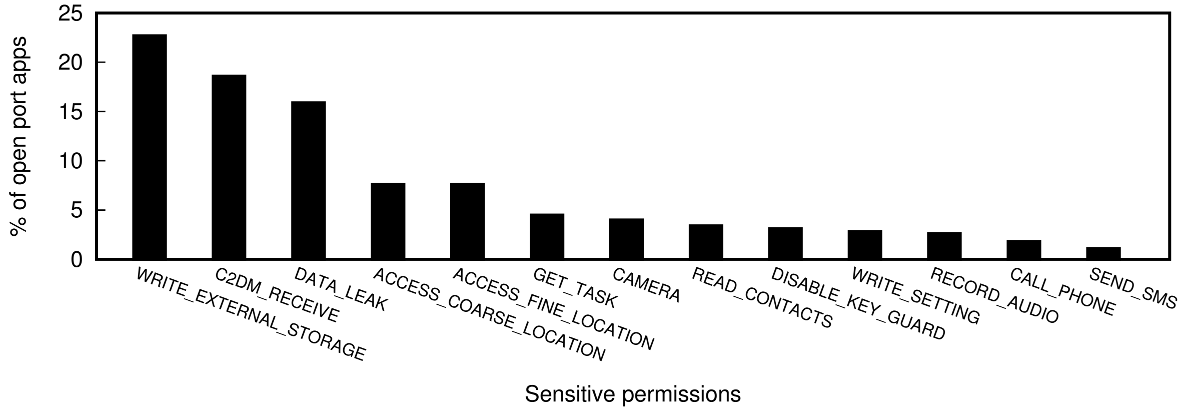


Figure 7: Permission protected APIs triggered by remote input.

Usage category	Pattern	Usage path #	Perc.	App #	Weak path #	Vulnerability
Data sharing	$\mathbf{accept}() \xrightarrow{\mathbf{I}} \{\mathbf{API_{data\_read}} \xrightarrow{\mathbf{E}} \mathbf{write}()\}$	1340	69.3%	425	775	V_1
Proxy	$\mathbf{accept}() \xrightarrow{\mathbf{E}} \mathbf{API_{out\_connection}}$	122	6.3%	59	101	V_1,V_3
Remote execution	$\mathbf{accept}() \xrightarrow{\mathbf{I\ or\ E}} \mathbf{API_{execution}}$	127	6.5%	41	69	V_2,V_3
VoIP call	$\mathbf{accept}() \xrightarrow{\mathbf{I}} \mathbf{API_{audio\_setting}}$	45	2.3%	27	11	V_3
PhoneGap	Categorized using code signature	282	14.6%	141	0	N/A
Uncategorized	N/A	18	0.9%	10	0	N/A

TABLE 3: Open port usage and potential vulnerability. V_1:sensitive data leakage, V_2: privileged remote execution, V_3: DoS.

the most commonly used protocol for data sharing is HTTP, while HTTPS, FTP, UPnP [15] and some customized protocols are also observed. As shown, nearly 60% of the paths in this category are found to be weakly protected without any client authentication, leaving them easily exploitable (examples in §6). By examining the apps in this category, we also identified a mobile-specific open port usage scenario, *data sharing in physical proximity*, e.g., allowing a user to transfer photo to her PC nearby. This usage turns to bring most exploits in this family: as shown later in §6, 24 out of 26 exploits in data sharing are associated with this particular usage. Also worth noting is that in this usage family, sensitive data can be leaked without invoking any permission-protected APIs along the tainted path. Due to our improvement in sensitive API selection (§4.3), our tool can successfully capture these cases.

Proxy path is defined as forwarding requests in remote input to other destinations. We find that all the paths in these apps are used as local proxy, e.g., for advertising and content filtering. For example, to overcome the content modification restrictions in Android `WebView`, a web browsing app starts as background service a local web proxy so that it can insert its own ads into the fetched web pages. If exposed to remote attackers, such local proxy can be used as a reflector in targeted DDoS attacks. Also, if it is configured to cache pages or store cookies, attackers can access personalized pages to harvest user privacy, or even hijack victim's email or social network accounts for spear phishing attacks.

Remote execution path refers to usage paths that can trigger certain actions on the device such as sending `SMS` and writing to storage. Besides common use cases such as push notification, we also observe interesting usage in *physical proximity*, which allows the same user to use mobile device functionality through PC interface, e.g., texting `SMS` using keyboard. However, there are also sensitive functionality that can be executed remotely and are beyond the declared functionality of the app, which we suspect to be "backdoors" left by app developers.

VoIP call paths are used in apps to listen on incoming call requests based on the Session Initiation Protocol (SIP). After accepting a SIP `invite` message from the port, the app extracts information such as caller ID and starts the ring tone to notify the user. Remote attackers in theory can send spoofed packet to ring the phone and spoof the caller ID. However, due to the IPSec support in SIP, such off-path attack is unlikely to be practical.

PhoneGap paths belong to apps developed by PhoneGap/Cordova, a hybrid app development framework allowing developers to quickly build apps using JavaScript and HTML5. It uses open ports to serve requests from the JavaScript client and handle the API calls. The result written to the incoming socket is not defined in the IDFG of the remote entry point, making it difficult for OPAnalyzer to capture sensitive APIs. To address this, we use the presence of several PhoneGap-specific classes such as `CallbackServer` as the code signatures to identify these usage paths. The port intended for IPC is falsely opened to the Internet. However, we find that the usage paths of PhoneGap are protected by strong security checks, which verifies whether the request contains a 128-bit token derived

Open port intention	Vulnerability description	Featured attacks	Vulnerable app example	App #[1]
Usage of the app user	Lack of authentication to verify that connections come from the app user	Data theft	WiFi file transfer	24
		Privilege escalation	AirDroid, PhonePal	10
Communication with backend	Lack of authentication to verify the requests are from authentic app backend server	Data theft	Lenjoy	15
		Privileged escalation	KindeeExpress	5
Local communication	Port used for on-device communication falsely opened to the network	DoS	VGet, Fast secure VPN	12
		Data theft	CachedProxy	1

[1] Number of vulnerable apps in the same category. An app may be vulnerable to both attacks in each category.

TABLE 4: Case study of verified exploitable app categorized by the intended usage of open port.

from the device's Universally Unique Identifier (UUID) [1]. Thus, we consider the open service of PhoneGap as well-protected.

5.3. Security Implications

Usage paths in different families, if not well protected, can lead to different security breaches. As shown in table 3, OPAnalyzer outputs 956 weak paths. We find that nearly half of the total usage paths are considered "weak". In the proxy category, over 80% of the paths are not protected. From these weak paths, we identify three vulnerability categories: sensitive data leakage (V_1), privileged remote execution (V_2), and DoS (V_3). Besides, we also discover a new problem that any open port on smartphones can be exploited to harvest cellular IPv6 addresses which allows attacker to collect victim IPs without scanning the huge IPv6 address space and further launch attack to exploit vulnerable ports. The vulnerability is detailed in the *Appendix A*.

V1: Sensitive data leakage. Sensitive data of a mobile device can be retrieved from many sources such as SD Card, sensor, etc. If these paths are not well protected, remote attacker can exploit them to steal sensitive data that are even protected by Android permission or *UNIX* `uid/gid` check. More importantly, if the victim IP is public, such vulnerabilities can be easily revealed using fast Internet-wide scanning tools such as ZMap [26], causing large scale data theft.

V2: Privileged remote execution. Vulnerable paths that trigger native actions can be leveraged by remote attackers to execute privileged functionality such as sending `SMS` and modifying contacts. Moreover, by exploiting the broadcasting `Intent` mechanism provided by Android, attackers can even execute functionality beyond the vulnerable app. For example, an unprotected usage path that sends `Intent` based on remote input can be leveraged to launch `YouTube` app to play the video from the URL passed by the remote attacker. By uploading a maliciously crafted `MP4` file to the URL, attacker can gain full control of the device exploiting the *Stagefright* vulnerability [6].

V3: Denial of service. Most remote execution paths are vulnerable to DoS attack against the device user. For the local proxy usage paths, they can also be used by attackers as reflectors in targeted DDoS attack against victims in the Internet to hidden their IP addresses. Other security problems of open proxies, such as leaking internal network data and IP spoofing based attack also apply.

6. Exploits Case Studies

To broaden the scope of our vulnerable study, and discover more exploitable open port usage, we extended our data set to include all the 78K apps from the `Tools` category of the PlayDrone dataset to our vulnerability analysis, based on the observation that the percentage of open-port apps in the `Tools`(10.9%) is significantly higher than the average (6.8%). Furthermore, we also crawled the top 3,000 most popular apps from a Chinese app market [2], where the Wormhole problem was reported from.

By analyzing the annotated usage paths from the OPAnalyzer output, we successfully discover several new exploits of sensitive data leakage and privileged remote execution in both apps and third-party libraries, including some high-profile ones with millions of downloads and even pre-installed apps. Moreover, we classify these vulnerable apps into 3 categories based on the *intended usage scenario* of the open port inferred from the manual analysis: (1) intended for use by app users; (2) intended for communication with the backend; and (3) intended for local communication. Such categorization helps better understand the challenges in securing the port opened for different purposes. Table 4 shows an overview of the 57 exploitable apps that are manually verified from the OPAnalyzer output. A case study of interesting vulnerability in each category is presented below. The video demos for some of the implemented attacks are shown on our website [11].

6.1. Intended for Use by App Users

Such apps open ports for different purposes intended for the app users, such as transferring file from the phone to another device of the same user. However, essential checks are found missing in many apps in this category, thus exposing the sensitive data and also privileged functionality of the device to attackers.

Virtual data cable is a popular app on China market that helps user transfer their photos to PC by opening a web server on the phone. The server port opens by default at app launch time and silently runs in the background. It does not authenticate clients nor notify incoming connections, thus can be easily scanned and exploited by remote attackers. Moreover, it does not check the requested file path, so that attacker can access files beyond the photo folder on SD card by adding "**../**" to the path and steal sensitive data from app cache and system directory. Similarly, a popular file sharing app *WiFi file transfer* with 10 million installs does not authenticate clients, while it opens the server port only when user presses a toggle button. However, an on-device malware that only has `Internet` permission in our threat model can listen on the status of the port by monitoring the `/proc` file system and steal data from the port as long as it is open.

PhonePal allows a user to remotely control his/her device with a web-browser, which contains highly sensitive usage paths such as open URLs in the Android browser, and open videos in the YouTube app. All these usage paths are found unprotected, which puts the device at the risk of phishing attack and even compromise [6]. Moreover, vulnerabilities such as allowing attacker to remotely install apps on the victim device, are identified in the high-profile app *AirDroid*, pre-installed on Samsung Chromebook and Smartisan phone [13]. We present a case study of this app in the mitigation strategy section (§7).

6.2. Intended for Communication with Backend

Open ports in these apps are intended to communicate with the app developer's backend server for various purposes. If the open port service does not authenticate the identity of the remote server, attackers can spoof the app server. Interestingly, by manually examining the apps, we find that open port usage of some apps are beyond the apps' declared functionality, implying potentially covert malicious behavior.

KindeExpress is the most popular mail/package tracking app on a China market with 1 million downloads, whose functionality is to provide tracking information from many delivery service providers. One of its usage path is able to start an `Activity` of the app and display the data from remote input on the app's UI, which also brings the app to the front even when it is running in the background. We verified that none of the declared functionality of the app depends on this usage path, and we suspect it to be used for advertising. Unfortunately, this path is not protected by any authentication mechanism, and remote attacker can send command to the app pretending as it comes from the app server to display deceptive content on the app UI.

Huang CheatMaker is a game modifier app that helps users cheat when playing mobile games. We identified a usage path that accepts data pushed from the app server, stores the data as a shared object (`.so`) file and loads it at run-time. The authentication along the path is weak, and the app dynamically loads the code without verifying where it comes from nor its integrity, which enables remote attackers to inject malicious payload to the app to that will be executed, thus compromising the device.

6.3. Intended for Local Communication

Usage paths in Proxy and PhoneGap families are intended for on-device communication, which can be either IPC among different components of an app, or proxy for different apps on the device to use. However, open ports in some apps that are intended for local usage are falsely exposed to the network, and thus lead to security breaches.

OpenVPN is an open-source VPN implementation that is integrated by several popular VPN apps on Google Play. Multiple interfaces are provided by OpenVPN for the app to configure the local VPN settings, while the open TCP port interface is identified to be the least secure one. A remote attacker can DoS the victim user by changing proxy settings such as port number on the device. Fortunately, some app developers paid extra attention when integrating OpenVPN, and closed the insecure configuration interface from the open TCP port, but VPN apps vulnerable to this problem still remain (e.g., *Fast secure VPN*).

CacheProxy is an app that opens local proxy with the capability of caching the web page content. Upon receiving a request, the proxy first checks whether the request can be served using cached content before fetching the page, with no authentication performed on the source of the incoming request. Remote attackers can thus easily access sensitive information of the victim, such as e-mails by requesting the e-mail page, since the page is retrieved from the cache for the attacker.

To get an initial estimate on the severity of open port vulnerabilities in the wild, we performed a small scale port scan in a subnet of a campus network. The ports scanned are those opened by the most popular vulnerable apps in our dataset, whose port number needs to be static and unique. Note that we only scanned for the existence of the open ports but did not send any data to the ports to verify the vulnerability for ethical concerns. We performed only one scan using a scanning tool [26], which finished in two minutes. Surprisingly, 40 hosts identified to be mobile devices open such ports. Although different apps that use the same port number may introduce false positives, the scanning result indicates that immediately exploitable open ports exist in the wild.

7. Mitigation Strategy

Traditional solutions to protect an open port from Internet attackers are through firewall, which monitors and controls incoming and outgoing traffic based on predetermined security policies. However, the firewall solution suffers from usability in the mobile context, since it is hard for individual users to configure suitable firewall rules for each app installed on the device, and coordinate both app functionality and security assurance. Moreover, in the physical proximity use scenario, since users can initiate connections from arbitrary hosts, it is hard to configure rules in advance.

Despite a variety of open port usage described, the fundamental problem is the lack of proper client authentications. However, we find that it is non-trivial to provide a general solution to patch the security problems for all usage cases, while preserving usability. We discuss the major challenges in different scenarios and propose countermeasures.

Intended for communication with backend. For the open port mobile app to verify that the incoming connection is from the authentic server, we suggest using secure tokens to perform authentication. Although tokens are already used in some open port apps, implementation flaws are commonly seen. For example, a third-party push notification service that distributes token to app developers for them to embed in the app is vulnerable, because the token can be extracted from the released app binary by attacker to exploit the vulnerable app installed on other victim devices. We suggest the app and server negotiate a shared token using mechanisms such as Diffie-Hellman [7] at app launching time. And open port app uses the token to authenticate further incoming connections.

Intended for use by app users. Compared to the previous usage scenario, open port apps in this case do not know in advance the legitimate remote hosts that should connect to them. Depending on the usage of the app user, the trusted remote hosts can be user's desktop or even his friend's laptop. A general solution is to use password or pin code to authenticate incoming connections; however, some security issues are raised in practice. As an example, all the apps we examined that use password, provide hard-code default password that does not require users to change before using the app, leading to the potential use of the default password and degrading app security. Randomly generated pin codes in the open port apps we examined are usually no longer than 4 characters, which is trivial to enumerate. Experiment in our local WiFi network indicates that it takes less than 5 minutes to send probing requests that enumerate all the 4-character strings.

Another authentication solution adopted by the top trending apps for this usage scenario is the incoming connection notification, which pops up a window when a new host connects to the open port and displays the IP address of the host. And the request is not served until user explicitly accepts the client by clicking the "allow" button. For ordinary users, the timing of the pop-up window is also an important indicator for them to make the decision of whether to allow or deny the request. However, on-device malware can infer the timing when there is an incoming connection to the port by monitoring `/proc/net/[tcp][tcp6]`, and send request immediately to trick user into also allowing its connection by overlaying the pop-up window. We further find that even the most popular apps in this category has implementation flaws that can be practically exploited by attackers in our threat model.

AirDroid[1] is a top-ranked app on the market that allows users to access and manage their Android device wirelessly from desktop by opening a server on the phone. It provides a rich set of functionality to users such as access camera and install apps remotely, and uses the incoming connection notification schema to authenticate client. However, if the timing of user initiated connection is inferred by the attacker with the help of an on-device malware, and attacker sends request to the open port before the user accepts the previous connection, the app won't pop up another window. Instead, the attacker connection silently replaces the previous legitimate connection in the waiting queue, without changing the IP address displayed on the pop-up window. And when a user clicks the button to allow the legitimate connection, the attacker client is allowed instead, and numerous sensitive capabilities of AirDroid are granted to the attacker.

1. AirDroid: https://www.airdroid.com/

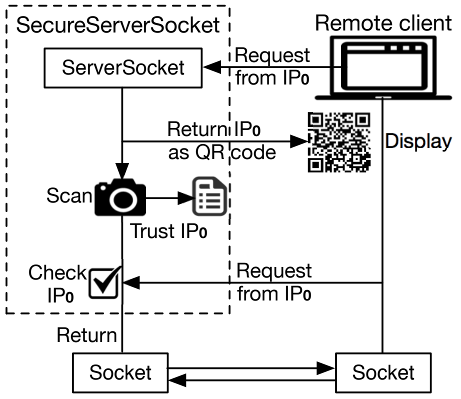


Figure 8: SecureServerSocket design

Usage in this category is usually for the cable-less communication with nearby hosts of the user. We confirmed that 24 out of the 26 vulnerable apps in this category are intended for the use in physical proximity. We demonstrate a transparent socket-level solution that addresses the security and usability challenges in this usage scenario. Shown in Figure 8, we provide `SecureServerSocket`, which encapsulates the Android `ServerSocket` API to accept incoming sockets. When a remote client, the user's desktop for example, tries to connect to the port, the `SecureServerSocket` first puts the connection on hold, and then encodes the IP address of the client into a QR code and returns to the remote client. The remote client is required to display the QR code and let the user scan it using the mobile device in order to get access.

Once the QR code is scanned, the decoded IP address is returned to the `SecureServerSocket` and the connection from this IP is allowed. It then returns the incoming socket to the upper application layer to be handled. This approach authenticates clients on the IP layer, and ensures that the open port only serves clients in physical proximity of the device user, and it achieves both security and usability. We provide `SecureServerSocket` as a library that app developers can simply use as a replacement of `ServerSocket`. No changes on the client side are required if the client is a web browser, which is the common case in our app dataset. For other client types that cannot display QR code, we suggest using one-time pin code to do the authentication instead. Demo of the SecureServerSocket implementation can also be found on our website [11]. And for future work, we plan to conduct a user study to evaluate the effectiveness of this approach.

Intended for local communication. For the local usage of on-device proxy, the best practice is to bind the proxy to the loopback address of the device, which makes the service unreachable from the network. For another local usage scenario where different components of an app communicate using open port, we suggest using other IPC mechanisms, such as `Intent`, `Binder`, and `LocalSocket` instead. And application layer authentications are required when using them. For example, `uid/gid` check should be enabled when using `LocalSocket` to ensure that the connections are from the same app.

8. Related Work

Security implications of open port usage. The security implications of using open ports on the network services have been studied using Internet-wide scanning tools such as ZMap [26], revealing various vulnerabilities [9], [14], [27], [41]. However, the open port usage and security concerns on mobile platform remain under-explored. Understanding this problem in the mobile context is non-trivial, since both the current usage and the existing defense solutions are not applicable to the mobile scenario. We design and implement OPAnalyzer to bridge this gap.

Static analysis on Android. Static analysis has been used extensively in vulnerability discoveries. Specifically, on Android platform, many tools have been built to identify system vulnerability [18], [23], [39], [42], [44] and malicious apps [20], [22], [28], [29], [32], [40], [46]. Among these work, TriggerScope [32] is most closely related to our work, which focuses on detecting malicious activities embedded in narrow conditions using static analysis. OPAnalyzer serves a different goal, which is detecting open port related vulnerabilities. However, the *trigger analysis* proposed by TriggerScope can be potentially integrated by OPAnalyzer to improve its accuracy of weak constraints analysis. FlowDroid [20] and Amandroid [46] are two static analysis tools similar to OPAnalyzer, both of which model the Android `Activity` lifecycle [20], and capture inter-component communications. Compared to these generic app analysis tools that evaluate the app as a whole, our tool focuses on examining the part that contains open port functionality. Another difference is that we integrate native code analysis for high accuracy and coverage of our analysis, which is commonly excluded in these tools due to high engineering effort.

Android app security. Mobile app security issues have gained much attention recently, and research efforts were made on detecting repackaged apps [19], [22], [36], [50], apps with malicious behavior [24], [31], [32], [35], [47], [52], or apps with vulnerability [25], [30], [37]. Different from these prior studies, we investigate the vulnerability in open port apps, which is not covered by related work, and has a new threat model not previously explored. Our analysis results shed important light on the common design and implementation flaws in these apps, and we also propose solutions to some mobile-specific usage scenarios.

9. Conclusion

In this paper, we develop a tool called OPAnalyzer, which can systematically characterize open port usage in Android apps and effectively detect exploitable vulnerabilities. Using this tool on 24K popular Android apps, we are able to classify 99% of the mobile usage into 5 families, and identify some unique usage scenarios on mobile platform. From the vulnerability analysis performed, we find that such usage is generally unprotected. We are able to discover a bunch of new exploits causing vulnerabilities such as information leakage, denial of service, and privileged execution. We also propose countermeasures and improved practices to mitigate these problems in different usage scenarios. As a potential future work, we want to apply OPAnalyzer to analyze Android system applications to discover more critical vulnerabilities.

References

[1] Android Universally Unique Identifier. http://developer.android.com/reference/java/util/UUID.html.

[2] Anzhi app market. http://www.anzhi.com.

[3] ARM processor specifications. https://www.hex-rays.com/products/ida/support/idadoc/1350.shtml.

[4] Arp-scan. http://linux.die.net/man/1/arp-scan.

[5] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[6] CVE-2015-6602. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6602.

[7] Diffie-Hellman Key Agreement. https://www.ietf.org/rfc/rfc2631.txt.

[8] Epoll I/O notification. http:linux.die.net/man/4/epoll.

[9] Heartbleed. http://heartbleed.com/.

[10] IDA-Pro. https://www.hex-rays.com/index.shtml.

[11] Mobile Open Port Security Project. https://sites.google.com/site/openportsec.

[12] One hundred days before and after Baidu Wormhole's Discovery. http://www.inforsec.org/wp/wp-content/uploads/2016/01/\wormhole_external_final.pdf.

[13] Smartisan technology. http://www.smartisan.com.

[14] The Conficker Worm. https://www2.sans.org/security-resources/malwarefaq/conficker-worm.php.

[15] UPnP: Universal Plug and Play. https://tools.ietf.org/html/rfc6970.

[16] US-CERT. https://www.us-cert.gov/.

[17] Xposed Module Repository. http://repo.xposed.info/.

[18] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *ACM CCS*. ACM, 2015.

[19] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. Sok: Lessons learned from android security research for appified software platforms. 2016.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*. ACM, 2014.

[21] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.

[22] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[23] Q. A. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. M. Mao. Static Detection of Packet Injection Vulnerabilities – A Case for Identifying Attacker-controlled Implicit Information Leaks. In *ACM CCS*, 2015.

[24] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, 2014.

[25] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *ACM Mobisys*, 2011.

[26] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[27] W. M. Eddy. TCP SYN Flooding Attacks and Common Mitigations. rfc4987, 2007.

[28] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information Flow Tracking System for Real-Time Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[29] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM CCS*, 2012.

[30] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.

[31] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android UI Deception Revisited: Attacks and Defenses. In *FC*, 2016.

[32] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *IEEE Security & Privacy*, 2016.

[33] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[34] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[35] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[36] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *TRUST*. 2013.

[37] Y. Z. X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.

[38] P. Lantz and B. Johansson. Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications. In *Wireless Communications and Mobile Computing Conference*, 2015.

[39] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[40] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM CCS*, 2012.

[41] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security & Privacy*, 2003.

[42] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[43] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ACM ASIACCS*, 2013.

[44] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*, 2016.

[45] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, 2014.

[46] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM CCS*, 2014.

[47] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE, 2015.

[48] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In *CCS*, 2010.

[49] L. Zhang, M. Gordon, R. Dick, Z. M. Mao, P. Dinda, and L. Yang. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *Proc. of International Conference on Hardware-Software Codesign and System Synthesis*, 2012.

[50] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[51] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS*, 2013.

[52] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.