Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment

Daoyuan Wu[1], Debin Gao[1], Rocky K. C. Chang[2], En He[3], Eric K. T. Cheng[2], and Robert H. Deng[1]

[1]Singapore Management University
[2]The Hong Kong Polytechnic University
[3]China Electronic Technology Cyber Security Co., Ltd.

Abstract—**Open TCP/UDP ports are traditionally used by servers to provide application services, but they are also found in many Android apps. In this paper, we present the first open-port analysis pipeline, covering the discovery, diagnosis, and security assessment, to systematically understand open ports in Android apps and their threats. We design and deploy a novel on-device crowdsourcing app and its server-side analytic engine to continuously monitor open ports in the wild. Over a period of ten months, we have collected over 40 million port monitoring records from 3,293 users in 136 countries worldwide, which allow us to observe the actual execution of open ports in 925 popular apps and 725 built-in system apps. The crowdsourcing also provides us a more accurate view of the pervasiveness of open ports in Android apps at 15.3%, much higher than the previous estimation of 6.8%. We also develop a new static diagnostic tool to reveal that 61.8% of the open-port apps are solely due to embedded SDKs, and 20.7% suffer from insecure API usages. Finally, we perform three security assessments of open ports: (i) vulnerability analysis revealing five vulnerability patterns in open ports of popular apps, e.g., Instagram, Samsung Gear, Skype, and the widely-embedded Facebook SDK, (ii) inter-device connectivity measurement in 224 cellular networks and 2,181 WiFi networks through crowdsourced network scans, and (iii) experimental demonstration of effective denial-of-service attacks against mobile open ports.**

I. INTRODUCTION

A network port is an abstraction of a communication point. Servers on the Internet offer their services by "opening" a port for clients to send requests to, e.g., web servers on TCP port 80. A TCP/UDP port is regarded as open if a server process listens for incoming packets destined to the port and potentially responds to them. Since mobile devices are generally not suitable for providing network services due to their non-routable addresses and lack of CPU and bandwidth resources, one may argue that mobile apps are not suitable for hosting open ports. However, a few recent studies have shown otherwise and these open ports are susceptible to various attacks. Lin et al. [57] demonstrated the insecurity of local TCP open ports used in non-rooted Android screenshot apps. Wu et al. [79] found that the top ten file-sharing apps on Android and iOS typically do not authenticate traffic to their ports. Bai et al. [83] further revealed the insecurity of Apple ZeroConf techniques that are powered by ports such as 5353 for mDNS.

Beyond these manual studies on specific apps, Jia et al. [52] recently developed a static tool OPAnalyzer to identify TCP open ports and detect vulnerable ones in Android apps. They identified potential open ports in 6.8% of the top 24,000 Android apps, among which around 400 apps were likely vulnerable and 57 were manually confirmed. Nevertheless, OPAnalyzer still suffers from the inherent limitation of static analysis (i.e., the code detected might not execute) and the incapability of typical Android static analysis to handle dynamic code loading [65], [67], complex implicit flows [43], [66], and advanced code obfuscation [46], [78]. Moreover, the focus of OPAnalyzer is about detecting permission-misuse-related vulnerabilities in TCP open ports (via pre-selected sink APIs), while the entire picture of open ports in the Android ecosystem is still largely unexplored.

In this paper, we aim to systematically understand open ports in Android apps and their threats by proposing the first analysis *pipeline* that covers the open port discovery, diagnosis, and security assessment. The key of this pipeline is to employ crowdsourcing, instead of static analysis, for the open port discovery, and use static analysis only for the diagnosis of discovered open ports. As shown in Fig. 1, our pipeline first adopts a novel crowdsourcing approach to continuously monitor open ports in the wild, and then employs static analysis to collect and diagnose the code-level information of discovered open ports. It also performs three security assessments: vulnerability analysis, inter-device connectivity measurement, and denial-of-service attack evaluation. We further elaborate our contributions as follows.

First, we design and deploy the first crowdsourcing platform (an on-device monitoring app and a server-side analytic engine) to continuously monitor open-port apps without user intervention, and show that such a crowdsourcing approach is more effective than static analysis in open port discovery. Our Android app, NetMon[1], has been available on Google Play for an IRB-approved crowdsourcing study since October 2016. It is still an on-going deployment cumulatively with 6K+ installs. In this paper, we base our analysis on the data over ten months (a period when most of our evaluations were performed and security findings were confirmed), which already generates a large number of port monitoring records (over 40 million) from a wide spectrum of users (3,293 phones from 136 countries). It enables us to observe the actual open ports in execution on 2,778 Android apps, including 925 popular ones from Google Play and 725 built-in apps pre-installed by over 20

---

[1]NetMon is short for "Network Scanner & Port Monitor" and is available at https://play.google.com/store/apps/details?id=com.netmon.
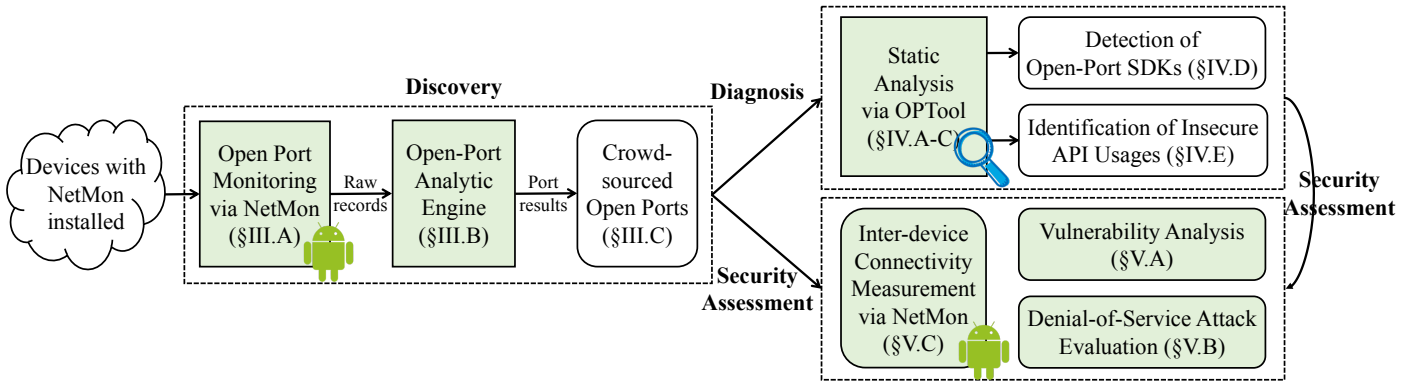
Fig. 1: The workflow of our open-port analysis pipeline (methodology shown in colored blocks and results shown in rounded blocks).

phone manufacturers. Besides the built-in apps missed by OPAnalyzer, NetMon also covers both TCP and UDP ports.

We further quantify the efficacy of crowdsourcing through a comparison with static analysis. Out of the 1,027 apps that are confirmed with TCP open ports by our crowdsourcing, 25.1% of them use dynamic or obfuscated codes for open ports, and only 58.9% can be detected by typical Android static analysis techniques. With the help of NetMon, we manage to quantify the pervasiveness of open ports in a controlled set of the top 3,216 apps from Google Play, and find TCP open ports in 492 of them. This level of pervasiveness (15.3%) is more than twice previously reported (6.8%) using static analysis [52]. Moreover, we are the first to measure the distribution of open-port apps across all 33 Google Play categories.

While crowdsourcing is effective in port discovery, it does not reveal the code-level information for more in-depth understanding and diagnosis. As the second contribution, we include a diagnosis phase through OPTool, a new static analysis tool enhanced with open-port context and semantics, to understand the code-level open port constructions and the corresponding security implications. We focus on two kinds of diagnoses: whether an open port is introduced by developers themselves or embedded via a third-party SDK (Software Development Kit) by default, and whether developers apply secure open-port coding practice. The detection results are quite alarming. First, 13 popular SDKs are identified with open ports and 61.8% of open-port apps are solely due to these SDKs, among which Facebook SDK is the major contributor. Second, 20.7% of the open-port apps make convenient but insecure API calls, unnecessarily increasing their attack surfaces.

In the last phase of our pipeline, we perform three novel security assessments of open ports:

Vulnerability analysis. Unlike OPAnalyzer which concentrates on the pre-defined vulnerability pattern, our vulnerability analysis aims to identify popular apps' vulnerabilities that may not contain a fixed pattern — therefore more difficult to detect. The five vulnerability patterns identified by us present themselves in apps, such as Instagram, Samsung Gear, Skype, and the widely-embedded Facebook SDK.

Denial-of-service attack evaluation. We experimentally evaluate the effectiveness of a generic denial-of-service (DoS) attack against mobile open ports. We show that DoS attacks can significantly and effectively downgrade YouTube's video streaming, WeChat's voice call, and AirDroid's file transmission via their open ports.

Inter-device connectivity measurement. Remote open-port attacks require the victim device to be connected (intra- or inter-network). To measure the extent to which this requirement is satisfied, we extend NetMon to conduct inter-device connectivity tests. With 6,391 network scan traces collected from devices in 224 cellular networks and 2,181 WiFi networks worldwide, we find that 49.6% of the cellular networks and 83.6% of the WiFi networks allow devices to directly connect to each other in the same network. Furthermore, 23 cellular networks and 10 WiFi networks assign public IP addresses to their users, which allows inter-network connectivity from the Internet.

II. BACKGROUND AND THREAT MODEL

Before presenting our analysis pipeline, we first introduce the necessary background and our threat model.

An open port, in this paper, is defined as a TCP/UDP port that binds to any legitimate IP address and is configured to accept packets. Legitimate IP address includes public, private, any (0.0.0.0), and also the local loopback IP address. We use such a generalized definition primarily due to the threat model in smartphones — any third-party apps running on the phone could be untrusted and could utilize even the local loopback address for attacks. To make it simple, we use *host IP address* to refer to all IP addresses except the loopback IP address, which will be explicitly stated. Under such a convention, a *local open port* refers to one that binds to the loopback address.

Open ports on Android are typically created using TCP stream or UDP datagram sockets. BluetoothSocket [10] (in Android SDK), NFCSocket [26] (an open-source library), and in particular, the previously studied UNIX domain socket [71] are out of our scope because they do not use network ports. For example, Unix domain sockets use file system as their address name space, and therefore there are no IP addresses and port numbers. The communication also occurs entirely within the operating system between processes.

We consider three types of adversaries in our threat model:

- A *local* adversary is an attack app installed on the device on which the victim app (with open ports) runs. Such a adversary does not require sensitive permissions but needs the INTERNET permission to access the open ports.
- A *remote* adversary resides in the same WiFi or cellular network to which the victim device connects. Such an

adversary can send TCP/UDP packets to other nodes if the network provides intra-network connectivity or even inter-network connectivity (with public IP addresses assigned to clients), surprisingly true for numerous networks as we will show in Sec. V-C.

- A *web* adversary remotely exploits a victim's open ports by enticing the victim to browse a JavaScript-enabled web page under the adversary's control. This threat is only applicable to HTTP-based ports with a fixed port number, because (i) JavaScript and WebSocket can issue only HTTP packets, and (ii) the resource constraint makes it infeasible for a web page to iterate the ephemeral port range [16] according to our test.

Note that local open ports could be attacked only by the first and the third adversaries, while other open ports may suffer from all three adversaries.

III. DISCOVERY VIA CROWDSOURCING

The first phase of our pipeline is to discover open ports. Instead of using static analysis as in [52], we propose the first crowdsourcing approach for the discovery of open ports. It has the following *unique* advantages: (i) it can monitor open ports in the wild, covering not only third-party apps but also built-in apps that are usually difficult to analyze due to the heavy Android fragmentation [5]; (ii) it results in no false positive; (iii) it captures the exact port number and IP address used as well as their timestamps; and (iv) it covers both TCP and UDP ports. Furthermore, as to be evaluated in Sec. III-C3, our crowdsourcing is much more effective in terms of port discovery than typical Android static analysis, which cannot handle dynamic code loading [65], [67], complex implicit flows [43], [66], and advanced code obfuscation [46], [78].

Our crowdsourcing platform consists of an on-device port monitoring app NetMon (Sec. III-A) and a server-side open-port analytic engine (Sec. III-B). We have deployed NetMon to Google Play and collected the crowdsourcing results from a large number of real users (Sec. III-C). Before moving to the technical details, it is worth highlighting the overall challenges in our crowdsourcing approach. The development of NetMon requires us to handle many product-level issues for a long-term and user-friendly deployment, let alone we are the first to explore on-device crowdsourcing for monitoring other open-port apps in real user devices. Moreover, compared to the typical app-based crowdsourcing (e.g., Netalyzer [75], MopEye [80], and Haystack [69]), our open-port crowdsourcing is unique in that the collected raw records cannot be directly analyzed due to the existence of *random* port numbers. We thus need to design an "intelligent" analytic engine that can effectively cluster raw records into per-app open port results.

A. On-device Open Port Monitoring

Different from ZMap [53] and Nmap [27] that probe ports by externally sending network traffic, we launch *on-device* port monitoring directly on crowdsourced devices to collect not only open port numbers but also their app information. Fig. 2 shows two NetMon user interfaces for port monitoring. Fig. 2(a) shows a partial list of apps running with open ports, while Fig. 2(b) shows the detailed records for a specific app (YouTube), including the TCP/UDP port numbers, IP addresses to which the ports bind, and the timestamps.

	NETWORK SCAN	PORT MONITOR				YouTube's raw open-port records:		
App Icon	App Name (# of port records)	TCP Ports	UDP Ports	Last Seen	Type	Port	IP address	Time
	Nearby Service 2421 total records	8187	3942 1900	11:47 Jan 9	UDP4	2708	192.168.1.184	00:24 Jan 9
	Messenger 3401 total records	52610 etc.	N/A	11:42 Jan 9	UDP4	44818	192.168.1.184	00:24 Jan 9
	Pages Manager 201 total records	40672 etc.	N/A	11:42 Jan 9	TCP6	43976	127.0.0.1	00:24 Jan 9
	WhatsApp 67 total records	N/A	58560 etc.	11:02 Jan 9	UDP4	2708	192.168.1.184	00:18 Jan 9
	YouTube 2004 total records	57885 etc.	48639 etc.	10:32 Jan 9	UDP4	44818	192.168.1.184	00:18 Jan 9
	DU Cleaner 27 total records	52433	N/A	23:53 Jan 8	TCP6	43976	127.0.0.1	00:18 Jan 9
	WeChat 194 total records	N/A	47463 etc.	23:43 Jan 8	UDP4	2708	192.168.1.184	00:13 Jan 9
					UDP4	44818	192.168.1.184	00:13 Jan 9

(a) A sample of open-port apps. (b) Detailed records for YouTube.

Fig. 2: User interfaces in NetMon showing open ports.

Port monitoring mechanism. NetMon leverages a public interface in the `proc` file system [29] to monitor open ports created by all apps on the device. The four pseudo files under the `/proc/net/` directory (i.e., `/proc/net/tcp|tcp6|udp|udp6`) serve as a *real-time* interface to the TCP and UDP socket tables in the kernel space. Each pseudo file contains a list of *current* socket entries, including both client and server sockets. Any Android app can access these pseudo files without explicit permissions, and this works on all Android versions including the latest Android 9. By using such an interface, NetMon can obtain the following port-related information:

- *Socket address.* It covers a port number and an IP address.
- *TCP socket state.* There are 12 possible TCP states [34], such as LISTEN and ESTABLISHED.
- *The app UID.* Using the `PackageManager` APIs, NetMon obtains the app's name from its UID (user ID).

According to the definition in Sec. II, NetMon considers server ports as open ports. Therefore, it identifies a TCP open port from the `proc` file when it is in the `LISTEN` state. On the other hand, since UDP has no state information, we rely on the server-side analytic engine to further identify UDP open ports. Hence, the collected UDP port records are only the initial results and not all of them will be treated as open ports (e.g., the client UDP port used by YouTube in Fig. 2(b)).

Challenges. The goal of long-term port monitoring on real user devices requires NetMon to *periodically* analyze those four `proc` files with minimal overhead. A simple idea of creating a "long-lived" service to periodically monitor open ports would not work as the service will be stopped by Android after a certain amount of time (e.g., after the device goes to sleep) or simply terminated by users. To overcome this, we leverage Android AlarmManager [2] to schedule periodic alarms to perform the `proc` file analysis robustly. We chose five minutes as the alarm interval because it provides a good sampling rate (excluding many client UDP ports) while incurring negligible overhead. Our experience shows that the potential information loss within the five-minute interval is well compensated by the large number of users contributing data in our crowdsourcing campaign. Moreover, we take advantage of the batched alarm mechanism [3] introduced since Android 4.4 and a characteristic in `/proc/net/tcp6|tcp` — the server socket entries always appear in the top rows — to further minimize the overhead. As a result, NetMon incurs less than 1% overhead on CPU and battery for a daily usage.
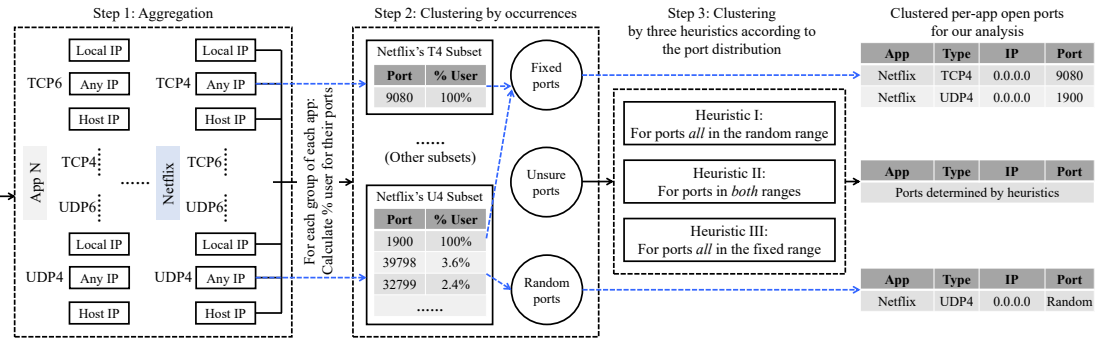
Fig. 3: An overview of our server-side open-port analytic engine to perform the three-step clustering (using Netflix as an example).

B. Server-side Open-Port Analytic Engine

The open port information gathered from individual phones, e.g., the Netflix app opens TCP port 9080 at time t_1 and opens UDP port 39798 at time t_2, constitute individual *observations* that need to be clustered to generate per-app open port results, e.g., Netflix has a fixed TCP port 9080 and a random UDP port. More specifically, different port records associated with the same "random" open port should be unified, and open ports with "fixed" port numbers should be recognized. This may sound straightforward, but it turns out to be a challenging task because fixed and random ports could exhibit indistinguishable observations. To overcome this challenge, we introduce a server-side analytic engine, as shown in Fig. 3, to perform a three-step clustering:

Step 1: Aggregation. We first aggregate each app's observations by different types of ports and IP addresses. This is a "narrow down" step to effectively reduce the complexity of clustering — open ports with different types or IP addresses shall be in different clusters, since they are created by different APIs or `InetAddress` parameters at the code level. Specifically, we divide the observations into 12 groups, enumerating the combination of four types of ports (TCP/UDP ports in IPv4 or IPv6) and three types of IP addresses (loopback address `127.0.0.1`, ANY address `0.0.0.0`, and the specific host address such as `192.168.X.X`). In the Netflix example shown in Fig. 3, we have two groups — TCP4 and UDP4 (both with IP `0.0.0.0`).

Step 2: Clustering by occurrences. A fixed port on an app presents itself as identical records on multiple user devices, while a random port presents its observations with different port numbers. Based on this observation, we can differentiate between fixed and random ports by analyzing the occurrences of a record within each group (constructed in Step 1). We define this occurrence as the fraction of user devices presenting a specific port number within the group. For example, the UDP port 39798 for IPv4 address in our Netflix set has an occurrence of 3.6%.

With this definition of the occurrence, we perform port clustering where fixed ports are those with a high occurrence and random ports are those with low ones. As shown in Fig. 3, Netflix's UDP port 39798 in our dataset is certainly a random port because its occurrence is only 3.6% among the 84 Netflix users in the UDP4 group, whereas TCP port 9080 is a fixed port because its occurrence has reached 100% in the TCP4 group. In practice, we use 50% as the upper

bound for the low-occurrence scenario, which is based on the assumption that fixed ports should cover at least more than half of the users in the group. We consider those with occurrences higher than 80% as fixed ports. However, the threshold-based occurrence strategy tends to be unreliable when group sizes are small because a random port exhibiting a number of different observations may have one or several of them show up with high occurrences. In these cases (and others with occurrences between 50% and 80%), we apply a heuristics approach, to be described next, to get a more accurate inference.

Step 3: Clustering by heuristics. For observations that cannot be reliably determined by occurrences, we further leverage three heuristics to handle them. We first separate port numbers into the "random" range (for port numbers between 32,768 and 61,000, i.e., those randomly assigned by the OS or the so-called ephemeral ports [16]) and the "fixed" range (for other port numbers). For each group, we count the numbers of unique port numbers within these two ranges, and denote them by N_r and N_f, respectively. We then have the following three port distribution patterns and their corresponding heuristics:

- *All ports are in the random range* ($N_r > 0$ and $N_f = 0$). We simply mark them as one random port based on the *conservative* principle that we can tolerate misclassifying a fixed port to be a random one but not the opposite.
- *Ports are in both ranges* ($N_r > 0$ and $N_f > 0$). We first consider all ports in the random range as presenting one random port. If N_r is significantly bigger than N_f (e.g., ten times) and N_f is relatively small (e.g., less than 3), we mark ports in the fixed range as fixed ports.
- *All ports are in the fixed range* ($N_r = 0$ and $N_f > 0$). We conservatively output just one random port if N_f is not small (e.g., larger than 3); otherwise, we consider them as fixed ports.

C. Crowdsourcing Results

We have deployed NetMon to Google Play for an IRB-approved[2] crowdsourcing study since 18 October 2016. In this paper, we base our analysis on the data collected till the end of July 2017 (a period of around ten months when most of our evaluations were performed and security findings were confirmed), which involves 3,293 user phones from 136 different

---

[2]IRB approval was obtained from Singapore Management University on 14 October 2016. Under this study, we do not collect personally identifiable information (PII) or IMEI. We use only the anonymized ANDROID_ID (hashed with a salt) for device identification. Users are also explicitly informed about all the information we collect through a pop-up confirmation dialog.
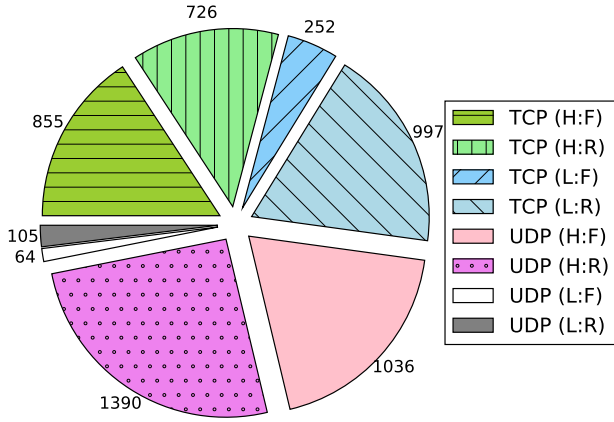
Fig. 4: Apps with open ports in different types of socket addresses (symbols are "H"/"L": host/local IP; "F"/"R": fixed/random port number), including 1,390 apps with long-lasting client UDP ports.

countries worldwide. Users of NetMon are attracted solely via Google Play without advertisements or other incentives. About a quarter of the devices (26%) are from the US, while the percentage for other countries is very diverse, which makes our dataset more representative.

In our dataset, we collect 40,129,929 port monitoring records and discover 2,778 open-port apps (2,284 apps with TCP open ports and 1,092 apps with UDP ones) and a total of 4,954 open ports (3,327 TCP ports and 1,627 UDP ports). Note that with the help of our analytic engine, we can classify UDP random ports bound to the host IP address as client UDP ports. Fig. 4 shows the distribution of open-port apps with different types of socket addresses. We find that both TCP and UDP open ports have their fair share in these apps, and many of these ports expose them to potential network attacks (e.g., bound to non-local IP addresses). In addition, we find that 1,390 apps use long-lasting (more than 5 minutes) client UDP ports to communicate with servers. To the best of our knowledge, this work constitutes the first report of crowdsourcing Android apps with open ports and their IP address and port number information.

1) Open Ports in Popular Apps: With the help of Selenium [31], a web browser automation tool, we obtain the number of installs of the 1,769 open-port apps on Google Play, and find that 925 apps (52.3%) have over one million installs. Among them, 100 apps even have over 100M installs each. We thus take a closer look at these 100 highly popular apps and present 28 representatives of them in Table I. We can see that popular apps such as Facebook, Instagram, Skype, WeChat, YouTube, Spotify, Netflix, and Plants vs. Zombies are surprisingly *not* free of open ports.

An interesting observation is that 89 out of the 925 popular apps (9.6%), including Firefox and Google Play Music as listed in Table I, use UDP port 1900 and/or 5353 for the UPnP and mDNS services, respectively. Furthermore, the open-port timeline analysis shows that both ports cumulatively last for over a month for each of their top ten apps, which provides enough time window for adversaries to launch attacks. In particular, Bai et al. [83] has demonstrated that such ports in iOS and OSX apps could suffer from Man-in-the-Middle attacks.

TABLE I: Representative apps that have open ports.

Category	App Name	Type	IP†	Port	# of Installs
Social	Facebook	TCP	L	Random	1B - 5B
	Instagram	TCP	L	Random	1B - 5B
	Google+	TCP	H	Random	1B - 5B
		TCP	L	Random	
	VK	TCP	H	48329	100M - 500M
		TCP	L	Random	
Communication	Messenger	TCP	L	Random	1B - 5B
	WeChat	TCP	H	9014	100M - 500M
	Skype	TCP	H	Random	500M - 1B
		TCP	L	Random	
	Chrome	TCP	L	5555	1B - 5B
	Firefox	TCP	H	8080	100M - 500M
		TCP	L	Random	
		UDP	H	1900	
Video Players or Music & Audio	YouTube	TCP	H	Random	1B - 5B
		TCP	L	Random	
	GPlay Music	TCP	L	Random	1B - 5B
		UDP	H	1900	
	Spotify	TCP	H	Random	100M - 500M
	Amazon Music	TCP	L	Random	100M - 500M
		TCP	H	Random	
Tools	Google Play Services	UDP	H	2346	5B - 10B
		UDP	H	5353	
	Google	TCP	H	20817	1B - 5B
	Clean Master	TCP	L	Random	500M - 1B
	360 Security	TCP	L	Random	100M - 500M
	Avast	TCP	H	20817	100M - 500M
		TCP	L	Random	
Productivity	Google Drive	TCP	L	Random	1B - 5B
	Cloud Print	UDP	H	5353	500M - 1B
	ES File Explorer	TCP	H	42135	100M - 500M
		TCP	H	59777	
		TCP	L	Random	
		UDP	H	5353	
Entertainment	GPlay Games	TCP	L	Random	1B - 5B
	Netflix	TCP	H	9080	100M - 500M
		UDP	H	1900	
		UDP	L	Random	
	Peer Smart Remote	TCP	L	Random	100M - 500M
		UDP	H	5353	
Games	Plants vs. Zombies 2	UDP	H	24024	100M - 500M
	Asphalt 8	TCP	H	7940	100M - 500M
	Solitaire	TCP	L	Random	100M - 500M
	Sonic Dash	TCP	L	Random	100M - 500M

† "L" is for the local IP address and "H" is for the host IP, as termed in Sec. II.

Compared to UDP, TCP open ports have more diverse usages. The top five open TCP port numbers, port 8080, 30102, 1082, 8888, and 29009, have no well-defined fixed usage (unlike the UDP port 1900 and 5353 above) and appear in only 14 to 64 apps. Despite this diversity, it is interesting to see some uncommon TCP port numbers (e.g., 30102 and 29009) appearing in multiple apps. To gain a better understanding of these open ports, we perform static analysis and find that many of them are introduced by SDKs (see Sec. IV-D for more details). As the most interesting example, Facebook SDK is the major contributor to 997 apps (of the entire dataset) for their random TCP ports bound to the local IP address (i.e., the fourth sector in Fig. 4). Such local random TCP ports appear in 62.8% of the 925 popular apps, and the percentage goes up to 78% in the 100 highly popular apps. As shown in Table I, even anti-virus apps, 360 Security, and Avast, are also affected.

2) Open Ports in Built-in Apps: Besides the popular apps on Google Play, we also identify 755 built-in apps (apps pre-installed by phone manufacturers) containing open ports (excluding those that also appear as standalone apps on Google Play, such as Facebook and Skype). We recognize them by collecting user devices' *system* app package names (via the SYSTEM flags of the ApplicationInfo class).

TABLE II: Top smartphone vendors that include open-port apps.

Vendor	# Apps		Top Five Open Port Numbers				
Samsung	186	UDP:	5060	68	1900	6100	6000
		TCP:	5060	6100	6000	7080	8230
LG	75	UDP:	68	1900	19529	5060	39003
		TCP:	5060	59150	59152	8382	39003
Sony	69	UDP:	68	1024	1900	1901	-
		TCP:	5000	5900	5001	9000	30020
Qualcomm	42	UDP:	68	5060	1900	32012	-
		TCP:	5060	6100	4000	4500	4600
MediaTek	26	UDP:	68	5060	50001	50002	50003
		TCP:	5060	50001	-	-	-
Lenovo	25	UDP:	68	5060	50000	50001	52999
		TCP:	2999	5060	50001	55283	39003
Motorola	21	UDP:	68	32012	16800	-	-
		TCP:	2631	20817	-	-	-
Huawei	13	UDP:	68	1900	8108	-	-
		TCP:	-	-	-	-	-
ASUS	13	UDP:	68	5353	11572	11574	-
		TCP:	2222	5577	8258	8282	8990
Xiaomi	11	UDP:	68	1900	5353	-	-
		TCP:	6000	8081	8682	-	-

With vendor-specific package keywords, we identify over 20 vendors that include open ports in their built-in apps. Table II lists the top ten according to the number of built-in apps with open ports. We can see that Samsung, LG, and Sony are the top three vendors, with 186, 75, and 69 open-port apps, respectively. Considering the huge numbers of phones sold by these vendors, their built-in open ports are expected to exist in a significant portion of the entire smartphone market. By analyzing each vendor's top five open ports, we identify three major reasons for including these open ports in these built-in apps.

First, more than half (489 apps, 64.8%) of these apps[3] contain UDP open port 68, which is for receiving DHCP broadcasts and updating the host IP address. As shown in Table II, UDP port 68 appears in all top ten device vendors, and it often affects the largest number of built-in apps in each vendor. Furthermore, we find that opening UDP port 68 is often long-lasting, with the median value of cumulative port-opening time being 32.3 hours per app. This port can leak the host name of the phone, which was fixed only in the latest Android 8 [11].

Second, about one quarter (175 apps, 23.2%) have TCP/UDP port 5060 open, which is for VoIP SIP connection setup [35]. These built-in apps are from five device vendors: Samsung, LG, Lenovo, Qualcomm, and MediaTek. By inspecting these apps, we find that quite a number of them do not seem to require the SIP capability, e.g., `com.lenovo.powersetting`, `com.sec.knox.bridge`, `com.sec.automation`, and `com.qualcomm.location`, to name a few.

Moreover, we surprisingly find that 41 Samsung models and 16 LG models modify some Android AOSP apps (e.g., `com.android.settings` and `com.android.keychain`) to introduce the open port 5060. Other cases where Android AOSP apps are customized to introduce open ports include TCP port 6000 in Xiaomi's `com.android.browser` app, and UDP port 19529 opened by LG's 18 system apps. Most of these apps, e.g., `com.lge.shutdownmonitor` and `com.lge.keepscreenon`, generally have no networking

---

[3]Note that 175 of them also contain other ports.

functionality. This suggests that their open ports could be unnecessary. We leave an in-depth analysis of these cases to our future work.

Third, the rest of the open ports are mainly for network discovery and data sharing. Besides common port numbers such as 1900 (UPnP) and 5353 (mDNS), vendors use custom ports to implement their own discovery and data sharing services. Examples include TCP ports 7080 and 8230 for Samsung's Accessory Service [30], TCP port 59150 and 59152 for LG's Smart Share [22], and TCP port 5000 and UDP port 1024 for Sony's DLNA technique [33]. We reverse engineer Samsung Accessory and identify a security bug; see Sec. V-A.

3) Pervasiveness and Effectiveness: The crowdsourcing results presented above have demonstrated the pervasiveness of open ports in Android apps and the efficacy of using crowdsourcing to discover open ports. For example, the number of apps found with TCP open ports (2,284 apps) is significantly more than that found in the state-of-the-art research [52] (1,632 apps), which is based on a large set of 24,000 apps. To further quantify those two metrics, we correlate the crowdsourcing results with two sets of apps used in static analysis.

To quantify the open-port pervasiveness, we crawled a set of top 9,900 free apps from Google Play in February 2017 (fitting the period of our crowdsourcing). These apps are comprised of the top 300 free apps from 33 Google Play categories, with all gaming apps consolidated into a single category. By looking into the overlapping of this set and the apps monitored by NetMon, we count a total of 3,216 apps (with vendor built-in apps excluded). Out of these 3,216 apps, our results show that 492 of them present TCP open ports, i.e., 15.3% of pervasiveness, which is significantly higher than a previous report (6.8%) based on static analysis [52].

To quantify the effectiveness of our crowdsourcing approach, we first prepare a baseline set of apps. Out of the 2,284 TCP open-port apps (some are built-in apps) discovered by crowdsourcing, we are able to obtain 1,027 apps from the public AndroZoo app repository [39]. According to the experimental results in Sec. IV-C, only 58.9% of these apps can be detected by typical Android static analysis. In particular, 25.1% of them use dynamic code loading [65] or advanced code obfuscation [78]. They are therefore not possibly detected by a pure static analysis [46], [67]. This indicates that crowdsourcing is much more effective than Android static analysis in the context of open port discovery.

IV. DIAGNOSIS VIA STATIC ANALYSIS

While crowdsourcing is effective in discovering open ports, it does not reveal the code-level information for more in-depth understanding and diagnosis. To understand how open ports are actually constructed at the code level and its security implication, our pipeline (Fig. 1) includes a diagnosis phase through OPTool, a static analysis tool we develop specifically for the open-port diagnosis. Note that the goal of our diagnosis is *not* to rediscover (and analyze) *all* open ports identified by our crowdsourcing as we have shown that crowdsourcing is more effective for port discovery. Instead, we aim to understand the *major* open-port usages by enhancing typical Android static analysis with open-port context and semantics. As a result, we limit our static analysis to TCP open ports

as similar to OPAnalyzer [52], since UDP open ports have much more fixed usages (mainly for providing system-level networking services) as we have seen in Sec. III-C. In addition, overcoming the common difficulties in existing Android static analysis (e.g., dealing with dynamic or reflected codes) is also not our focus.

In this section, we first cover the background of code-level open port construction and the objectives of our analysis (Sec. IV-A), and then present the details of our static analysis tool OPTool (Sec. IV-B). Finally, we present the experiments we have performed (Sec. IV-C) and the diagnosis results (Sec. IV-D and Sec. IV-E).

A. Open Port Construction and Our Analysis Objectives

At the code level, an open port on Android could be constructed in either Java or C/C++ native code. The native construction is similar to the traditional server-side programming by calling `socket()`, `bind()`, `listen()`, and `accept()` system calls sequentially, while the Java construction is to simply initialize a `ServerSocket` object and call the `accept()` API. The *first objective* of our static analysis is to trace each construction to (i) differentiate if the construction constitutes a "live port" or a "dead port," and (ii) determine if a third-party SDK is on the call hierarchy. Such understanding is important because we want to filter out false positives of open-port constructions, and Android apps usually include various SDKs [41], especially the advertisement or analytics SDKs [50], [73], which could introduce open ports without developers' awareness. This analysis is challenging because many networking libraries included in the app may contain open-port code that is never invoked by the host app. We therefore need a backward slicing analysis that can accurately trace back to every node on the call hierarchy. Such analysis has to be sensitive to the calling contexts, class hierarchy, implicit flows, and so on.

After digging deeper into the Java constructions, we find a total of 11 open-port constructor APIs shown in Listing 1. These `ServerSocket` APIs were originally from Java SDK, and have been directly ported over to Android. A convenient way of invoking these APIs is to pass only the port number parameter, and the APIs will automatically assign the `addr` and `backlog` parameters. The default setting of `addr`, interestingly, is the ANY IP address instead of the local loopback IP address. Moreover, if `addr` is set to `null`, the ANY IP address is also used by default. This legacy design in the original Java SDK might be appropriate for open ports on PCs but not for mobile — as we saw earlier in Table I, many Android open ports are designed for local usages. We consider this kind of "convenient" usage potentially *insecure* in the sense that they could inadvertently increase the attack surface.

In view of such potentially insecure use of the APIs, we come up our *second objective* of identifying the precise parameter values of all open-port constructions, so that we can evaluate the extent to which Android developers adopt such convenient but potentially insecure Java APIs. Note that these parameters might evolve across different objects, fields, arrays, and involve arithmetic operators and Android APIs. We need to understand all these semantics and calculate a complete representation of the parameters (instead of just capturing

```
// API #1-#3
ServerSocket(int port);
ServerSocket(int port, int backlog);
ServerSocket(int port, int backlog, InetAddress addr);

// API #4-#6
SSLServerSocket(int port);
SSLServerSocket(int port, int backlog);
SSLServerSocket(int port, int backlog, InetAddress addr);

// API #7-#9
//class ServerSocketFactory:
createServerSocket(int port);
createServerSocket(int port, int backlog);
createServerSocket(int port, int backlog,InetAddress addr);

// API #10-#11
//ServerSocket socket = new ServerSocket();
socket.bind(SocketAddress addr);
socket.bind(SocketAddress addr, int backlog);
```

Listing 1: All `ServerSocket` constructor APIs.

isolated constants in SAAF [51]). Last but not the least, it is important for our analysis to be efficient and scalable with a large number of Android apps.

B. OPTool's Design and Implementation

We design and implement a new static analysis tool called OPTool to specifically handle these challenges. Instead of generating traditional slicing paths, OPTool uses a structure called *backward slicing graph* (BSG) to simultaneously track multiple parameters (e.g., `port` and `addr`) and capture a complete representation of the parameters. On the generated BSGs, OPTool performs graph traversal and conducts *semantic-aware constant propagation*. We also include a preprocessing step in OPTool to quickly search for open-port constructions to improve its scalability.

Locating open-port constructions. This can be done by searching for the `accept()` API of `ServerSocket` and `ServerSocketChannel` classes, which are the only Android APIs to open TCP ports in Java. To enable fast searching and to handle the *multidex* issue (where Android apps split their bytecodes into multiple DEX files to overcome the limit of having a maximum of 65,536 methods [12]), we use `dexdump` [15] to dump (multiple) app bytecodes into a (combined) plaintext file and then perform the searching. Additionally, for the native code, OPTool searches each `.so` file for the four socket system calls.

Backward parameter slicing via BSG. After locating the open-port constructions, we apply backward slicing on their parameters to generate BSGs. Each BSG corresponds to one target open-port call site and records the slicing information of all its parameters and paths. The BSG not only enables OPTool to track multiple parameters in just one backward run, but also makes our analysis flow- and context-sensitive, e.g., the process of constructing BSG naturally records the calling context when analyzing the target of a function call so that it can always jump back to the original call site. OPTool is also sensitive to arrays and fields. With the help of forward constant propagation shown below, our backtracking just needs to taint both the instance field (or the array index) and its class object. Handling static fields does not need the extra help, but requires us to add their statically uninvoked `<clinit>` methods (where static fields get initialized) into the BSG.

7

A notable challenge for Android backward slicing is to deal with implicit flows and callbacks. OPTool builds in support of class hierarchy, interface methods, asynchronous execution (e.g., in `Thread`, `AsyncTask`, and `Handler`), and major callbacks in the EdgeMiner list [44]. Furthermore, we support backtracking across (explicit) inter-component communication (ICC) [62], and model Android component lifecycle [40].

Semantic-aware constant propagation. After performing the inter-procedural backward slicing, we calculate the complete parameter representation in a forward manner. Besides the instruction semantics as in the typical forward propagation [48], we handle the following semantics:

Maintaining object semantics. To determine the correct object for each instance field, we perform *points-to* analysis [54] for all `new` statements in the BSG. Specifically, we define an `InstanceObj` structure and initialize a unique `InstanceObj` object for each `new` statement. We then propagate the `InstanceObj` objects along the path and update their member fields if necessary. As a result, whenever a target instance field is to be resolved, we can retrieve its corresponding `InstanceObj` and extract its value. Array and ICC objects can be treated similarly with our modeling of the `Intent` APIs for updating/retrieving the ICC object fields.

Modeling arithmetic and API semantics. We model not only the five major arithmetic operators, `+`, `-`, `*`, `/`, and `%` (by extracting the two operands and generating a corresponding statement in Java code), but also mathematical APIs, e.g., `Math.abs(int)` and `Math.random()` (via a special constant "RANDOM"). We also model all other encountered Android framework APIs, which include IP address APIs, `Integer` and `String` APIs, and `SharedPreferences` APIs. There are also a few APIs that are statically unresolvable, e.g., retrieving values from user interface via `EditText.getText()` and from database via `Cursor.getInt(int)`. We save these cases to the final results without resolving their parameters.

Removing dead ports and resolving SDK names. An important feature in OPTool is the removal of "dead ports" that are never executed. We analyze the port liveness in three steps of OPTool. First, during the backward slicing, we perform reachability analysis to exclude slices that cannot trace back to the app entry functions. Second, in the forward propagation, we consider ports with unresolvable parameters as dead ports. Third, the post-processing step excludes dead ports with illegal parameters, e.g., we have detected tens of cases with port parameter -1.

Resolving names of the SDKs is non-trivial due to code obfuscation. That said, we have had successes with (i) extracting the name of each "sink" class that directly calls `ServerSocket` constructor APIs — typically the non-obfuscated portions, e.g., `com.facebook.ads.internal.e.b.f` for the Facebook Advertisement SDK; (ii) extracting Android Logcat tags [6] of the sink classes which may embed plaintext class names, as demonstrated in Google's official document [6]; and (iii) correlating different apps' open-port parameters and tags, e.g., most Alibaba AMap SDK [4] classes are obfuscated, but we can still find non-obfuscated instances, e.g., `com.amap.api.location.core.SocketService`.
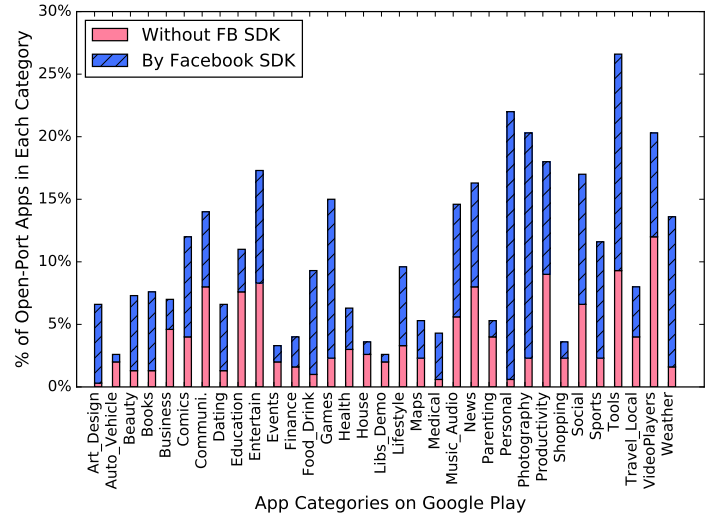

Fig. 5: Percentage of open-port apps in each Google Play category.

C. Static Analysis Experiments

As explained in Sec. III-C3, we have two sets of apps for analysis: (i) the top 9,900 apps across 33 Google Play categories and (ii) the 1,027 apps from AndroZoo that are confirmed with TCP open ports.

We use the first set to measure the distribution of open-port apps across different categories. Out of the 9,900 apps statically analyzed by OPTool, we identify 1,061 apps and their corresponding 1,453 TCP open ports. Fig. 5 plots a bar chart of the percentage of open-port apps in each Google Play category. It clearly shows that open port functionality has been planted into apps in all 33 Google Play categories, ranging from the lowest 2.67% in "Libraries & Demo" to the highest 26.67% in "Tools". After excluding Facebook SDKs, the percentage drops to between 0.33% in "Art & Design" and 12.0% in "Video Players & Editors". This suggests that open ports may have a wider adoption in mobile systems than that in the traditional PC environment.

We then use the second set of apps to quantify the effectiveness of crowdsourcing in a comparison with static analysis, as mentioned in Sec. III-C3. Out of the 1,027 open-port apps as ground truth, OPTool flags 671 apps with potential Java open-port constructions and 98 apps with native open-port constructions. Among the remaining 258 (25.1%) apps, 110 of them implement open ports via dynamic code loading[4], and the rest of 148 apps are likely equipped with advanced code obfuscation (e.g., multiple anti-virus apps, such as Avast shown in Table I, appear in this set). For the 671 apps analyzed by OPTool for open-port parameters, it successfully recovers the parameters of 459 apps and identifies 48 statically unresolvable cases (e.g., values from `EditText`). Other cases are mainly due to the complex implicit flows (e.g., [43], [55]) that OPTool currently cannot address, even we have adopted the state-of-the-art methods [40], [42], [44]. We argue that in an "ideal" situation (where all 98 apps with native constructions are successfully analyzed and 48 statically unresolvable cases are included), a typical static analysis tool can detect only 58.9% of open-port apps that are discovered by our crowdsourcing approach.

---

[4]We measure it via `DexClassLoader` and `PathClassLoader` APIs.

TABLE III: Open-port SDKs detected in our dataset, and the number of apps affected by them.

SDK	Pattern	#
Facebook Audience Network SDK [17]	`Class='com.facebook.ads.%', Tag=ProxyCache, Ip=127.0.0.1, Port=0, Backlog=8`	897
Yandex Metrica SDK [38]	`Class='com.yandex.metrica.%', Port=29009|30102`	28
CyberGarage UPnP SDK [14]	`Class=org.cybergarage.http.HTTPServer, Ip=getHostAddress(), Port=8058|8059`	19
MIT App Inventor SDK [25]	`Class=com.google.appinventor.components.runtime.util.NanoHTTPD, Port=8001`	19
Tencent XG Push SDK [36]	`Class=com.tencent.android.tpush.service.XGWatchdog, Port=RANDOM+55000`	13
Corona Game Engine SDK [13]	`Class=com.ansca.corona.CoronaVideoView, Port=0, Backlog=8`	11
Alibaba AMap SDK [4]	`Class='com.amap.%', Port=43689`	9
Millennial Ad SDK [24]	`Class='com.millennialmedia.android.%', Tag=MillennialMediaAdSDK, Ip=null, Port=0`	8
PhoneGap SDK [28]	`Class=com.phonegap.CallbackServer, Port=0`	6
Titanium SDK [37]	`Class=org.appcelerator.kroll.common.TiFastDev, Tag=TiFastDev, Port=7999`	6
Aol AdTech SDK [8]	`Class=com.adtech.mobilesdk.publisher.cache.NanoHTTPD, Port=RANDOM+9000`	6
Apache Cordova SDK [9]	`Class=org.apache.cordova.CallbackServer, Port=0`	4
Getui Push SDK [18]	`Class='com.igexin.push.%', Port=48432|51688, Ip=0.0.0.0`	3

Considering both sets of apps and focusing on those with their parameters successfully recovered by OPTool, we further analyze the 1,520 (1,061 + 459) apps with open ports in the next two subsections.

D. Detection of Open-Port SDKs

Out of these 1,520 apps, we are able to detect 13 open-port SDKs that affect at least three apps each in our dataset. Table III lists their details, including the class pattern (we use "%" to represent obfuscated fields), the Android Logcat tag (if any), raw open-port parameters, and the number of affected apps. Note that the app number here is the number of apps that actually invoke the SDK code, because some apps may embed an open-port SDK but never invoke it. For example, we found a total of 1,110 apps embedding Facebook Audience Network SDK [17] but only 897 of them triggering the SDK code.

These SDKs are invoked in 1,018 apps (a few apps embed multiple SDKs), and only 581 open-port apps are not affected at all. In other words, 61.8% of the 1,520 open-port apps are solely due to SDKs, among which Facebook SDK is the major contributor. Even after excluding the impact of Facebook SDK, we could still count 117 (16.8%) open-port apps that are solely due to SDKs. These results indicate that SDK-introduced open ports are significant and should be considered seriously in terms of their necessity as we will discuss in Sec. VI.

We take a closer look at Table III to see what kinds of SDKs introduce open ports and whether it could raise an alarm to developers. We find that only three SDKs, the UPnP SDK from CyberGarage [14] and two mobile push SDKs [18], [36], are networking related. The others are about advertisements [8], [17], [24], [38] (e.g., Facebook and Yandex), Javascript generation [9], [25], [28], [37] (e.g., App Inventor and PhoneGap), gaming engines [25] and map navigation [4]. Hence, we argue that developers could hardly realize the existence of these open ports by simply examining their functionality.

E. Identification of Insecure API Usages

We further analyze the 581 apps whose open ports are not introduced by SDKs, and their corresponding 869 open ports. We find that 515 port constructions did not set the IP `addr` parameter and 96 ports set it as "null". Hence, the default setting of `addr`, i.e., the ANY IP address, is automatically used for these ports. In total, these convenient API usages account to 611 open ports from 390 apps (67.1%). Furthermore, 164 of these ports (coming from 120 apps) have their `port` parameter set as random, which has nearly no chance of being able to accept external connections and thus binding to the ANY IP address clearly increases their attack surfaces. This translates to a (lower bound) estimation of 26.8% of the 611 convenient API usages being insecure, and correspondingly 20.7% (120/581) open-port apps adopting convenient but insecure API usages.

Such an insecure coding practice is not limited to app developers but also SDK producers. In Table III, six SDKs make a random port yet using the default `addr` parameter binding the port to ANY IP addresses. Hence, Google may reconsider the design of `ServerSocket` APIs to enhance its security at the API level.

V. SECURITY ASSESSMENT

In the last phase of our pipeline (Fig. 1), we perform comprehensive security assessment of open ports in three directions: vulnerability analysis in Sec. V-A, denial-of-service attack evaluation in Sec. V-B, and inter-device connectivity measurement in Sec. V-C.

A. Vulnerability Analysis of Open Ports

According to our experience of analyzing open-port vulnerabilities over more than two years, it is easy for open-port apps to become vulnerable, especially for TCP open ports that do not provide system networking services as UDP open ports (as explained in Sec. III-C1). Therefore, instead of developing tools to detect *individual* vulnerable open ports, we attempt to uncover vulnerability *patterns* in popular apps that are usually more representative and more difficult to detect. Hence, our vulnerability analysis is quite different from the previous work [52] that uses pre-defined pattern for vulnerability detection. Instead, we explore all possible ways in which an open port could become vulnerable, as long as they fit our threat model discussed in Sec. II, by performing in-depth reverse engineering via the state-of-the-art JEB Android decompiler [21] and extensive dynamic testing.

Table IV summarizes the five vulnerability patterns we have identified. The first two have been reported in [52], while the third is a new variant of the crash vulnerability mentioned in the traditional Android app security research [49]. The last two have not been reported and they are specific to open ports.

TABLE IV: Vulnerability patterns identified in open ports.

ID	Vulnerability Patterns	Representative Apps Affected
P1	No/insufficient checks for information transmission	Samsung Gear, Cloud Mail.Ru, Vaulty, ES File Explorer
P2	No/insufficient checks for command execution	Tencent XG Push SDK, Baidu Root, Coolpad V1-C Phone
P3	Crash-of-Service (CoS)	Skype, Instagram
P4	Stealthy Data Inflation	Facebook SDK, Instagram
P5	Insecure Analytics Interface	Sina Weibo, Alibaba & Baidu SDKs

P1: No or insufficient checks for information transmission. One major usage of (TCP) open ports is to transmit data to the connecting parties. However, apps may employ weak authentication or even no authentication, which allows unauthorized access to sensitive contents. We identify this type of vulnerabilities in ES File Explorer, Cloud Mail.Ru, and a popular photo/video hiding app called Vaulty. For example, Cloud Mail.Ru's TCP port 1234 leaks users' videos at `http://127.0.0.1:1234//filename`, where the name can be leaked by eavesdropping Cloud Mail.Ru's broadcast messages [45]. Similarly, Vaulty leaks users' sensitive videos and pictures to a remote adversary through port 1562, and the adversary does not even need to know the target filename because only an integer starting from one is required. ES File Explorer's always-on TCP port 59777 performs some security checks by validating the IP addresses of incoming requests with a white list. However, there is also an implicitly exposed [45] `Activity` component for adding a remote adversary's IP address to the white list.

A particularly interesting example is Samsung Gear and other built-in apps based on the Accessory service [30] mentioned in Sec. III-C. Samsung Accessory provides an *automatic (service) discovery* feature via TCP port 8230, but replies with sensitive information, e.g., `GT-I9305;samsung;UserName(GT-I9305);SWatch;SAP_TokenId(omitted)`, to any connecting party. Generally, it is important, yet challenging, to return only appropriate information in such UPnP-like apps (e.g., 19 apps using CyberGarage UPnP SDK; see Table III).

P2: No or insufficient checks for command execution. Another usage of open ports is to execute commands sent by authorized clients. We can see such open-port usage in Tencent XG Push SDK for executing push commands and the Coolpad V1-C phone's vpowerd system daemon for `shutdown` and `reboot` commands. However, the command interfaces in both cases are not well protected.

We also notice that some open ports are used as a debugging interface. For example, MIT App Inventor [25] and Titanium SDK [37] in Table III use open ports for instant debugging or the so-called living programming [70]. This debugging interface, however, must be disabled in release versions; otherwise, sensitive debugging information could be leaked. For example, Baidu Root, a popular rooting app in China, suffers from this vulnerability in its TCP port 10010 (bound to the host IP address).

P3: Crash-of-Service. Apps could crash when receiving malformed inputs from their open ports — we call this *Crash-of-Service* (CoS). Traditionally, Android apps suffer from CoS due to inter-component communications [49]. Now open ports provide a new channel for launching CoS. For example, we
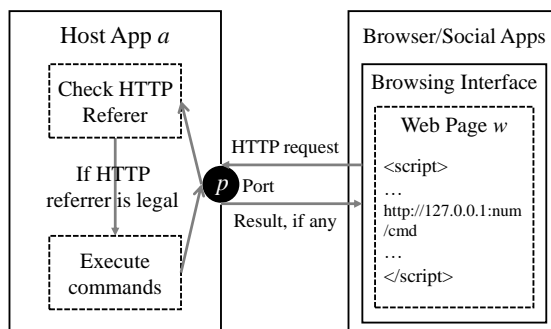

Fig. 6: The model of using open ports for analytics.

can crash Instagram by sending it an invalid HTTP URL via the open ports. We also find that SIP VoIP apps (e.g., built-in apps using the standard VoIP port 5060 as discussed in Sec. III-C) could be victims of CoS attacks. Here we analyze Skype voice/video calls' VoIP-like mechanism — it uses one UDP port for receiving control messages from a Microsoft Azure server, and another UDP port for exchanging media data with the other Skype user in a P2P mode. Unfortunately, a remote adversary can terminate the on-going Skype session by just sending two packets to the first UDP port. This leads to a very effective CoS attack without even involving application-layer packets.

P4: Stealthy data inflation. Many open ports are for caching purposes (or as connection proxies in VPN apps). For example, Facebook SDK uses its open ports to cache video-based advertisements. Individual apps, such as Instagram, can also build their own cache servers upon an open-source library called AndroidVideoCache [7]. Since these apps typically support opening arbitrary URLs via the open ports, one can easily launch *stealthy* data inflation attacks. Specifically, an adversary can send special URLs, e.g., an URL pointing to a large file, to maliciously inflate victim apps' cellular data usage in the background. This process is fully stealthy without catching user attention, and the data usage is attributed to the victim app.

Our vulnerability reports on Facebook SDK and Instagram were confirmed by Facebook in March 2017 with two bug bounty awards, which demonstrate the effectiveness of the stealthy data inflation attack. Generally, it is applicable to any open port with the caching or proxy functionality, e.g., most of the 997 apps with a local random TCP port (see Sec. III-C1) and Corona Game Engine SDK (in Table III). The only exception we have seen is the open port on YouTube, which uses a checksum to restrict opening illegal URLs.

P5: Insecure analytics interface. Lastly, we present a special vulnerability pattern that appears in open port used as an analytics interface, which is used by host apps/SDKs' campaign websites to retrieve analytics information. Fig. 6 depicts its basic architecture, in which a victim user has installed an app a that hosts an analytic open port p (with a fixed port number num). Whenever a user visits a web page w (that has a campaign relationship with a) from her mobile browser or from user-shared links in social apps, w sends an HTTP request to `http://127.0.0.1:num/cmd` with the by-default added HTTP referrer pointing to the URL of w. The analytics app receives the request over its open port

(a) WeChat's voice call (DoS at ∼26s).

(b) YouTube's video streaming (DoS at ∼32s).
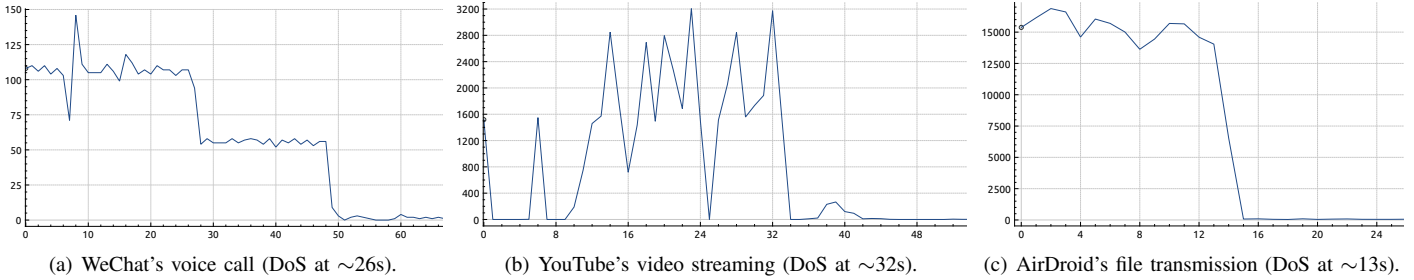
(c) AirDroid's file transmission (DoS at ∼13s).

Fig. 7: DoS attacks against open ports. The x-axis is the time in seconds, and the y-axis is the victim apps' throughput (packets/sec).

and checks whether the request is from a campaign website through the HTTP referrer. If it is, the app executes one of its pre-defined commands as requested by the `cmd` parameter. A common command is `geolocation`, upon executing which the geographical location of the device is returned to the web page.

However, such open-port usage is fundamentally insecure, because any other local apps or even a remote adversary (if the open port bound to the host IP address, which is often the case) can set an arbitrary referrer in their HTTP requests to execute privileged commands (e.g., retrieving IMEI and list of installed apps). We uncover this class of vulnerabilities in Sina Weibo, Alibaba AMap SDK, and two Baidu SDKs (which were fixed quite long ago and thus not in Table III). We reported these issues to the vendors in the first half of 2015, much earlier than the subsequent industrial reports (e.g., WormHole [32])[5].

B. Denial-of-Service Attack Evaluation

We now evaluate denial-of-service (DoS) attacks against mobile open ports and their effectiveness. Note that this analysis differs from those in Sec. V-A in that DoS attacks are typically possible even without exploiting any code-level vulnerabilities. Different from the traditional DoS attacks that often require a large number of bots (i.e., compromised computers), we show that DoS targeting mobile open ports can be performed by a single adversary using much less powerful devices (e.g., just one laptop), because the victim has much more limited computation, memory, and networking capabilities. Specifically, an adversary can first scan a WiFi/LTE network to identify targets (those with open ports) and then send large (number and/or size of) packets to deny victims from certain services or downgrade their quality of service. Therefore, this DoS attack is mostly effective for UDP ports that are open for communication purposes (recall that we have discovered 1,390 apps containing such ports; see Sec. III-C).

Fig. 7 shows the experimental results of DoS attacks against WeChat, YouTube, and AirDroid in an isolated WiFi network. The victim is a Samsung S6 edge+ phone, and we use `hping3` [19] on a MacBook Pro (with 2.9 GHz CPU and 16GB memory) to flood UDP ports opened by WeChat and YouTube as well as TCP ports opened by AirDroid. Fig. 7(a) shows that the throughput of WeChat's voice call drops to 50% when the attack launches at the 26-second mark, and is fully denied at around 50 seconds (forcing WeChat to automatically terminate the voice call). Fig. 7(b) and Fig. 7(c)

(a) A list of networks scanned.

(b) Detailed results of one scan.

Fig. 8: User interfaces in NetMon for network scans.

respectively show that the throughput of video streaming on YouTube and file transmission on AirDroid drop significantly right after the attack begins. Cellular networks, on the other hand, are less affected by such DoS attacks according to our tests, mainly because of their limited uplink throughput (attackers have to also use cellular to launch DoS as user devices in most cellular networks use private IP addresses; see our measurements in Sec. V-C). We expect the effectiveness of the attacks on cellular networks be significantly improved when client devices are assigned with public IP addresses and in the upcoming 5G era [1], [20].

C. Inter-device Connectivity Measurement

Most of the vulnerabilities and attacks demonstrated so far rely on connectivity to the victim device. To measure the extent to which such inter-device connectivity is allowed in public and private networks around the world, we embed a second service, the network testing component, in NetMon. Fig. 8 presents its two user interfaces, in which Fig. 8(a) shows a partial list of networks scanned and the detailed results are shown in Fig. 8(b). We can see that NetMon provides most of the functionality in typical network scanning apps (for attracting users to use this service in our app), and performs tests for the inter-device connectivity. The following three policies are tested in both WiFi and cellular networks, an effort never pursued before.

Inter-Pingable: whether an ICMP Ping packet could be transmitted from one device to another. This tests the basic inter-device connectivity of a network. To measure it, we leverage the `ping` program to issue ICMP requests to neighboring hosts whose IP addresses share a common 24-bit prefix (i.e., ping around 2^8 IP addresses).

Inter-TCPable and *Inter-UDPable*: whether a TCP/UDP packet could be transmitted from one device to another. To test them, NetMon launches TCP SYN and UDP

---

[5]A list of our original reports (in Chinese) can be found at https://tinyurl.com/opWooyun, and cached at https://tinyurl.com/opDropbox.

scans to all Pingable hosts. In each scan, NetMon sends a SYN packet or a small UDP packet to a target port number (randomly selected from the list of TCP/UDP open ports based on the results in Sec. III-C). If NetMon could receive a response (including failure packets, e.g., RST for TCP and ICMP port unreachable for UDP), we conclude that the inter-TCPable or inter-UDPable policy is employed.

Through the crowdsourcing deployment discussed in Sec. III-C, NetMon performs network connectivity tests in the wild. Similar to its port monitoring component, the network testing component is also very energy efficient — only 33.01KB consumed on average in one scan in an LTE network. By gathering and aggregating 6,391 network scans, we report the result and analysis on the inter-device connectivity for the first time for 224 cellular networks and 2,181 WiFi networks worldwide.

We find that almost 50% of the cellular networks (111 networks, 49.6%) allow their devices to ping each other, including AT&T, T-Mobile, Verizon Wireless, China Mobile, EE (in UK), Orange (in France), Airtel (in India), Celcom (in Malaysia), and SingTel (in Singapore). All of these 111 cellular networks also allow cross-device TCP packets, but the inter-UDPable tests fail in 14 networks, probably because they filter the ICMP unreachable messages sent by a closed UDP port. Note that we did not test networks that filter Ping packets while allowing TCP/UDP packets.

WiFi networks seem to have even worse security in terms of the inter-device connectivity in that 83.6% (1,823 out of 2,181) allow devices to ping each other. The inter-TCPable and inter-UDPable policies are also generally supported among the inter-Pingable WiFi networks with 95.6% and 88.3% success rates, respectively. The unsuccessful cases are probably due to their WiFi routers/APs filtering TCP RST and ICMP unreachable packets. University campus WiFi, enterprise office WiFi, airport WiFi, hotel WiFi, public transportation WiFi, and department store WiFi are among those that support inter-device connectivity. Allowing inter-device connectivity in these public-domain WiFi will facilitate remote open-port attacks.

Furthermore, 23 cellular networks (10% of all cellular networks tested) and 10 WiFi networks (including the "eduroam" WiFi provided by two universities in the US) assign public IP addresses to their users, which allow not only intra-network connectivity but connectivity from any host on the Internet. This is astonishing as it opens up exploit opportunities to any adversary on the Internet.

VI. MITIGATION SUGGESTIONS

To mitigate the threats of open ports, we propose counter-measures for different stakeholders in the Android ecosystem, including app developers, SDK producers, system vendors, and network operators.

App developers. The first thing developers need to assess is whether an open port is necessary. For example, for local inter-app communication, using `LocalServerSocket` [23] is more secure than establishing `ServerSocket`. If open ports are really needed, developers should minimize the attack surface by avoiding insecure coding behaviors as discussed

in Sec. IV-E and employ effective authentication against unintended access. Moreover, we suggest developers to use our NetMon app to evaluate a third-party SDK before including it.

SDK producers. Similarly, SDK producers should use open ports only when there are no better alternatives. For example, Facebook could reconsider its caching mechanism via an open port in its SDK. In particular, SDKs should abandon using open ports for the analytics purpose, because it is fundamentally insecure (see Sec. V-A).

System vendors. Besides having vendors assess open ports in their built-in apps carefully, Google can consider taking more proactive measures. For example, a new permission dedicated for the open port functionality, beyond the general `INTERNET` permission, could be introduced, so that both developers and users are better aware of it. As explained in Sec. IV-A, Google could also modify existing `ServerSocket` APIs to better cope with open ports in mobile environment.

Network operators. To stop remote open-port attacks, a quick mitigation is to restrict inter-device connectivity. For cellular or certain public WiFi networks (e.g., in airports), it is reasonable for them to prioritize the security for the safety of their users. Private WiFi networks (e.g., enterprise networks) may even leverage software-defined networking to better regulate such connectivity.

VII. RELATED WORK

Open port research. Traditionally, research on open ports focus on DoS attacks [64] and Internet scanning studies [53], [72]. This has been changed in the mobile era — more specific attacks [57], [79], [83] have been demonstrated on open ports of mobile apps. Another relevant study is on Android Unix domain sockets [71], as discussed in our background section (Sec. II). However, studies specifically focused on mobile open ports are not available until recently [52].

Although OPAnalyzer [52] is closely related to our paper, there are a number of significant differences. The foremost difference is in the objectives. We aim at a systematic understanding of open ports in the wild, while OPAnalyzer focused on detecting vulnerable apps that satisfy the taint-style code patterns. The approaches adopted are therefore very different. For example, there are no crowdsourcing or networking analysis in OPAnalyzer, and its static analysis does not resolve open-port parameters for an in-depth analysis (e.g., identifying SDKs and diagnosing insecure API usages) as our paper does. Furthermore, OPAnalyzer does not show any results for UDP ports and built-in apps.

Crowdsourcing for security. With the high popularity of mobile apps, it becomes realistic to leverage the crowd to discover security problems in the wild. By deploying NetMon to Google Play for a crowdsourcing study, we are among the first in this line of research. Other related works include Netalyzr [75] for studying middleboxes in cellular networks, FBS-Radar [56] for uncovering fake base stations in the wild, UpDroid [74] for monitoring sensitive API behaviors on non-rooted devices, and Haystack [69] for detecting mobile apps' privacy leakage via on-device app traffic analysis [80].

Android static analysis. Static analysis has been used extensively to understand the (in)security of Android apps.

They have been applied to malware analysis (e.g., [59], [68], [77], [85]), privacy leakage detection (e.g., [58], [60], [61]), vulnerability discovery (e.g., [47], [63], [81], [82], [84], [87]), and so on. Two analysis frameworks, FlowDroid [40] and Amandroid [76], are proposed to simplify tool development. For example, OPAnalyzer [52] is built upon Amandroid to forwardly track the flows between server sockets' `accept()` calls and sinks. However, it cannot analyze open-port parameters due to the lack of a backward-style parameter tracking engine. There are a few static tools for parameter analysis, but they cannot be applied to our problem due to limitations, such as no complete parameter representation in SAAF [51], no array handling [86], and no open port relevant API modeling [42]. We address these issues by introducing the backward slicing graph and semantic-aware constant propagation. Besides uncovering open-port parameters, our OPTool is also the first static analysis tool able to detect open-port SDKs in Android apps.

VIII. CONCLUSION

In this paper, we proposed the first open-port analysis pipeline to conduct a systematic study on open ports in Android apps and their threats. By first deploying a novel crowdsourcing app on Google Play for ten months, we observed the actual execution of open ports in 925 popular apps and 725 built-in apps. Crowdsourcing also provided us a more accurate view of the pervasiveness of open ports in Android apps: 15.3% discovered by our crowdsourcing as compared to the previous estimation of 6.8%. We then showed the significant presence of SDK-introduced open ports and identified insecure open-port API usages through the static analysis enhanced with open-port context and semantics. Furthermore, we uncovered five vulnerability patterns in open ports and reported vulnerabilities in popular apps and widely-embedded SDKs. The feasibility of remote open-port attacks in today's networks and the effectiveness of denial-of-service attacks were also experimentally evaluated. We finally discussed mechanisms for different stakeholders to mitigate open-port threats.

REFERENCES

[1] "5G Carrier Grade Wi-Fi: Addressing the Needs for Uplink Throughput, Dense Deployments and Cellular-like Quality," http://tinyurl.com/5gNeedOfUplink.

[2] "AlarmManager," https://developer.android.com/reference/android/app/AlarmManager.html.

[3] "AlarmManager change since Android 4.4," https://developer.android.com/about/versions/android-4.4.html#BehaviorAlarms.

[4] "Alibaba AMap SDK," http://lbs.amap.com/api/android-sdk/summary.

[5] "Android Fragmentation Report August 2015," https://opensignal.com/reports/2015/08/android-fragmentation/.

[6] "Android Logcat," https://developer.android.com/reference/android/util/Log.html.

[7] "AndroidVideoCache," https://github.com/danikula/AndroidVideoCache.

[8] "Aol AdTech SDK," http://www.aolpublishers.com/support/documentation/mobile/ads.md.

[9] "Apache Cordova SDK," https://cordova.apache.org/docs/en/latest/guide/platforms/android/.

[10] "BluetoothSocket," https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html.

[11] "Changes to Device Identifiers in Android O," https://android-developers.googleblog.com/2017/04/changes-to-device-identifiers-in-android.html.

[12] "Configure Apps with Over 64K Methods," https://developer.android.com/studio/build/multidex.html.

[13] "Corona Game Engine SDK," https://docs.coronalabs.com/native/android/index.html.

[14] "CyberGarage UPnP SDK," https://github.com/cybergarage/cybergarage-upnp.

[15] "Disassemble Android dex files," http://blog.vogella.com/2011/02/14/disassemble-android-dex/.

[16] "The ephemeral port range," http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html.

[17] "Facebook Audience Network SDK," https://developers.facebook.com/docs/audience-network/android-native.

[18] "Getui Push SDK," http://docs.getui.com/mobile/android/overview/.

[19] "hping3," http://linux.die.net/man/8/hping3.

[20] "Huawei's 5G Vision: 100 Billion connections, 1 ms Latency, and 10 Gbps Throughput," http://www.huawei.com/minisite/5g/en/defining-5g.html.

[21] "JEB Decompiler for Android," https://www.pnfsoftware.com/jeb/android.

[22] "LG Smartshare," http://www.lg.com/support/smart-share.

[23] "LocalServerSocket — Android Developers," https://developer.android.com/reference/android/net/LocalServerSocket.html.

[24] "Millennial Ad SDK," http://docs.onemobilesdk.aol.com/android-ad-sdk/.

[25] "MIT App Inventor," https://github.com/mit-cml/appinventor-sources.

[26] "NFCSocket: Android Play Near Flied Communication in the Socket way," https://github.com/Chrisplus/NFCSocket.

[27] "Nmap: the network mapper," https://nmap.org/.

[28] "PhoneGap SDK," https://phonegap.com/.

[29] "proc(5): process info pseudo-file system - Linux man page," http://linux.die.net/man/5/proc.

[30] "Samsung Accessory SDK," http://developer.samsung.com/galaxy/accessory.

[31] "Selenium - web browser automation," http://docs.seleniumhq.org/.

[32] "Setting the Record Straight on Moplus SDK and the Wormhole Vulnerability," http://tinyurl.com/wormholevulnerability.

[33] "Sony DLNA Support," https://esupport.sony.com/US/p/support-info.pl?info_id=884.

[34] "The tcp_states.h file in Linux kernel," http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp_states.h?id=HEAD.

[35] "Tcp/udp port numbers used in samsung system," ftp://81.24.117.226/Samsung/OS%20ports/attachment.pdf.

[36] "Tencent XG Push SDK," http://docs.developer.qq.com/xg/.

[37] "Titanium Mobile Intro Series: Fastdev for Android," http://www.appcelerator.com/blog/2011/05/titanium-mobile-intro-series-fastdev-for-android/.

[38] "Yandex Metrica SDK," https://github.com/yandexmobile/metrica-sdk-android.

[39] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. ACM MSR*, 2016.

[40] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM PLDI*, 2014.

[41] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proc. ACM CCS*, 2016.

[42] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-Droid: Leveraging Android app analysis with static slice optimization," in *Proc. ACM AsiaCCS*, 2016.

[43] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for Android using systematic mutation," in *Proc. USENIX Security*, 2018.

[44] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *Proc. ISOC NDSS*, 2015.

[45] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. ACM MobiSys*, 2011.

[46] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about Android (un)packers: A systematic study based on whole-system emulation," in *Proc. ISOC NDSS*, 2018.

[47] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proc. ACM CCS*, 2013.

[48] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proc. ISOC NDSS*, 2011.

[49] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. USENIX Security*, 2011.

[50] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. ACM WiSec*, 2012.

[51] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proc. ACM SAC (Symposium on Applied Computing)*, 2013.

[52] Y. Jia, Q. Chen, Y. Lin, C. Kong, and Z. Mao, "Open doors for Bob and Mallory: Open port usage in Android apps and security implications," in *Proc. IEEE EuroS&P*, 2017.

[53] M. Johns, S. Lekies, and B. Stock, "ZMap: Fast Internet-wide scanning and its security applications," in *Proc. USENIX Security*, 2013.

[54] O. Lhotak and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proc. Springer Compiler Construction*, 2003.

[55] L. Li, T. F. Bissyande, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," in *Proc. ACM ISSTA*, 2016.

[56] Z. Li, W. Wang, C. Wilson, J. Chen, C. Qian, T. Jung, L. Zhang, K. Liu, X. Li, and Y. Liu, "FBS-Radar: Uncovering fake base stations at scale in the wild," in *Proc. ISOC NDSS*, 2017.

[57] C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your Android screen for secrets," in *Proc. ISOC NDSS*, 2014.

[58] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. ISOC NDSS*, 2015.

[59] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building markov chains of behavioral models," in *Proc. ISOC NDSS*, 2017.

[60] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "UIPicker: User-input privacy identification in mobile applications," in *Proc. USENIX Security*, 2015.

[61] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps," in *Proc. ISOC NDSS*, 2018.

[62] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. USENIX Security*, 2013.

[63] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes, "The rise of the citizen developer: Assessing the security impact of online app generators," in *Proc. IEEE Symposium on Security and Privacy*, 2018.

[64] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the DoS and DDoS problems," in *Proc. ACM CSUR*, 2007.

[65] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proc. ISOC NDSS*, 2014.

[66] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proc. ACM ISSTA*, 2018.

[67] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, "DyDroid: Measuring dynamic code loading and its security implications in Android applications," in *Proc. IEEE DSN*, 2017.

[68] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," in *Proc. ISOC NDSS*, 2016.

[69] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *Proc. ISOC NDSS*, 2018.

[70] J. Schiller, F. Turbak, H. Abelson, J. Dominguez, A. McKinney, J. Okerlund, and M. Friedman, "Live programming of mobile apps in App Inventor," in *Proc. ACM Workshop on Programming for Mobile & Touch*, 2014.

[71] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The misuse of Android Unix domain sockets and security implications," in *Proc. ACM CCS*, 2016.

[72] D. Springall, Z. Durumeric, and J. A. Halderman, "FTP: The forgotten cloud," in *Proc. IEEE DSN*, 2016.

[73] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in Android Ad libraries," in *Proc. IEEE Mobile Security Technologies (MoST)*, 2012.

[74] X. Tang, Y. Lin, D. Wu, and D. Gao, "Towards dynamically monitoring Android applications on non-rooted devices in the wild," in *Proc. ACM WiSec*, 2018.

[75] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson, "Beyond the radio: Illuminating the higher layers of mobile networks," in *Proc. ACM MobiSys*, 2015.

[76] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM CCS*, 2014.

[77] M. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. ISOC NDSS*, 2016.

[78] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in Android with TIRO," in *Proc. USENIX Security*, 2018.

[79] D. Wu and R. K. C. Chang, "Indirect file leaks in mobile applications," in *Proc. IEEE Mobile Security Technologies (MoST)*, 2015.

[80] D. Wu, R. K. C. Chang, W. Li, E. K. T. Cheng, and D. Gao, "MopEye: Opportunistic monitoring of per-app mobile network performance," in *Proc. USENIX Annual Technical Conference*, 2017.

[81] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared SDK versions and their consistency with API calls in Android apps," in *Proc. Conference on Wireless Algorithms, Systems, and Applications (WASA)*, 2017.

[82] D. Wu, X. Luo, and R. K. C. Chang, "A sink-driven approach to detecting exposed component vulnerabilities in Android apps," *CoRR*, vol. abs/1405.6282, 2014.

[83] L. Xing, X. Bai, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu, "Staying secure and unprepared: Understanding and mitigating the security risks of Apple ZeroConf," in *Proc. IEEE Symposium on Security and Privacy*, 2016.

[84] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postMessage enabled mobile applications," in *Proc. IEEE Symposium on Security and Privacy*, 2018.

[85] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM CCS*, 2014.

[86] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in Android apps," in *Proc. ACM WiSec*, 2015.

[87] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proc. IEEE Symposium on Security and Privacy*, 2019.