5-2019

On-the-fly Android static analysis with applications in vulnerability discovery

Daoyuan WU
Singapore Management University, dywu.2015@phdis.smu.edu.sg

ON-THE-FLY ANDROID STATIC ANALYSIS WITH APPLICATIONS IN VULNERABILITY DISCOVERY

DAOYUAN WU

SINGAPORE MANAGEMENT UNIVERSITY

2019

On-the-fly Android Static Analysis with Applications in Vulnerability Discovery

by
Daoyuan Wu

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Debin GAO (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Robert H. DENG (Co-supervisor)
Professor of Information Systems
Singapore Management University

Yingjiu LI
Associate Professor of Information Systems
Singapore Management University

Rocky K. C. CHANG
Associate Professor at the Department of Computing
The Hong Kong Polytechnic University

Singapore Management University
2019

I hereby declare that this PhD dissertation is my original work
and it has been written by me in its entirety.
I have duly acknowledged all the sources of information
which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree
in any university previously.

Daoyuan Wu

29 May 2019

On-the-fly Android Static Analysis with Applications in Vulnerability Discovery

Daoyuan Wu

Abstract

Static analysis is a common program analysis technique extensively used in the software security field. Widely-used static dataflow analysis tools for Android, e.g., Amandroid and FlowDroid, perform a whole-app analysis that starts from all entry points and ends in all reachable code nodes. Such analysis is comprehensive yet at the cost of huge overheads, and is therefore difficult to keep pace with modern apps that are constantly expanding their app sizes. Since security studies are usually interested in only a small portion of codes that involve the flows of security-sensitive sink APIs (e.g., the `sendTextMessage()` API), it is desirable to have a more security-oriented tool that can perform an on-demand analysis of the selected sinks.

In this dissertation, we make a first attempt to explore a novel on-demand analysis that does not generate a whole-app call graph but creatively leverages bytecode search to guide inter-procedural analysis on the fly or just in time. We develop such *on-the-fly* static analysis into a novel tool, called BackDroid, for efficient and effective targeted security vetting of Android apps. Specifically, BackDroid employs a novel on-the-fly backward search technique to search over Java polymorphism, threads, implicit callback flows, and Android inter-component communication. Furthermore, BackDroid performs backward taint analysis of sink API parameters to generate a backward slicing graph (BSG), and conducts forward points-to analysis to propagate dataflow facts on top of the generated BSGs.

This dissertation further explores how the core technique of on-the-fly static analysis in BackDroid can enable different vulnerability studies and their corresponding new findings. Following this direction, we first perform an evaluation study by applying BackDroid to detect crypto and SSL misconfigurations in modern

apps and comparing it with the state-of-the-art Amandroid tool. The results show that BackDroid achieves a much better performance than Amandroid, ten times faster on average, and at the same time, maintains similar detection effectiveness.

In the second study, we explore how BackDroid can facilitate a systematic security study of open ports in Android apps. To this end, we first design an on-device crowdsourcing app to discover 2,778 open-port apps, including 925 popular apps and 725 built-in system apps. We then enhance BackDroid with the SDK identification capability and open-port related semantics, e.g., random port number via `Math.random()` and IP address array like `byte[]{127,0,0,1}`, to detect insecure open ports. Our diagnosis shows that 61.8% of the 1,520 open-port apps on Google Play are solely due to embedded SDKs and 20.7% suffer from insecure API usages. We further perform three in-depth security assessments, including vulnerability analysis revealing five vulnerability patterns, denial-of-service attack evaluation, and network feasibility measurement of the remote open-port attacks.

The first two studies focus on the dataflow analysis of one or two particular kinds of sink APIs each. We further explore how on-the-fly bytecode search can benefit a study of measuring the inconsistency between declared SDK (or DSDK) versions in Android manifest and multiple API calls in app code. We thus customize a lightweight version of BackDroid by focusing on the control-flow information, i.e., those SDK conditional statements, of the searched sink APIs, and employ it to analyze the SDK-API inconsistency for over 22K modern popular apps. We find that (i) $\sim$50% apps under-set the minimum DSDK versions and could incur crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above; and (ii) $\sim$2% apps, due to under-claiming the targeted DSDK versions, are potentially exploitable.

To conclude, this dissertation makes this core contribution: *On-the-fly Android static analysis guided by bytecode search can efficiently and effectively analyze the security of modern apps. It enables us to perform vulnerability studies with different kinds of sink analysis requirements, and to obtain new findings on crypto and SSL/TLS misconfigurations, insecure open ports, and SDK-API inconsistency.*

Table of Contents

List of Figures

List of Tables

List of Publications

The following papers are the partial outputs of my four-year PhD study at SMU.

Conference Papers

Daoyuan Wu, Debin Gao, Robert H. Deng, and Rocky K. C. Chang. BackDroid: On-the-fly Static Dataflow Analysis via Bytecode Search for Targeted Security Vetting of Android Apps. In submission to a top-tier security conference.

Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In *Proc. ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, Auckland, New Zealand, July 2019.

Shiwei Zhang, Weichao Li, **Daoyuan Wu**, Bo Jin, Rocky K. C. Chang, Debin Gao, Yi Wang, and Ricky K. P. Mok. An Empirical Study of Mobile Network Behavior and Application Performance in the Wild. In *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*, Phoenix, USA, June 2019.

Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment. In *Proc. ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, February 2019.

Xiaoxiao Tang, Yan Lin, **Daoyuan Wu**, and Debin Gao. Towards Dynamically Monitoring Android Applications on Non-rooted Devices in the Wild. In *Proc. ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Stockholm, Sweden, June 2018.

Daoyuan Wu, Yao Cheng, Debin Gao, Yingjiu Li, and Robert H. Deng. SCLib: A Practical and Lightweight Defense Against Component Hijacking in Android Applications. In *Proc. ACM Conference on Data and Applications Security and Privacy (CoDASPY)*, Tempe, USA, March 2018.

Daoyuan Wu, Rocky K. C. Chang, Weichao Li, Eric K. T. Cheng, and Debin Gao. MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance. In *Proc. USENIX Annual Technical Conference (ATC)*, Santa Clara, USA, July 2017.

Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. Measuring the Declared SDK Versions and Their Consistency with API Calls in Android Apps. In *Proc. Springer Conference on Wireless Algorithms, Systems, and Applications (WASA)*, Guilin, China, June 2017.

Weichao Li, **Daoyuan Wu**, Rocky K. C. Chang, and Ricky K. P. Mok. Demystifying and Puncturing the Inflated Delay in Smartphone-based WiFi Network Measurement. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Irvine, US, December 2016.

Yu Liang, Xinjie Ma, **Daoyuan Wu**, Xiaoxiao Tang, Debin Gao, Guojun Peng, Chunfu Jia, and Huanguo Zhang. Stack Layout Randomization with Minimal Rewriting of Android Binaries. In *Proc. Conference on Information Security and Cryptology (ICISC)*, Seoul, Korea, November 2015.

Journal Articles

Daoyuan Wu, Debin Gao, David Lo. Scalable Online Vetting of Android Apps for Measuring Declared SDK Versions and Their Consistency with API Calls. Under review of *Springer Empirical Software Engineering*, 2019.

Weichao Li, **Daoyuan Wu**, Rocky K. C. Chang, and Ricky K. P. Mok. Toward Accurate Network Delay Measurement on Android Phones. In *IEEE Transactions on Mobile Computing*, vol. 17, no. 3, March 2018.

Posters

Daoyuan Wu, Weichao Li, Rocky K. C. Chang, and Debin Gao. MopEye: Monitoring Per-app Network Performance with Zero Measurement Traffic. In *ACM CoNEXT Student Workshop*, Heidelberg, Germany, December 2015.

Acknowledgments

It has been around four years since I came to study at SMU in August 2015. I am lucky, to be able to study at such a convenient campus with full financial support. During this period, I have met and known the following people, who help me grow in many ways. I would like to give my sincere thanks to all of them.

Foremost, I feel very honored to have Prof. Debin Gao and Prof. Robert H. Deng as my advisors. As my chief supervisor, Debin gave me sufficient freedom to explore research topics that I am interested in, discussed with me in countless weekly research meetings, provided enough research support to my works, and gave me helpful revisions and comments to my papers and presentations, and wrote reference letters for all kinds of my applications. Robert is also always there whenever I need help on research or other matters. His high-level thoughts broaden my views, his critical thinking improves my works, and his professional advices help shape my academic career. It is my fortune to have both of them on my PhD journey.

Special thanks are given to the to the rest of members in my Dissertation Committee, Prof. Yingjiu Li and Prof. Rocky K. C. Chang. Besides thanking them for their invaluable comments on this dissertation, I thank Prof. Li for a number of research discussions with me and his writing guidance on the CODASPY'18 paper, and thank Prof. Chang for his endless help since I started my MPhil study at PolyU.

I would like to also thank other professors at SMU: Prof. David Lo for his contribution to the DSDK paper, Prof. Lingxiao Jiang for his discussion on the VulnBot paper, Prof. Hwee Hwa Pang, Prof. Younsoo Kim, Prof. Pradeep Varakantham for their interesting courses, and also Prof. Xuhua Ding and Prof. Baihua Zheng.

I am very grateful to the past and current members in the Cybersecurity Research Group at SMU: Jiayun Xu and Ximing Liu for their inputs to the WASA'17 paper and invaluable friendship; Xiaoxiao Tang, Yan Lin, Yu Liang, Xinjie Ma, Baihe Jiang, and Weijie Liu for their collaboration in our team; Dr. Yao Cheng for her contribution to the CODASPY'18 paper; Dr. Ke Xu, Dr. Su Mon Kywe, Dr. Yan Li for their research discussion with me; and Dr. Siqi Ma, Dr. Siqi Zhao, Jiaqi Hong, Yunshi Lan, Xiongwei Wu, and many other SIS students for their friendship. Additionally, I thank my several external collaborators, Eric K. T. Cheng, En He, Dr. Weichao Li, and Dr. Ricky K. P. Mok.

My deepest thanks go to my family, especially my wife for the happiness, encouragement, help, and love she gives to me along my PhD journey in Singapore. Special thanks always go to my parents for their constant love and support.

Last but not least, I thank administrative staffs, especially SEOW Pei Huan, YEO Yar Ling, ONG Chew Hong, and Caroline TAN, in the General Office of School of Information Systems for their kind help on various administrative matters. Thanks also go to all the anonymous reviewers of my published papers for their professional reviews.

To my wife, Ning Liu, and my parents, Zhigang Wu and Jie Tang.

Chapter 1

Introduction

Static analysis is a common program analysis technique extensively used in the software security field. Widely-used static dataflow analysis tools for Android, e.g., FlowDroid [69] and Amandroid [132], perform a whole-app analysis that starts from all entry points and ends in all reachable code nodes. Such analysis is comprehensive yet at the cost of huge overheads. For example, a data-flow mining study [71] based on FlowDroid had to use a compute server with 730 GB of RAM and 64 Intel Xeon CPU cores. Even with such a powerful configuration, they stated that "*the server sometimes used all its memory, running on all cores for more than 24 hours to analyze one single Android app*".

By nature, it is extremely challenging for the whole-app analysis to deal with large apps or third-party library codes, and this is why existing studies often selected small apps (e.g., apps under 5MB in AppContext [149], and apps from 14KB to 461KB in TriggerScope [88, 116]) and ignored library code (e.g., Amandroid [133] by default skipped the analysis of 139 popular third-party libraries) for their analysis. However, modern apps are constantly expanding their sizes over the years. According to our measurement (in Chapter 3), the average size of popular apps expands around three times from 13.8MB in 2014 to 42.6MB in 2018. Hence, an on-demand analysis is necessary to keep pace with this trend in modern apps.

Fortunately, security studies are usually interested only in a small portion of

codes that involve the flows of security-sensitive sink APIs. For example, Android malware detection [153] is mostly interested in the sink APIs that can make security harms (e.g., the `sendTextMessage()` API), and vulnerability analysis works often just need to spot a particular pattern from the entire app code [85, 106, 154]. Therefore, it is possible for security-oriented tools to perform an on-demand analysis of the selected sinks.

1.1 BackDroid: On-the-fly Android Static Analysis

In this dissertation, we make a first attempt to explore a novel on-demand analysis that does not generate a whole-app call graph but creatively leverages bytecode search to guide inter-procedural analysis on the fly or just in time. We develop such *on-the-fly* static analysis into a novel tool, called BackDroid, for efficient and effective targeted security vetting of Android apps. Specifically, BackDroid leverages bytecode search to not only initiate the analysis directly from given sinks but also creatively guide the backward inter-procedural analysis step by step. As a result, generating an expensive whole-app call graph is no longer needed in BackDroid, which makes the required CPU and memory resources always under control regardless of app size. Such a novel design, however, requires us to solve a number of unique technical issues that never appear before. Notably, it is challenging to perform effective bytecode search over Java polymorphism (e.g., parent classes and interfaces), threads, implicit callback flows, and Android ICC (inter-component communication). Moreover, our search is conducted in a backward manner, which further increases the difficulty since it is the reverse of normal program execution.

To enable BackDroid's inter-procedural analysis, we propose a novel on-the-fly backward search technique that comprises of several parts. First, we present a basic method signature based search that constructs appropriate search signatures to directly locate caller methods for static, private, and constructor callee methods. It can also search over child classes. However, this basic search is not effective to ad-

dress complex situations with super classes, interfaces, and implicit Java/Android flows. We thus further propose an advanced search mechanism. Specifically, instead of directly searching caller methods, we first search the callee class' object constructor(s) that can be accurately located. After that, starting from those constructors, we then perform instant forward analysis until reaching caller methods. Furthermore, for Android ICC, we conduct a two-time search for both ICC calls and parameters and merge their search results. Alongside the inter-procedural analysis enabled by bytecode search, BackDroid performs backward taint analysis of sink API parameters to generate a new structure called backward slicing graph (BSG), and further conducts forward points-to analysis to propagate dataflow facts on top of the generated BSGs.

Compared with existing Android static analysis tools (e.g., the state-of-the-art Amandroid), the main advantage of BackDroid is that it can analyze all modern apps and third-party library codes regardless of their sizes. In contrast, Amandroid could fail on a significant portion of modern apps, such as timed out on 69 out of 144 modern apps analyzed, as we will introduce in Section 1.2. Hence, BackDroid and its core technique of on-the-fly static analysis build a foundation for researchers to thoroughly analyze security problems and obtain new findings that exiting tools would miss. In principle, BackDroid can facilitate the analysis of any problems due to misusing sink APIs, but it may require different customization for each specific problem. As a result, although this dissertation targets at Android app vulnerabilities, BackDroid has the potential to also investigate Android malware/adware.

Following this direction, we explore how the core technique of on-the-fly static analysis in BackDroid can enable different vulnerability studies on Android and their corresponding new findings. To this end, we select three vulnerability analysis problems on Android as three representatives, since they require different extents of BackDroid customization in their methodology. As shown in Figure 1.1, we conduct the following three vulnerability studies that have different analysis requirements according to their respective sink APIs:

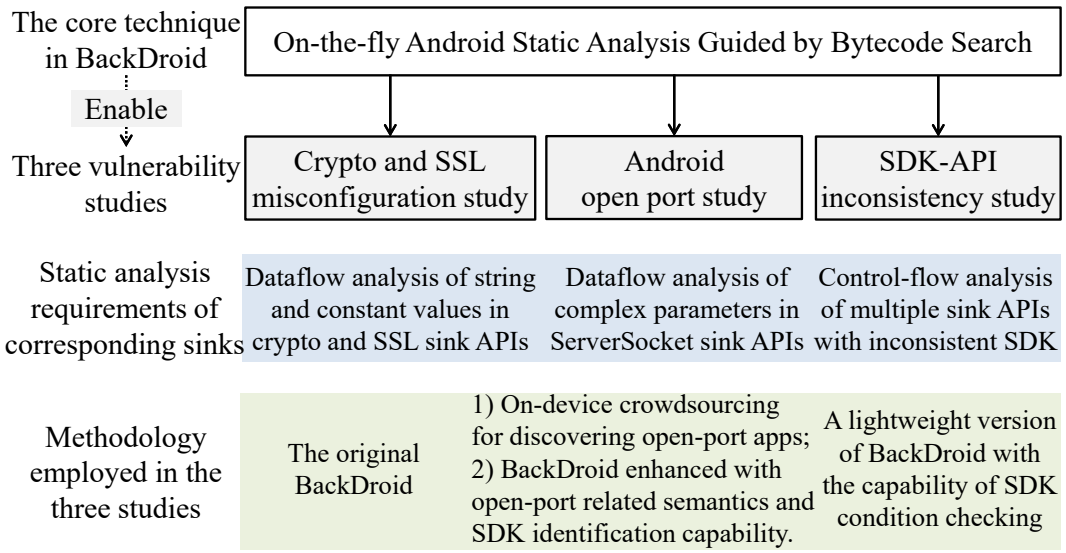


Figure 1.1: An overview of techniques and studies conducted in this dissertation.

- *The crypto and SSL misconfiguration study*: This study can directly use BackDroid without particular customization, because the crypto and SSL/TLS sink APIs use only string (e.g., "AES/ECB/NoPadding") and constant (e.g., `ALLOW_ALL_HOSTNAME_VERIFIER`) values that are by default supported by BackDroid. Hence, we also use this vulnerability study as an evaluation of BackDroid and compare it with the state-of-the-art Amandroid tool [132].

- *The Android open port study*: The `ServerSocket` sink APIs in this study involve complex parameters that BackDroid requires relevant semantics to resolve. For example, BackDroid needs to understand the semantic of `Math.random()` to conclude a random port number used, and to know how to assemble an array like `byte[]{127,0,0,1}` into a local loopback IP address. Moreover, SDK identification is also required besides the dataflow analysis of sink APIs' parameters. The enhanced BackDroid can deliver a security diagnosis of open ports, but for a systematic study, we still need to combine it with a crowdsourcing approach for effectively discovering open-port apps in the wild, as well as in-depth security assessments.

- *The SDK-API inconsistency study*: The first two studies focus on the dataflow

analysis of one or two particular kinds of sink APIs each, whereas this study involves the control-flow analysis of multiple sink APIs. Specifically, it aims to measures the inconsistency between declared SDK (or DSDK) versions in Android manifest and API calls in app code, which requires us to probe many potentially inconsistent APIs (i.e., APIs with a SDK level inconsistent with DSDK) and identify those not guarded with SDK version checking. Hence, we customize a lightweight version of BackDroid by focusing on the control-flow SDK version checking information of searched sink APIs.

With all these works, this dissertation makes this core contribution: *On-the-fly Android static analysis guided by bytecode search can efficiently and effectively analyze the security of modern apps. It enables us to perform vulnerability studies with different kinds of sink analysis requirements, and to obtain new findings on crypto and SSL/TLS misconfigurations, insecure open ports, and SDK-API inconsistency.*

In the rest of this chapter, we continue to introduce more details about the three vulnerability studies and their new findings from Section 1.2 to Section 1.4, and then list an outline of this dissertation in Section 1.5.

1.2 Detecting Crypto and SSL Misconfigurations

In this section, we introduce the crypto and SSL misconfiguration study that BackDroid by default supports. Crypto and SSL/TLS misconfigurations are two known yet serious vulnerabilities commonly appeared in Android apps [83, 86]. In both cases, the root cause is due to using insecure parameters in their corresponding sink APIs. Specifically, the insecure ECB mode parameter, either explicit (e.g., "AES/ECB/NoPadding") or implicit (e.g., "AES"), is used to create the `javax.crypto.Cipher` instance [83, 107]. Similarly, the insecure verifier parameter, such as `ALLOW_ALL_HOSTNAME_VERIFIER`, is used in `setHostnameVerifier()` [86]. Since the state-of-the-art Amandroid [132, 133] tool also supports the detection of these two misconfigurations, we thus make

a comparison on the efficiency and efficacy of both tools in analyzing modern apps.

To perform such an evaluation study, we first select 3,178 popular apps that not only have at least one million installs each but also get updated recently in 2018, and then pre-process them to obtain 144 apps with all relevant sink APIs as our dataset so that Amandroid would not waste its analysis even without a bytecode search capability. We use a default parameter configuration of Amandroid and run both tools on a machine with 8-core Intel i7-4790 CPU and 16GB of physical memory, a memory configuration often used in many academic Android app analysis works (e.g., [116, 124, 149, 151]). Moreover, we give Amandroid sufficient running time with a timeout of 100 minutes per app, while BackDroid's configuration is only 20 minutes per app.

Our evaluation shows that BackDroid achieves a much better performance while maintaining similar, or even better in some cases, detection effectiveness as Amandroid. First, BackDroid's overall performance is around ten times faster than that in Amandroid, requiring only 3.3m (or minutes) and 1.1m for the average and median analysis time, respectively, whereas that in Amandroid is 24.7m and 15.5m, respectively. In particular, BackDroid finished the analysis of 108 (75%) apps within 10 minutes each, whereas only 24 (16.7%) apps were completed by Amandroid in the same 10-minute slot per app. Moreover, BackDroid still achieves similar detection effectiveness for the 25 vulnerable apps detected by Amandroid: 22 of them were also uncovered by BackDroid and the rest of three failures are due to a third-party library called `com.skt.arm.ArmSeedCheck`. Specifically, this library uses an AIDL (Android Interface Definition Language) function that Amandroid considers whereas BackDroid does not. Furthermore, BackDroid discovered 18 additional vulnerable apps that were missed by Amandroid: 10 of them were due to Amandroid's default configuration of skipping the analysis of some popular libraries and static initializers while the rest of eight were timed out in Amandroid. This strongly demonstrates that on-the-fly static analysis in BackDroid not only shortens the analysis time but also enables new detection results that would otherwise be missed.

1.3 Analyzing the Security of Android Open Ports

In this section, we introduce our second vulnerability study that aims to systematically analyze the security of open ports in Android apps. Open TCP/UDP ports are traditionally used by servers to provide application services, but they also exist in many Android apps as shown in our study. Moreover, a few recent studies have shown that these open ports are susceptible to various attacks. For example, Lin et al. [103] demonstrated the insecurity of local TCP open ports used in non-rooted Android screenshot apps, and Wu et al. [139] found that the top ten file-sharing apps on Android and iOS typically do not authenticate traffic to their ports.

As explained in Section 1.1 (see Figure 1.1), we need not only an enhanced version of BackDroid for dataflow analysis of complex parameters in open ports' `ServerSocket` sink APIs but also an effective crowdsourcing approach for discovering open-port apps in the wild. In this way, we build the first analysis pipeline that covers the open port discovery, diagnosis, and security assessment.

Our study starts with a crowdsourcing discovery of open-ports apps in the wild. Specifically, we design and deploy an on-device monitoring app and a server-side analytic engine to continuously monitor Android apps' open ports without user intervention. Our Android app, NetMon, has been available on Google Play for an IRB-approved crowdsourcing study since October 2016. In this dissertation, we base our analysis on the data over ten months, 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 phone manufacturers.

While crowdsourcing is effective in port discovery, it does not reveal the code-level information for more in-depth understanding and diagnosis. We then enhance BackDroid in the following two aspects to diagnose 1,027 TCP open-port apps that can be retrieved from the AndroZoo repository [65]. First, we add the SDK identi-

fication capability into BackDroid to identify 13 popular open-port SDKs and show that 61.8% of the open-port apps are solely due to these embedded SDKs, among which Facebook SDK is the major contributor. Second, we supply BackDroid with open-port related semantics, e.g., random port number via `Math.random()` and IP address array like `byte[]{127,0,0,1}`, to reveal that 20.7% make convenient but insecure API calls, unnecessarily increasing their attack surfaces.

As the last part of our analysis pipeline, we perform three in-depth security assessments of open ports. First, we perform vulnerability pattern analysis and identify five kinds of open-port vulnerabilities, three of which were not reported previously, in popular apps, such as Instagram, Samsung Gear, Skype, and the widely-embedded Facebook SDK and Alibaba SDK. Second, we experimentally evaluate the effectiveness of a generic denial-of-service attack against mobile open ports, and show that it can significantly downgrade YouTube's video streaming, WeChat's voice call, and AirDroid's file transmission via their open ports. Third, to understand the effectiveness of launching remote open-port attacks in real networks, we conduct inter-device connectivity tests in 224 cellular networks and 2,181 WiFi networks worldwide and 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.

1.4 Measuring the SDK-API Inconsistency

In this section, we introduce our third study on measuring the inconsistency between declared SDK versions in Android manifest and API calls in app code. Specifically, to better manage the application's compatibility across multiple platform versions, Android allows apps to declare the supported platform SDK versions in their manifest files. We term these declared SDK versions as `DSDK` versions. The `DSDK` mechanism is a modern software mechanism with which few systems are equipped until Android. Nevertheless, so far it receives little attention and few understandings are known about its effectiveness of the `DSDK` mechanism.

Compared with the first two studies that focus on the dataflow analysis of one or two particular kinds of sink APIs each, our SDK-API study involves the control-flow analysis of multiple sink APIs. Specifically, it requires us to probe many potentially inconsistent APIs (i.e., APIs with a SDK level inconsistent with `DSDK`) and identify those not guarded with SDK version checking (developers can use such checking to invoke an API only in certain Android platforms). According to our measurement in Chapter 5, 22.2% of modern apps invoke more than 10 sink APIs each that are inconsistent with their `DSDK` versions. To address this challenge, we customize a lightweight version of BackDroid that operates on the original bytecode level and leverages lightweight bytecode search with the capability of SDK conditional statement checking to detect the `DSDK` inconsistency in a large number of modern apps. By focusing on the control-flow information of searched sink APIs, our lightweight BackDroid preserves a scalability suitable for online vetting: the median and average time for analyzing an app in our dataset is only 4.75s and 5.39s, respectively.

We then employ this custom BackDroid to analyze the SDK-API inconsistency for 22,687 modern popular apps. Our study obtains the following three findings:

- First, 4.76% apps still do not claim the targeted `DSDK` attribute, causing their `DSDK` versions to be by default set to the minimum `DSDK` attribute, although this percentage has significantly dropped from 2015 to 2018.
- Second, around 50% apps under-set the `minSdkVersion` value, causing them to crash when running on lower versions of Android platforms. These runtime crashes allow an adversary to easily launch the app-level denial-of-service attack. Fortunately, a further analysis reveals that only 11.3% apps could crash on Android 6.0 and above.
- Third, around 2% apps still set an outdated `targetSdkVersion` attribute when a common WebView API is vulnerable, making them exploitable by remote code execution. In particular, around a half of these vulnerable apps invoke the vulnerable API call because of their embedded third-party libraries.

1.5 Organization of the Dissertation

The reminder of this dissertation is organized as follows. Chapter 2 reviews the related literature. Chapter 3 presents BackDroid's design and implementation and its evaluation study in detecting crypto and SSL/TLS misconfigurations. Chapter 4 presents our study on the security of open ports in Android apps, and Chapter 5 further studies the inconsistency between declared SDK versions and API calls. Finally in Chapter 6, we conclude this dissertation and outline future research directions.

Chapter 2

Literature Review

In this chapter, we review research works in the literature that are closely related to this dissertation. Specifically, we first summarize prior developments of Android static analysis techniques in Section 2.1, and then examine dedicated works that are relevant to our three vulnerability studies from Section 2.2 to Section 2.4.

2.1 Android Static Analysis Techniques

In this section, we review the major developments of Android static analysis techniques over the past ten years. Specifically, in Section 2.1.1, we first review two pre-processing techniques that are commonly used in many static analysis tools. After that, in Section 2.1.2, we present the core techniques of Android static analysis, namely control and dataflow analysis. Finally, we explain some difficult technical issues and review how prior works attempted to handle them in Section 2.1.3.

2.1.1 Manifest and Bytecode Preprocessing

An Android app consists of manifest, bytecode, and resource files. Before performing the actual control and data flow analysis, a static analysis tool needs to first pre-process manifest and bytecode.

Manifest analysis. A manifest is a binary-form XML file that describes all

component information of the app. A few early works focused only on the manifest files in Android apps, because they are relatively easy to analyze. For example, Chan et al. [78] analyzed the component exposure information in manifest to infer privilege escalation vulnerabilities [80]. Since then, manifest analysis had been used in nearly all Android static analysis. A common way of performing manifest analysis is to leverage a tool called `apktool` [11] to uncompress the entire app and decode the manifest. However, as we will explain in Chapter 5, it is not robust enough for analyzing modern apps. To address this limitation, we propose a new way of manifest analysis that leverages `aapt` (Android Asset Packaging Tool) [2], which has been successfully used in all our three studies in this dissertation. Besides analyzing manifest for pre-processing, Xu et al. [146] recently showed that manifest files can be well trained with deep learning to effectively detect Android malware.

Converting bytecode to IR. To launch meaningful analysis of Android bytecode, an important step is to convert them into a suitable intermediate representation (IR). Two commonly used tools are dex2jar [22] and baksmali [53]. The former converts Android bytecode to Java bytecode, whereas the latter translates bytecode into a plaintext format called smali. Besides these two industrial tools, the academia community also proposed a tool called Dare [113] for converting Android bytecode to Java bytecode, and a tool called Dexpler [74] that translates Android bytecode directly to Soot Jimple IR. In this dissertation, we use dex2jar in our first two studies and directly work on the bytecode level in the third study because it employs only the lightweight bytecode search.

2.1.2 Control and Dataflow Analysis

With a suitable IR, researchers can launch various analysis on Android bytecode. They can be roughly classified into control flow based reachability analysis and dataflow based taint analysis, or even combining both.

Reachability analysis. RiskRanker [92] and Woodpecker [91] are the two pio-

neer works using reachability analysis for malware detection and vulnerability discovery, respectively. Both works tested the reachability from entry points to the selected sink APIs. To do so, they need to first construct a whole-app control flow graph and then traverse the entire graph to find suspicious or vulnerable paths that are reachable from entry points. In the course of such analysis, the major challenge is how to accurately construct a whole-app control flow graph because there are many implicit flows in Android apps. Both RiskRanker and Woodpecker used predefined domain knowledge, e.g., connecting `start()` and `run()` methods for a thread, to partially handle some implicit flows.

Dataflow analysis. Most prior works, on the other hand, employed dataflow analysis to taint propagation flows of entry point values or sink API parameters. They have been applied mainly to malware analysis (e.g., [108, 121, 136, 150]), privacy leakage detection (e.g., [105, 111, 112]), and vulnerability discovery (e.g., [83,115,143,144,147,156]). Among them, CHEX [106], FlowDroid [69], and Amandroid [132] are the three representative works. In particular, FlowDroid and Amandroid have been used or customized in many follow-up static analysis tools (e.g., [149] [71] [98] [93] [124] [95]). Compared with RiskRanker and Woodpecker mentioned earlier, these three works tried to systematically handle Android implicit flows by employing lifecycle modeling and object type analysis.

One common thing between reachability analysis and dataflow analysis is that they both require to generate an app call graph, the precision of which affects the entire analysis accuracy. However, generating a high-precision call graph requires expensive object pointer analysis [132], and this scalability problem motivates us to propose on-the-fly analysis via bytecode search in this dissertation.

Condition-aware analysis. Furthermore, some studies were concerned with conditions that trigger a dangerous flow. They usually employed symbolic execution to perform a condition-aware analysis. Two representative works in this domain are TriggerScope [88] and HSOMiner [116]. Specifically, TriggerScope leveraged symbolic execution to identify and characterize the trigger conditions of malicious

application logic, while HSOMiner utilized efficient feature extraction and proposed lightweight machine learning based methods for a similar analysis.

2.1.3 Handling Difficult Technical Issues

There are several common difficulties to Android static analysis, notably Java reflection, native code, dynamically loaded code, and Webview code. Although there are no silver bullets yet for these challenging issues, some dedicated works were proposed to attempt these problems.

Handling reflection and native code. These two issues are still statically analyzable, because both reflection and native code are directly contained in app binaries. Notably, DroidRA [99] employed app instrumentation to transform reflection code into a non-reflection version so that other static analysis tools can directly use the transformed app for a whole-program analysis. Several recent works [95, 124, 131] further analyzed Android native code by leveraging traditional binary analysis tools like IDA Pro [31] and Angr [125].

Handling dynamic and Webview code. In contrast, dynamically load and Webview code usually require a dynamic method to retrieve those code. For example, StaDynA [152] was a pioneer work to address the problem of dynamic code updates for a more completed security analysis of Android applications. Poeplau et al. [118] further systematically analyzed unsafe and malicious dynamic code loading in Android apps. Besides dynamically loaded code via `DexClassLoader` and `PathClassLoader` APIs, Webview code is loaded only when the corresponding web pages are viewed. As a result, a hybrid analysis with both static and dynamic methods is often adopted in prior systems, such as FileCross [138], BridgeScope [148], and OSV-Hunter [147].

2.2 Crypto and SSL Misconfigurations on Android

In this section, we review the previous works that studied the problem of crypto and SSL/TLS misconfigurations in Android apps.

Crypto API misuse in Android apps. Enck et al. [85] was the first to mention the misuse of cryptography in their comprehensive security study of Android apps. Following this direction, CryptoLint [83] performed the first systematic study of cryptographic misuse in 15,134 Android apps using a static program slicing approach. Out of 11,748 apps successfully analyzed, they found that 88% of them made at least one mistake. This demonstrated the pervasiveness of crypto API misuse in Android apps. To help developers automatically mitigate this serious problem, CDRep [107] proposed a method to automatically repair those misused crypto API calls in app bytecode by first defining patch templates and then replacing those insecure crypto parameters with correct ones. Additionally, as mentioned earlier, Amandroid supported the insecure ECB mode detection since its first release [132].

Both CryptoLint and CDRep used static analysis as their methodology, but there are several major differences between their static analysis and our on-the-fly static analysis in Chapter 3. The most significant one is that our on-the-fly analysis does not need to generate a whole-app call graph for the inter-procedural analysis. In contrast, although CryptoLint intended to extract only crypto-related backward slices, it had to first build a so-called super control flow graph. As a result, CryptoLint failed on 3,379 apps out of the entire 15,134 apps due to timeouts and the lack of memory. This 22.3% failure rate indicates the necessity of launching on-the-fly analysis, especially for modern apps that have larger app sizes. Additionally, CDRep used only intra-procedural analysis and thus cannot repair many insecure parameters that flow across multiple methods. This inaccuracy is also the reason why CDRep required just 20 seconds to patch an app.

Android SSL/TLS misconfiguration. Besides crypto misuse, SSL/TLS misconfiguration is another common class of vulnerabilities in Android apps. Mallo-

Droid [86] and the work conducted by Georgiev et al. [89] were the two pioneer studies on vetting SSL/TLS misconfiguration in mobile apps. To facilitate an automatic and accurate security testing, SMV-Hunter [126] combined both static and dynamic analysis techniques to validate whether a SSL/TLS misconfiguration are actually exploitable or not. Furthermore, Zuo et al. [157] employed a similar approach to check SSL/TLS misconfiguration in hybrid mobile web apps. Compared to these works, our BackDroid supports the static detection of SSL/TLS misconfiguration in a way similar to that in Amandroid.

2.3 Works Related to Our Android Open-Port Study

In this section, we present the prior works that are related to our open-port study, including the general open port research, static analysis techniques specifically related to ours, and crowdsourcing techniques that are also for security research.

Open port research. Traditionally, research on open ports focus on DoS attacks [117] and Internet scanning studies [96, 127]. This has been changed in the mobile era — more specific attacks [103, 139, 145] have been demonstrated on open ports of mobile apps. However, studies specifically focused on mobile open ports are not available until recently in the OPAnalyzer paper [95]. Although it is closely related to our study in Chapter 4, there are a number of significant differences. The foremost difference is 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. As a result, the approaches proposed to solve the problems are very different. For example, there is 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 work does. Furthermore, OPAnalyzer does not show any results for UDP ports and built-in apps.

Relevant Android static analysis. Technically, OPAnalyzer [95] was built

upon Amandroid [132] 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 open-port problem due to limitations, such as no complete parameter representation in SAAF [94], no array handling [155], and no open-port relevant API modeling [73]. Our enhanced version of BackDroid in Chapter 4 address these issues by introducing the backward slicing graph and semantic-aware constant propagation. Besides uncovering open-port parameters, it is also the first static analysis tool able to detect open-port SDKs in Android apps.

Crowdsourcing for security research. With the high popularity of mobile apps, it becomes realistic to leverage the crowd to discover security problems in the wild. By deploying an on-device monitoring app, NetMon, to Google Play for a crowdsourcing study, our work in Chapter 4 is a pioneering study on using crowdsourcing for open-port security research. Other security-oriented crowdsourcing works include Netalyzr [130] for studying middleboxes in cellular networks, FBS-Radar [102] for uncovering fake base stations in the wild, UpDroid [129] for monitoring sensitive API behaviors on non-rooted devices, and Haystack [122] for detecting mobile apps' privacy leakage via on-device app traffic analysis [140].

2.4 Declared SDK Versions and Android APIs

In this section, we review the prior research that also studied declared SDK versions and Android APIs as our SDK-API study in Chapter 5.

Research on Declared SDK versions. There were no systematic studies on declared SDK versions previously, except for some specific studies on `targetSdkVersion` or `minSdkVersion` in different scenarios. Notably, Wu and Chang [138] showed that due to using outdated `targetSdkVersion` versions, many Android browser apps were vulnerable to `file://` vulnerabili-

ties. They further demonstrated more security consequences caused by outdated `targetSdkVersion` versions [139]. Following this line of research, Mutchler et al. [110] conducted a large-scale measurement of multiple vulnerabilities affected by fragmented `targetSdkVersion` versions. Wei et al. [134] also studied Android fragmentation with the focus on compatibility issues. In particular, the published paper version [143] of our Chapter 5 has triggered two recent follow-up works [100] [93] on detecting compatibility issues caused by inappropriate `minSdkVersion` versions. Compared to all these works, our study in Chapter 5 is the first systematic work to measure all kinds of `DSDK` versions and their inconsistency with API calls.

Android API studies. Besides `DSDK` and fragmentation, our work in Chapter 5 is also related to prior studies on Android APIs or SDKs. Among these studies, the work performed by McDonnell et al. [109] is the closest to our study. They also studied the Android API evolution, but their focus was how client apps follow Android API changes whereas we focus on the consistency between apps' `DSDK` and API calls. Other related works have studied the relationship between apps' API change and their success [104], the deprecated API usage in Java-based systems [76], the inaccessible APIs in Android framework and their usage in third-party apps [101]; and the Android Alarm API usage and their impacts to network latency [66]. In particular, the work performed by Almeida et al. [66] further analyzed the `targetSdkVersion` in apps that invoke Alarm APIs. Additionally, several security papers analyzed the mappings between Android APIs and their permissions [87] [70] [135].

Chapter 3

On-the-fly Android Static Analysis: The Core Technique and Its Evaluation Study in Detecting Crypto and SSL Misconfigurations

Both Amandroid and FlowDroid were initially proposed in 2014. Although they are still improving over these years, they did not consider handling large apps as a design objective. However, as we will show below, apps have expanded their sizes dramatically over the last five years from 2014 to 2018. To be able to successfully handle these modern apps, we are thus motivated to propose on-the-fly static analysis in this dissertation. This chapter presents its core technique in tool called BackDroid and its evaluation study in detecting crypto and SSL Misconfigurations.

To measure the changes in the app sizes, we first obtain a set of popular apps. Specifically, we collected a set of 22,687 Google Play apps on 11 November 2018 by correlating the AndroZoo repository [65] with the top app lists available on `https://www.androidrank.org`. Each app in this set has at least one million installs on Google Play. We then record the app sizes and DEX file dates (if any) in our dataset.

Table 3.1: A summary of average app size from 2014 to 2018.

Year	Average Size	Median Size	# Samples
2014	13.8MB	8.4MB	2,840
2015	18.8MB	12.4MB	1,375
2016	21.6MB	16.2MB	3,510
2017	32.9MB	30.0MB	1,706
2018	42.6MB	38.0MB	3,178

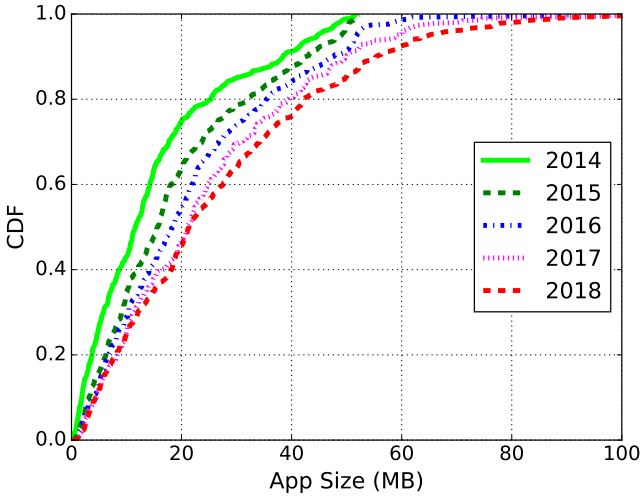


Figure 3.1: CDF plot of size change for the same set of 503 apps.

Table 3.1 summaries the average app size from 2014 to 2018. We can see that in 2014, the average and median app size is only 13.8MB and 8.4MB, respectively. This number almost doubles in 2016, with an average size of 21.6MB and a median size of 16.2MB. It further doubles after two years, with an average app size of 42.6MB in 2018. This clearly shows that modern apps have dramatically expanded their app sizes over the last five years.

To have a more fair comparison with the same sample set, we further select a fixed set of 503 popular apps that have different versions in all last five years. Figure 3.1 presents a CDF (cumulative distribution function) plot of their size change. It is very clear that popular apps constantly expand their sizes every year. For medium apps, the app size almost doubles from 2014 to 2018, while that for large apps, the increase is even more significant.

The remainder of this chapter is organized as follows. We first give an overview of BackDroid in Section 3.1. We then present BackDroid's novel on-the-fly backward search technique in Section 3.2, followed by its implementation in Section 3.3.

In Section 3.4, we evaluate BackDroid and compare it with Amandroid. Finally, Section 3.5 discusses some limitations and Section 3.6 concludes this chapter.

3.1 Overview and Challenges

Given the upscaling trend of app sizes, we design the first on-the-fly static analysis for targeted security vetting of Android apps. Figure 3.2 presents a high-level overview of our novel tool, BackDroid. It works in the following four major steps:

1. Given an input of any Android app(s), BackDroid first extracts original bytecode and manifest files. After that, BackDroid not only transforms bytecode into a suitable intermediate representation (IR) as in typical Android analysis tools, but also employs `dexdump` [23] to dump (merged, if multidex [17] is used) bytecode to a bytecode plaintext.

2. With the dumped bytecode text, BackDroid immediately locates the targeted sink API calls by performing a text search of bytecode and initiates the analysis from there. To further enable inter-procedural analysis with no call graph, BackDroid performs novel (backward) bytecode search to identify caller methods on the fly or just in time.

3. Alongside the inter-procedural analysis enabled by bytecode search, BackDroid performs backward taint analysis to trace sink parameters and their dataflow. To construct a complete representation of such dataflow, BackDroid generates a backward slicing graph (BSG), instead of individual slices, for each sink API call analyzed.

4. On top of the generated BSGs, BackDroid further launches forward analysis to propagate dataflow facts from entry points to sink APIs and to output final sink parameter values (or expression representation if not a constant value). It can also remove potential ambiguity during this process via state-of-the-art points-to analysis.

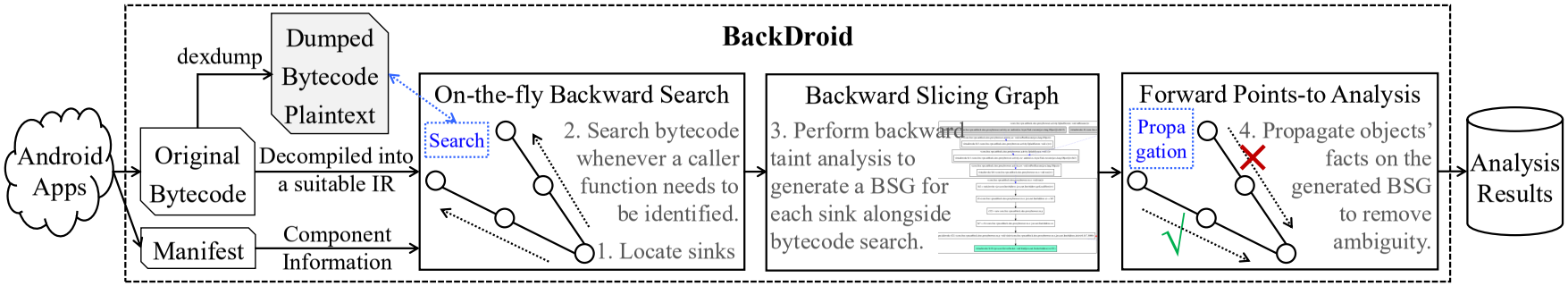


Figure 3.2: A high-level overview of BackDroid.

Challenges. Given that BackDroid is the first inter-procedural dataflow analysis tool without relying on a whole-app call graph, its major novelty and biggest challenge is how to perform the on-the-fly backward search to locate caller methods. This is very difficult because of Java polymorphism (e.g., parent classes and interfaces), threads, Android system callbacks, implicit flows, and inter-component communication, all of which make a basic method signature based search infeasible. We will present our novel bytecode search technique in Section 3.2. One more challenge is how to perform backward taint analysis and forward points-to analysis with a new structure of backward slicing graph. We will present relevant challenges and our corresponding solutions in Section 3.3.

3.2 On-the-fly Backward Search

In this section, we present our backward bytecode search technique to locate caller methods on the fly, which is the key to enable BackDroid's inter-procedural analysis. We first present basic method signature based search in Section 3.2.1, and then elaborate our advanced search with instant forward analysis in Section 3.2.2. Lastly in Section 3.2.3, we explain how BackDroid searches over inter-component communication (ICC), a fundamental cross-app collaboration mechanism on Android [141].

3.2.1 Basic Search by Constructing Appropriate Search Signature

To better illustrate our search process, we use a real popular app, LG TV Plus, which has over 10 million installs on Google Play[1], as a running example. As depicted in Figure 3.3, we have used initial bytecode search to find a target method (the one with a sink API call), `<com.connectsdk.`

[1] `https://play.google.com/store/apps/details?id=com.lge.app1`

service.netcast.NetcastHttpServer: void start()>. For inter-procedural analysis, our next step is to uncover its caller method (i.e., <com.connectsdk.service.NetcastTVService$1: void run()>) and its call site (i.e., statement virtualinvoke $r13.<com.connectsdk.service.netcast.NetcastHttpServer: void start()>()). Since the target callee method here is a regular method (as we will explain soon), searching its caller can be done directly with the following method signature based search.

The basic signature-based search. As illustrated in Figure 3.3, there are five steps to perform the method signature based search. Given a callee method, we first translate its method signature from Soot's [63] IR format to dexdump's bytecode format. With the transformed method signature, we can search the entire bytecode text to locate its invocation(s), as highlighted in the bottom of Figure 3.3. In the second step, we identify the corresponding method that contains the invocation found in the bytecode text. Here it is com.connectsdk.service.NetcastTVService.$1.run:()V, where an inner class needs to add back the symbol "$". With this caller method signature (in bytecode format), we perform another format translation in the third step, and locate its method body via Soot. Next, we conduct a quick forward analysis via Soot to find the actual call site in the caller method body. With all these steps done, we finally connect an edge from the caller (site) to the callee method in BSG (backward slicing graph).

An important question we have not answered is: which kinds of (callee) methods are suitable for method signature based search. We call such methods *signature methods*. Typical signature methods include static methods (either class or method is marked with static keyword), private methods (similarly, methods declared with private keyword), and constructors (e.g., <init> methods of a class). For some searches over child classes, we can also simply launch signature-based search, as explained below.

Searching over child class. Suppose that the NetcastHttpServer class in Figure 3.3 has a child class called ChildServer, we can still use

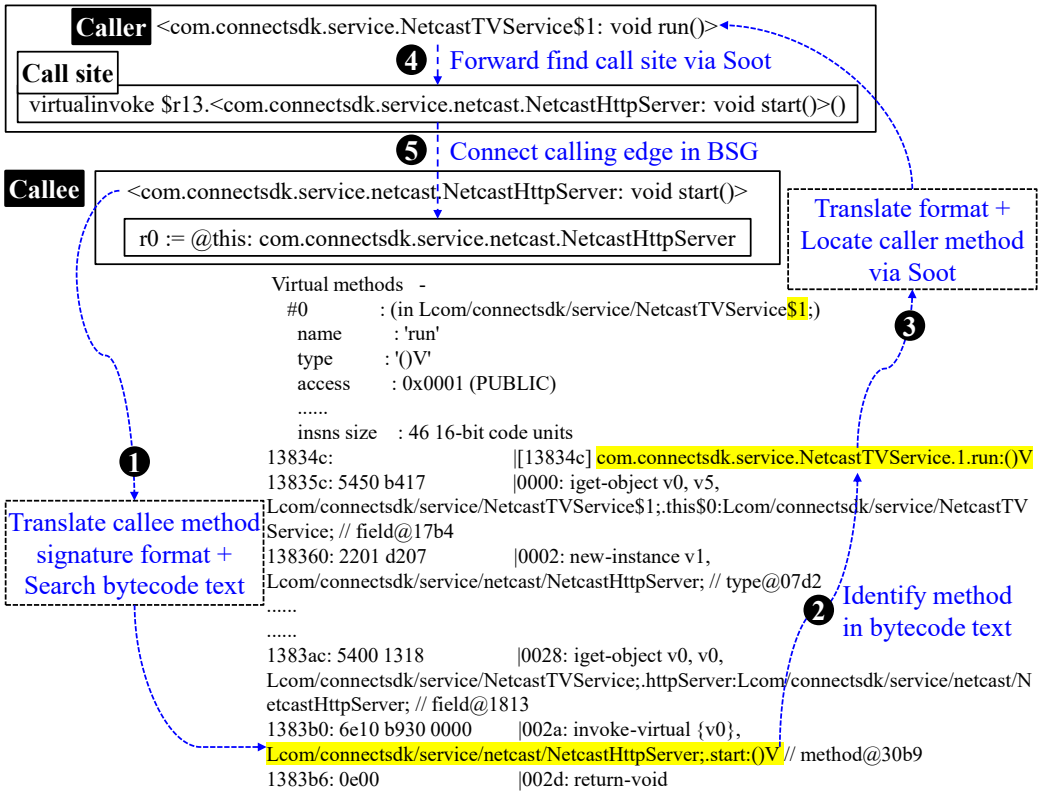


Figure 3.3: Illustrating BackDroid's basic bytecode search process using a method signature based search example.

method signature based search but need to construct appropriate search signatures. We handle it according to whether `ChildServer` overloads the callee method `void start()` or not. If it is not overloaded, an invocation of the callee method `start()` may also come from a child class object. Hence, besides the original signature search, we need to add one more signature search with the child class, namely `Lcom/connectsdk/service/netcast/ChildServer;.start:()V`. The returned caller(s) might be from both searches, or just one of them, depending on how app developers invoke that particular callee method. On the other hand, if `ChildServer` does overload the `start()` method, we can still perform only one search with the original callee method signature. This is because the child class search signature now corresponds to the overloaded child method only.

3.2.2 Advanced Search with Instant Forward Analysis

Although the basic search presented in the last section can handle many callee methods in an app bytecode, it is not effective to address complex situations with super classes, interfaces, and implicit Java/Android flows. We first explain the difficulty of searching in the case where `NetcastHttpServer.start()` in Figure 3.4 has a super class method called `SuperServer.start()`. Under this condition, the original signature search may not reveal any valid callers, which is because developers may write code in this way: `SuperServer server = new NetcastHttpServer(); server.start();`. In this example, the bytecode signature of `server.start()` is `Lcom/connectsdk/service/netcast/SuperServer;.start:()V`. As a result, searching with `NetcastHttpServer`'s method signature would hint nothing. We also cannot use super class `SuperServer`'s signature to launch the search, because it could return callers of the super method itself and other class methods that inherit from `SuperServer`. Second, if a callee method implements an interface, searching using the interface method signature would not work because an interface method might be implemented by arbitrary classes. Finally, searching over implicit Java/Android flows could be even more difficult, because they employ different sub-method signatures for a pair of caller and callee methods.

We design a novel mechanism to accurately handle all these complex searches. The basic idea is that instead of directly searching caller methods, we first search the callee class' object constructor(s) that can be accurately located. Then starting from those object constructors, we perform instant forward propagation until we detect caller methods. We depict this process in Figure 3.4, using the same LG TV Plus app. This time the callee method is `<com.connectsdk.service.NetcastTVService$1: void run()>`, which continues the search flow in Figure 3.3. We now present the four major steps involved, as shown in Figure 3.4.

Searching for the object constructor. After determining a callee method that

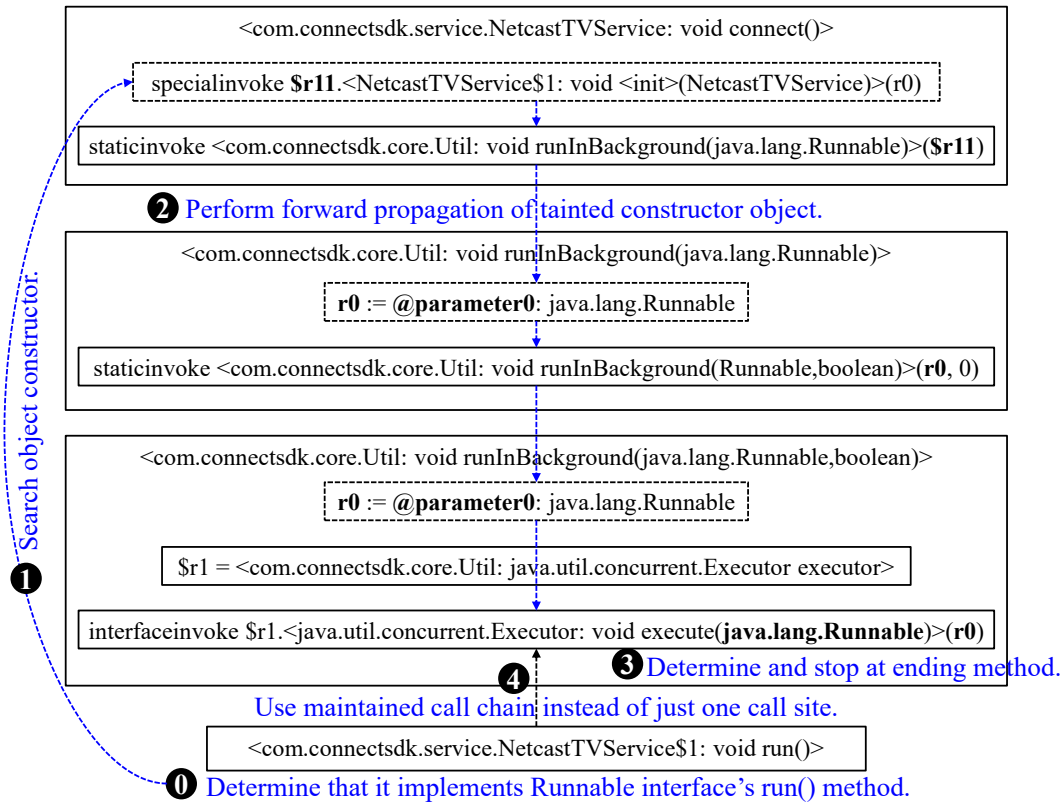


Figure 3.4: Using advanced search with instant forward analysis to recover a caller chain of an interface method, `NetcastTVService$1.run()`. Note that statement blocks with square dots are not shown in this app's backward slicing graph.

requires advanced search, we first retrieve all its constructors. In Figure 3.4, the callee class `NetcastTVService$1` has only one constructor, `void <init>(com.connectsdk.service.NetcastTVService)`. We then launch a bytecode search using this method signature to locate that the constructor is initialized in a method called `NetcastTVService: void connect()`, the process of which is similar to that in Section 3.2.1.

Propagating object using taint analysis. In the second step, we perform forward propagation of the located constructor object, i.e., `$r11` in Figure 3.4, using taint analysis. Specifically, an object can be propagated via a definition statement, e.g., `r0 := @parameter0: java.lang.Runnable`, via an invoke statement, e.g., `runInBackground($r11)`, or via a return statement. Therefore, we track only three kinds of statements, namely `DefinitionStmt` [20], `InvokeStmt` [34], and `ReturnStmt` [48].

Determining the ending method to stop. An important step is to determine at which *ending method* our forward propagation should stop. This is easy for the case of super class, because we can simply stop at a tainted statement with the same sub-method signature as the callee method. However, it is difficult for the cases of interface and implicit flow, because their sub-method signature might be different from that in a callee method. Some previous works (e.g., [91, 144, 150]) used pre-defined domain knowledge to connect those implicit flows, e.g., a common example is to connect `Thread` class' `start()` and `run()` methods. However, as shown in Figure 3.4, it will miss the ending method `Executor.execute()`.

To better determine the ending method, we propose a mechanism that does not rely on prior knowledge but leverages interface's class type as an indicator. For example, in Figure 3.4, since the interface class type is `java.lang.Runnable`, we thus determine which Java/Android API call contains a tainted parameter that satisfies this class type.

Maintaining and returning a call chain. Different from the basic search that returns just one call site, here we need to maintain and return a call chain, i.e., a chain from `NetcastTVService.connect()` to `Util.runInBackground(Runnable)`, and further to `Util.runInBackground(Runnable,boolean)`. Assuming that we only return a call site and one caller method, we would still launch backward search of `Util.runInBackground(Runnable,boolean)` and it may have multiple search results or flows. However, only the flow shown in Figure 3.4 could eventually trace back to the constructor object. Therefore, to avoid mis-added flows, we need to maintain a call chain during the forward taint analysis.

3.2.3 Searching over Android ICC

Although the basic and advanced searches in the last two sections are useful in most scenarios, they are not designed to handle Android inter-component communication

(ICC). In this section, we present our special search mechanism for tracking data flows over Android ICC.

Our search is based on the inner working mechanism of Android ICC. Specifically, ICC is different from typical API call because it relies on its `Intent` parameter values to determine a target callee. A callee could be *explicitly* specified by setting the target component class (e.g., via `Intent i = new Intent(activity,HttpServerService.class);`), or *implicitly* specified by setting an `Intent` action that will be delivered by the OS to the target component.

Based on this observation, we propose a two-time search mechanism to handle ICC. The basic idea is to launch two searches: one is for searching ICC calls (e.g., `startService()`), and the other is to search ICC parameters. For explicit ICC, the second parameter search directly searches component class names, e.g., `const-class .*, Lcom/lge/app1/fota/HttpServerService;`. For implicit ICC, we search `Intent` action names instead. After that, we merge the two search results and check whether an ICC call satisfies both searches. If there is such an ICC call, it is the caller method we are looking for.

3.3 Implementation

In this section, we present the major technical challenges in implementing BackDroid and our solutions. We start with some implementation enhancements to the on-the-fly backward search presented in the last section. After that, we summarize the challenges in generating backward slicing graph (BSG) and performing forward points-to analysis over BSG.

3.3.1 Enhancements to Backward Search

In the course of implementing on-the-fly backward search in Section 3.2, we identify and make several important technical enhancements to guarantee the performance of backward search.

Search caching. The first enhancement is to cache different search commands and their corresponding results. This is necessary because in a valid app analysis, BackDroid will make a number of searches and a portion of them could be executed repeatedly (especially when similar paths are explored across different sinks). Caching can avoid repeating the same searches. We perform caching with different granularities, including the caching of invoked class search, caller method search, static or instance field search, and the caching of various raw search commands.

On-the-fly dead library elimination. Different from Amandroid that skips the analysis of many third-party libraries[2], BackDroid considers all embedded libraries as long as they contain sink APIs. However, blindly triggering the analysis of any sink-containing library could lead to unnecessary or wasted analysis, because at a sink point, we cannot determine whether the library would be eventually invoked by the main app code, i.e., dead library code or not. To avoid dataflow analysis of dead libraries, we propose a lightweight and yet effective mechanism called *on-the-fly dead library elimination*. For each sink class, we leverage this mechanism to determine whether or not to start the actual dataflow analysis.

We detect and eliminate dead libraries in several steps. First, we extract the root class name for a given sink class, e.g., "`com.connectsdk`" for class `com.connectsdk.service.RokuService`. A root name is determined according to the class hierarchy: it is the top-level package name that contains direct Java classes, e.g., several raw class files under package `com.connectsdk`. After that, we match the extracted root class name with app components' core class names that are extracted from manifest in the preprocessing (see Section 3.1). As

[2]Amandroid defines a `liblist.txt` file that contains 139 packages (e.g., "`cn.immob.*`" and "`com.facebook.*`") to skip the analysis of these libraries by default.

soon as one overlap (e.g., "com.lge" overlapping with "com.lge.app1") is identified, we stop further estimation and trigger the dataflow analysis immediately. Otherwise, we continue to check which app classes invoke the sink class by performing an on-the-fly class search of the extracted root class name. If one of the searched classes matches with manifest names, we consider that the sink class is valid. For example, the class search of "com.connectsdk" returns `Lcom/lge/app1/activity/MainActivity$10;` class, which matches with component names in `com.lge.app1` app's manifest. Otherwise, it is from a dead library, e.g., the "`org.apache.log4j`" library in the `com.lge.app1` app.

3.3.2 Generating the Backward Slicing Graph

During the inter-procedural analysis enabled by bytecode search, we perform (backward) taint analysis and generate a backward slicing graph (BSG) for each sink API call analyzed. We have addressed three major challenges in the course of our implementation.

Defining a self-contained graph structure to cover all slicing information. The first is to define a structure that can cover all slicing information across different parameters tracked, different paths traced, and all kinds of bytecode instructions. Instead of generating individual path-like slices as in typical Android slicing tools (e.g., [73, 94, 155]), we propose a self-contained graph structure called *backward slicing graph* (BSG) to cover all slicing information. In this dissertation, one BSG corresponds to one unique sink API call, and we may also extend such per-sink BSG to per-app BSG in the future. Figure 3.5 shows an example BSG that is automatically generated by BackDroid for the app package `com.proxybrowser.vpn.unblock.sites.browser`. Compared with traditional slides, our BSG contains the following additional slicing information within its structure:

- *Hierarchical taint map.* Although not displayed in Figure 3.5, a hierarchical taint map is actually maintained during our inter-procedural backtracking. Specifically, our BSG assigns a taint set to each tracked method and organizes all sets hierarchically according to their method signatures. For static fields, we also maintain a global taint set. With this hierarchical taint map, BackDroid's taint analysis module can easily retrieve the current taint set from BSG whenever its tracking jumps in or out from any (caller or inner) method, and can also track multiple sink parameters simultaneously.

- *Inter-procedural relationships.* To differentiate different taint paths without using individual slices, we maintain inter-procedural relationships via different kinds of cross-method edges in BSG. The most common one is the edge connecting a caller method, e.g., the edge from caller `a.w.onPostExecute()` to `m.o.run()` in Figure 3.5. It is also possible for a tracked method to invoke its inner method (e.g., method `m.p.<init>()` in Figure 3.5), and we use both calling and return edges to record this special inter-procedural relationship.

- *Raw typed bytecode statements.* Lastly, to enable BackDroid to recover full semantics during the forward analysis, it is necessary to keep raw typed bytecode instructions in BSG. We thus define a node structure called `BSGUnit` to wrap the original bytecode statements in Soot's `Unit` format [61]. In this structure, we record the node ID, the signature of corresponding method, and most importantly, the typed bytecode `Unit` statement.

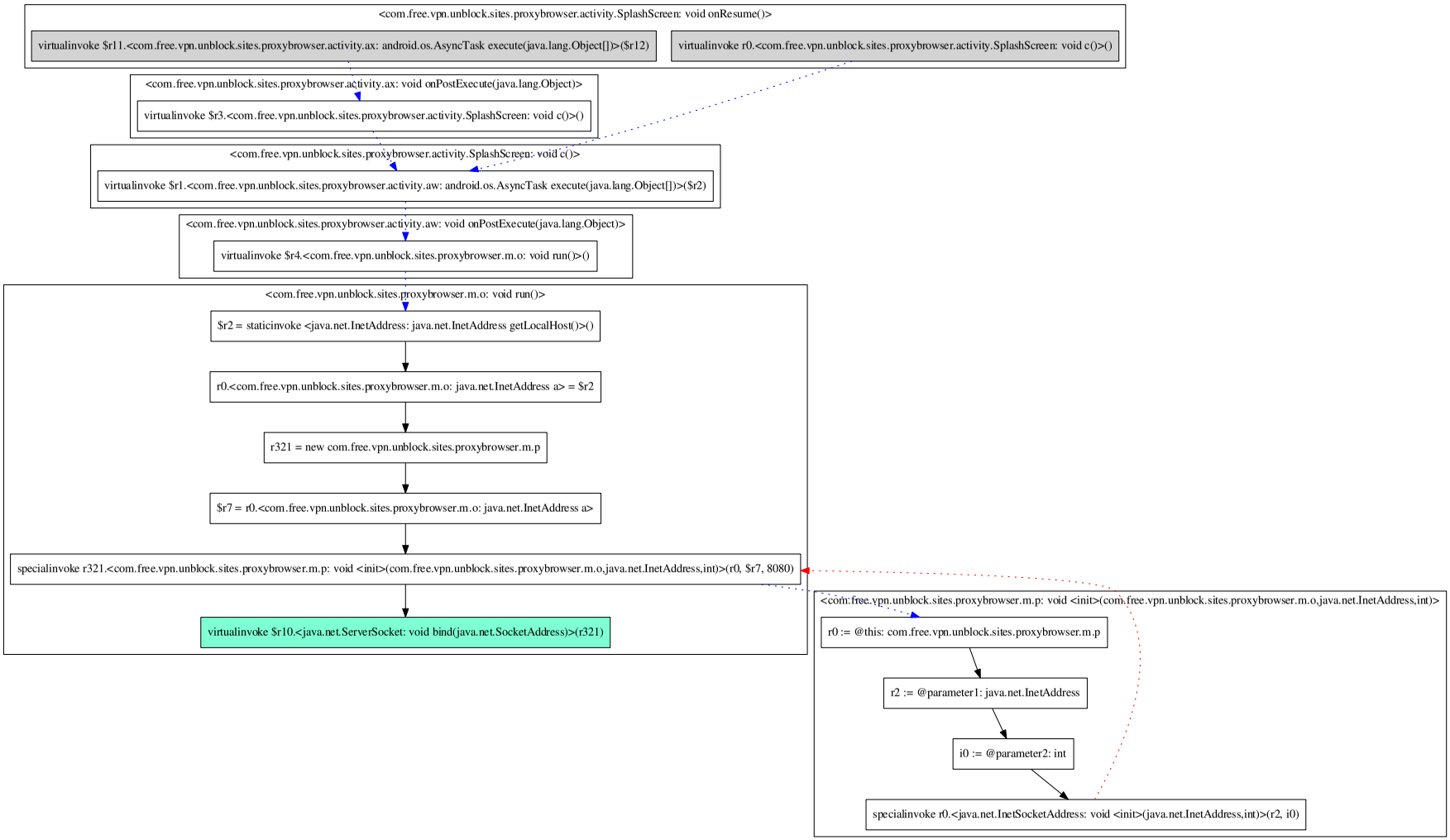


Figure 3.5: A BSG automatically generated by BackDroid, where the green block is sink API call and gray blocks are entry points.

Tainting across fields, arrays, and inner methods. With the BSG structure defined, our next challenge is to perform precise and efficient *backward* taint analysis for the BSG generation. Compared to the forward taint analysis in Amandroid and FlowDroid, our taint analysis is more difficult because it reverses normal program execution and thus has no insights into the earlier execution of tainted variables. In particular, we have the following special taint process for fields, arrays, and inner methods. First, for an instance field to be tainted, we add not only the instance field itself (i.e., `obj.field`) to the taint set but also its class object (i.e., `obj`) so that we can trace the same field no matter the class object gets aliased or across method boundaries. Moreover, when the instance field needs to be untainted, we first remove `obj.field` from the taint set and further detect whether there are more fields for the same instance. If there are no other such fields, then we remove `obj` from the taint set as well. Arrays are handled in a similar way.

One more special tainting is to handle inner methods when there are static fields in the taint set. In this scenario, a normal processing is to jump into all inner methods (even when their parameters are not tainted) and analyze them, because we cannot determine whether an inner method uses a tainted static field or not. Analyzing all inner methods on the backtracking paths certainly slows down the analysis, and we have proposed a more elegant solution. Specifically, whenever a new static field is tainted, we launch bytecode search of this field signature to capture all methods that invoke this particular static field. Hence, we only need to analyze the inner methods that are matched with search results.

Adding static initializers into BSG on demand. Analyzing static fields in a whole-app analysis fashion is expensive, because static initializers of all invoked classes (i.e., not only those app component classes) and all statements contained in those initializers need to be analyzed. As a result, Amandroid by default does not analyze static initializers via the configuration "static_init = false", and FlowDroid also provides the option "–nostatic" for its users to reduce the running time for large

apps[3].

Since BackDroid performs targeted analysis via bytecode search, we can fully track all tainted static fields. Specifically, after the main taint process is done, if there are still unresolved static fields in the BSG's taint map, we retrieve their corresponding classes and obtain the `<clinit>` methods (which are only implicitly executed by the Java/Android virtual machine (VM) when the corresponding classes are loaded to the VM). We then perform backward taint analysis of these `<clinit>` methods, and add only relevant statements into a special track of BSG. During the forward analysis, we first analyze this special track and then handle the main track of BSG.

3.3.3 Forward Points-to Analysis over BSG

After producing a complete BSG, our forward analysis iterates through each BSG node, analyzes each statement's semantic, and propagates dataflow facts along the graph traversal. The main challenge is how to perform *points-to* analysis [97] over our new BSG representation. Below we explain three major steps in our forward points-to analysis, namely to *traverse* over BSG, to *analyze* statement semantics, and to *propagate* points-to information.

Overall traversal process over BSG. As mentioned at the end of Section 3.3.3, a BSG includes two tracks, the special static field track and the normal track. Our traversal always starts with analyzing the static field track so that we can resolve fields referred in the normal track. In the course of analyzing each track, we first retrieve a set of tail nodes (e.g., two entry points in Figure 3.5) and initialize analysis from each of them. To record facts generated by our analysis, we maintain fact maps for each analysis flow, but we use only one global fact map for analyzing all static fields.

Whenever we reach at a new BSG node, we perform graph traversal in several

[3] `https://github.com/secure-software-engineering/FlowDroid/issues/27`

steps. First, we determine whether the node is an initial BSG node with a sink API call; and if it is, we correlate and output dataflow facts of all tainted parameters. For a normal BSG node, we first jump into and analyze its inner methods if any. After that, we analyze the current node itself and move to the next node(s).

Analyzing and modeling statement semantics. During the traversal of each BSG node, we parse its bytecode statement and analyze the semantics. There are three kinds of statements to handle, namely `DefinitionStmt` [20] (and its subclass `AssignStmt` [13]), `InvokeStmt` [34], and `ReturnStmt` [48]. After determining the type of the statement, we further extract six kinds of statement expressions, including `CastExpr`, `InvokeExpr`, `BinopExpr`, `InstanceFieldRef`, `ArrayRef`, and `PhiExpr`. We then follow these expression instructions to understand their semantics.

Two special expressions, `BinopExpr` [14] and `InvokeExpr` [33], require dedicated modeling. For the `BinopExpr` expression, we extract its two operands and generate a corresponding Java code statement to mimic the semantics of six major arithmetic operators, +, -, * , /, %, and ^. We further model Android or Java APIs to handle `InvokeExpr`. We provide interfaces to support different kinds of APIs, and currently BackDroid has modelled Android `Intent` APIs (e.g., `Bundle.putInt(String,int)`), mathematical APIs (e.g., `Math.random()` and `Math.abs(int)`), Java String or Integer APIs (e.g., `String.charAt(int)` and `Integer.parseInt(String)`), IP address APIs (e.g., `InetAddress.getByAddress(byte[])`), and configuration APIs (e.g., `SharedPreferences.getInt(String,int)`).

Propagating constant and points-to information. To enable dataflow propagation, we maintain a fact map to correlate each variable and its dataflow fact. Propagating constant facts among different variables is easy — just retrieve the value from an old variable and assign it to a new variable in the fact map. To propagate points-to information, we design an object structure to preserve original points-to information along flow paths.

We refer such object structure to as `InstanceObj` and initialize a unique `InstanceObj` object for each `new` statement. Each `InstanceObj` object contains a pointer to its creator class and a map of member objects (in any class type) and their reference names. As a result, `InstanceObj` can be used to save very complicated points-to information, e.g., one `InstanceObj` embedding another inner `InstanceObj`. To propagate points-to information, we just need to propagate `InstanceObj` objects along flow paths so that all corresponding objects being traced can point to the same `InstanceObj` object. Inner members of `InstanceObj` can also be updated by checking classes' `<init>` functions or any other value-assignment statements. Besides the class objects' points-to information, we further define an `ArrayObj` object to wrap points-to information of any array expression (i.e., `NewArrayExpr` [42]) and its array map between indexes and values.

3.4 Evaluation

In this section, we evaluate the efficiency and efficacy of BackDroid in analyzing modern apps. In particular, we compare BackDroid with Amandroid [132, 133], the state-of-the-art Android static dataflow analysis tool. Note that we do not choose FlowDroid [69] for comparison because its SPARK-based call graph generation is not context-sensitive [55]. Moreover, FlowDroid by default does not track ICC flows, and even after the integration of IccTA [98], it is still less accurate than Amandroid's ICC tracking [133].

3.4.1 Experimental Setup

To perform an evaluation study for both BackDroid and Amandroid, we select two known yet serious vulnerability patterns that are supported in both tools, namely crypto and SSL/TLS misconfigurations. In both cases, the root cause is due to insecure parameters. For example, the ECB mode is used to create the `javax.`

`crypto.Cipher` instance [83, 107] and the insecure parameter `ALLOW_ALL_HOSTNAME_VERIFIER` is used in `setHostnameVerifier()` [86]. BackDroid by default supports dataflow analysis of all sink-based API misuse, while Amandroid also tested these two vulnerabilities as recently reported in [133]. This makes a fair comparison between Amandroid and our BackDroid possible. In the following paragraphs, we describe the dataset tested, computing environment used, and tool parameters configured.

Dataset. We use a set of modern popular apps that satisfy two conditions: (i) have at least one million installs each, and (ii) were updated recently in 2018. Specifically, we first select all such 3,178 apps in our app repository (see Table 3.1). However, since not all of them contain the specific sink APIs, we preprocess them to search apps with all three selected sink APIs, namely `Cipher.getInstance()`, `SSLSocketFactory.setHostnameVerifier()`, and `HttpsURLConnection.setHostnameVerifier()`. This can help Amandroid avoid wasting the analysis, because it has no bytecode search as in BackDroid. As a result, we use the searched 144 apps for our experiments. The average and median app size in this dataset are 41.5MB and 36.2MB, respectively. The smallest app size is 2.9MB while the largest is 104.9MB.

Environment. For the computing environment, we use a desktop PC with Intel i7-4790 CPU (3.6GHZ, eight cores) and 16GB of physical memory. Note that a memory configuration with 16GB or less is often used in many academic Android app analysis works, e.g., [116,124,149,151]. To guarantee sufficient memory for the OS itself, we assign 12GB RAM to the Java VM heap space in running Amandroid. Since BackDroid is not sensitive to memory, we use only 4GB (i.e., `-Xmx4g`). The OS is 64-bit Ubuntu 16.04, and we use Java 1.8 and Python 2.7 to run the experiments. Additionally, BackDroid employs the latest dex2jar (version 2.1-nightly-28) to convert Android bytecode to Java bytecode.

Tool configuration. While our BackDroid can always run full-capability analysis, both Amandroid and FlowDroid need to configure a set of parameters to balance

their performance and precision. In this chapter, we use the default Amandroid parameters as follows:

- `k_context = 1`: context length for k-context sensitive analysis;
- `static_init = false`: handle static initializer or not;
- `timeout = 2`: timeout setting for analyzing one component (minutes);
- `third_party_lib_file = /liblist.txt`: a third-party library file that lists 139 Java packages to skip.

In particular, we use the latest Amandroid 2.0.5 that supports inter-procedural API misuse analysis[4]. We give Amandroid sufficient running time with a timeout of 100 minutes for each app, while BackDroid's configuration is only 20 minutes per app.

3.4.2 Performance Results

Out of the 144 apps analyzed, BackDroid successfully finished the analysis of 139 apps while that for Amandroid was 141 apps. The failures are mainly due to manifest errors in those APK files (BackDroid introduced two more errors because of dex2jar's failures). Figure 3.6 and Figure 3.7 show the distribution of analysis time used by BackDroid and Amandroid, respectively. By correlating these two figures, we make the following three observations on the performance of BackDroid and Amandroid.

First, *BackDroid's analysis time is always under control, with significant fewer timeouts as compared to Amandroid.* Even though we set a much higher timeout for Amandroid (five times more than that in BackDroid), there are still as many as 69 app timeouts in Amandroid, as shown in Figure 3.7. In other words, almost half of all the 141 apps analyzed by Amandroid were time-outed. In contrast, Figure 3.6

[4]Amandroid after version 2.0.5 uses only *intra*-procedural dataflow analysis to analyze API misuse, see details at `https://github.com/arguslab/Argus-SAF/issues/55`.

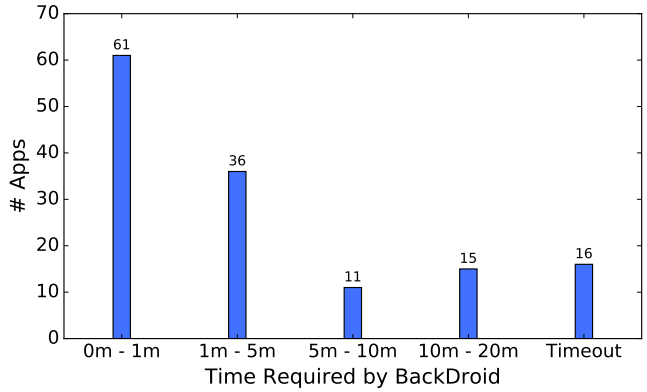


Figure 3.6: The distribution of analysis time in BackDroid.

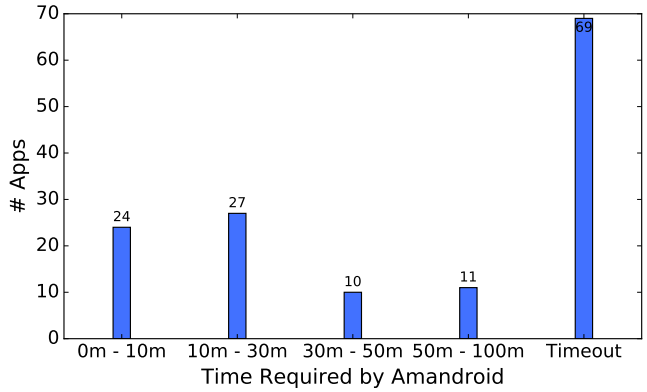


Figure 3.7: The distribution of analysis time in Amandroid.

shows that only 16 apps reached the 20min timeout in BackDroid. Moreover, after further analyzing these timeouts, we find that they were mainly caused by a dead search loop failure in the current BackDroid implementation, which can be fixed in our next version.

Second, *BackDroid can quickly finish the analysis of most of the apps, with 75% apps analyzed within 10 minutes*. After analyzing the cases of large analysis time, we now focus on apps with shorter analysis time. According to Figure 3.7, only 24 (16.7%) apps can be analyzed by Amandroid within 10 minutes. In contrast, that percentage is as high as 75% in BackDroid with 108 apps' analysis time shorter than 10 minutes. Particularly, as shown in Figure 3.6, the analysis of 61 apps were quickly finished within just one minute. This gives BackDroid a great potential to be deployed by app markets for online vetting.

Third, *the overall performance of BackDroid is around ten times faster than*

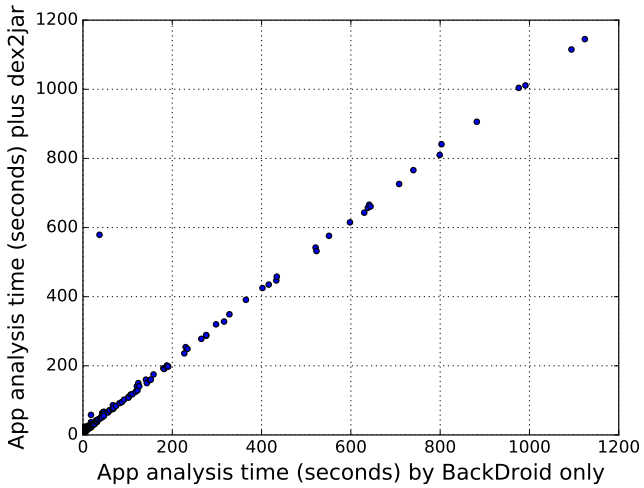


Figure 3.8: Scatter plot of the relationship between BackDroid's analysis time and the entire analysis time plus dex2jar.

that in Amandroid, even after excluding timeouts. After studying apps that take relatively long and short analysis times, we further analyze the overall performance. We find that after excluding time-outed apps for both tools, the average and median analysis time of BackDroid is 200s and 66s, while that in Amandroid is 24.7m and 15.5m, respectively. In other words, BackDroid's overall performance is 7.4 times (for mean) or 14 times (for median) faster than that in Amandroid. Moreover, after including the time-outed apps, the performance gap between the two tools is even more significant: the overall median time of BackDroid is around 50 times faster than that in Amandroid (92s versus 87m).

One performance overhead of BackDroid we have not measured is the additional pre-processing time introduced by dex2jar. Figure 3.8 presents a scatter plot of the relationship between BackDroid's analysis time with and without time spent in dex2jar. It is clear that dex2jar introduces very small additional overhead for all apps except one outlier, the `com.jio.myjio` app. We find that this app requires significant pre-processing time (542s) because of an `OutOfMemoryError` exception in dex2jar.

3.4.3 Detection Results

After comparing BackDroid's and Amandroid's performance, we further analyze and compare their detection accuracy. We present their detection results from the following two perspectives:

Vulnerabilities detected by Amandroid but not BackDroid. We first analyze whether BackDroid could achieve a close detection rate for the app vulnerabilities that are detected by Amandroid. For the crypto API usage, Amandroid detects that five apps are still using insecure ECB mode. We find that BackDroid can accurately detect all of them. The vulnerable apps include the popular Adobe Fill & Sign app (`com.adobe.fas`) and a bank app called IDBI Bank GO Mobile+ (`com.snapwork.IDBI`). Both apps must guarantee a secure encryption in their design.

Compared to crypto API misuse, Amandroid detects more SSL misconfigurations in our dataset, with 20 apps discovered with wrong SSL hostname verification. Among these apps, BackDroid failed on three of them. A further diagnosis shows that all of these failures are caused by a third-party library called `com.skt.arm.ArmSeedCheck`. Specifically, this library uses an AIDL (Android Interface Definition Language) function that Amandroid considers as an entry function, whereas BackDroid does not.

Vulnerabilities detected by BackDroid but not Amandroid. We further find that for some apps, BackDroid can achieve better detection performance than Amandroid. In particular, BackDroid discovered 15 apps with insecure ECB mode that were not detected by Amandroid. Due to timeouts, Amandroid failed to detect eight of these 15 apps. For the other seven apps, an important reason for Amandroid's failures is because it skipped the analysis of some popular libraries that are specified in its `liblist.txt` configuration file. Specifically, among the 14 sink classes in those seven apps, Amandroid ignored six of them, including class names, such as `com.amazon.appexpan.client.util.`

`CipherUtils`, `com.tencent.mm.sdk.platformtools.LogHelper`, and `com.inmobi.commons.internal.InternalSDKUtil`.

On the other hand, for SSL misconfiguration, BackDroid also discovered three vulnerabilities that were missed by Amandroid. All the three corresponding apps were not time-outed, so they are true positives not handled by Amandroid. The root cause of these failures is that Amandroid by default does not process static initializers.

To summarize our analysis above, we give this takeaway regarding the detection accuracy between BackDroid and Amandroid:

> **Takeaway:** BackDroid achieves close detection effectiveness for apps that can be detected by Amandroid, and obtains better detection results for apps with popular libraries and static initializers that are skipped by Amandroid.

3.5 Discussion

So far, we have elaborated our approach in the context of Android bytecode. There are some common technical issues in typical Android app analysis works, namely Java reflection, native code, dynamically loaded code, and packed code. Although addressing these issues is not our focus in this dissertation, we discuss our plan to mitigate them in the future work.

Java reflection. To mitigate Java reflection, an immediate solution is to leverage DroidRA [99] to transform an original app APK to a version without reflection calls. In the long run, we plan to first resolve reflection parameters using our backtracking capability and then build caller edges to directly cache them.

Native code. To extend BackDroid's design principle also to native code, a potential way is to replace `dexdump` with `objdump`. Furthermore, given small size of native code in Android apps and their limited entry points, it is possible to launch full-scale forward analysis, as demonstrated in recent SInspector [124] and

JN-SAF [131] works.

Dynamically loaded and packed code. Static analysis fundamentally suffers from dynamically loaded or packed code. Fortunately, we can leverage earlier dynamic analysis works [137, 152] to first extract those hidden code before running our BackDroid.

3.6 Summary

In this chapter, we presented BackDroid, an on-the-fly static dataflow analysis tool for targeted security vetting of Android apps. Different from existing Android static analysis tools, BackDroid does not generate an expensive whole-app call graph but creatively leverages bytecode search to guide inter-procedural analysis on the fly. Specifically, BackDroid employs a novel on-the-fly backward search technique to search over Java polymorphism, threads, implicit callback flows, and Android inter-component communication. To evaluate BackDroid's efficiency and efficacy, we compared it with the state-of-the-art Amandroid tool in analyzing modern apps for crypto and SSL/TLS misconfigurations. The results showed that BackDroid achieves a much better performance, around ten times faster on average, while maintaining close detection effectiveness as Amandroid. Moreover, BackDroid can detect additional vulnerabilities for apps with some popular libraries and static initializers that are by default skipped by Amandroid.

Chapter 4

Analyzing the Security of Open Ports in Android Applications

4.1 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. [103] demonstrated the insecurity of local TCP open ports used in non-rooted Android screenshot apps. Wu et al. [139] found that the top ten file-sharing apps on Android and iOS typically do not authenticate traffic to their ports. Bai et al. [145] further revealed the insecurity of Apple ZeroConf techniques that are powered by ports such as 5353 for mDNS.

Besides these manual studies on specific apps, Jia et al. [95] 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 [118, 120], complex implicit flows [75, 119], and advanced code obfuscation [82, 137]. 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 chapter, 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 first discover open-port apps using crowdsourcing and then use an enhanced version of BackDroid to identify insecure open ports and open-port SDKs in the discovered open-port apps. Specifically, one enhancement is to supply BackDroid with open-port related semantics, e.g., random port number via `Math.random()` and IP address array like `byte[]{127,0,0,1}`, which are often used by the `ServerSocket` sink API in the open port problem. The other is to add the SDK identification capability into BackDroid. As shown in Figure 4.1, 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.

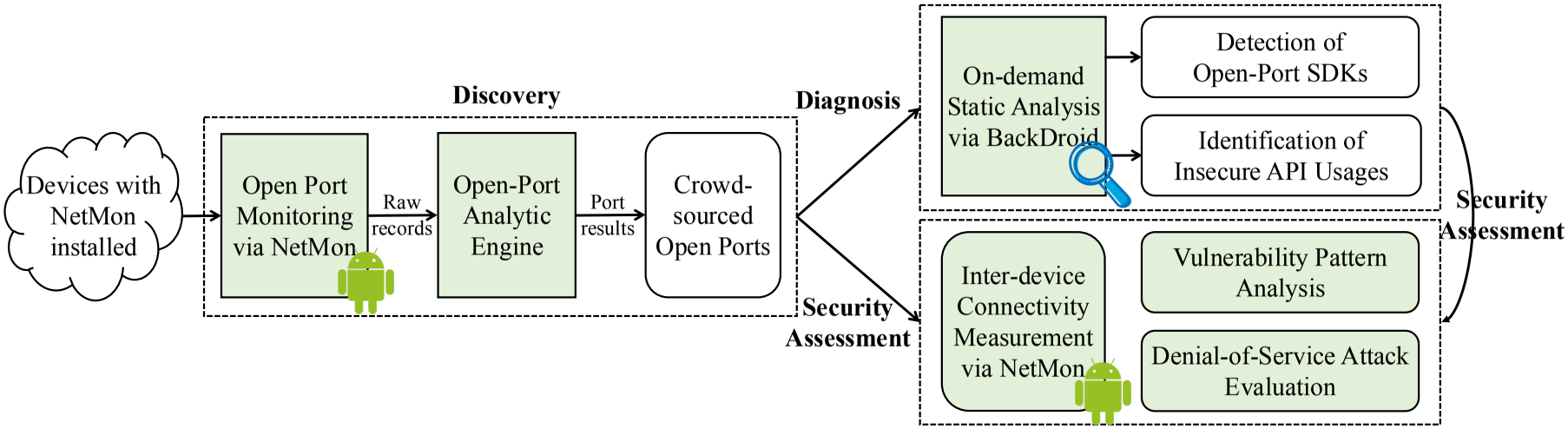


Figure 4.1: The workflow of our open-port analysis pipeline (methodology shown in colored blocks and results shown in rounded blocks).

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 chapter, 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 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 [95]. 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 enhancing BackDroid with open-port related semantics and SDK identification capability, to understand the

[1]NetMon is short for "Network Scanner & Port Monitor" and is available at `https://play.google.com/store/apps/details?id=com.netmon`.

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:

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.

4.2 Background and Threat Model

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

An open port, in this chapter, 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` [15] (in Android SDK), `NFCSocket` [43] (an open-source library), and in particular, the previously studied UNIX domain socket [124] 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 Section 4.5.3.

- 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 [25] 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.

4.3 Discovery via Crowdsourcing

The first phase of our pipeline is to discover open ports. Instead of using static analysis as in [95], 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 [6]; (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 Section 4.3.3, our crowdsourcing is much more effective in terms of port discovery than typical Android static analysis, which cannot handle dynamic code loading [118, 120], complex implicit flows [75, 119], and advanced code obfuscation [82, 137].

Our crowdsourcing platform consists of an on-device port monitoring app NetMon (Section 4.3.1) and a server-side open-port analytic engine (Section 4.3.2). We have deployed NetMon to Google Play and collected the crowdsourcing results from a large number of real users (Section 4.3.3). 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

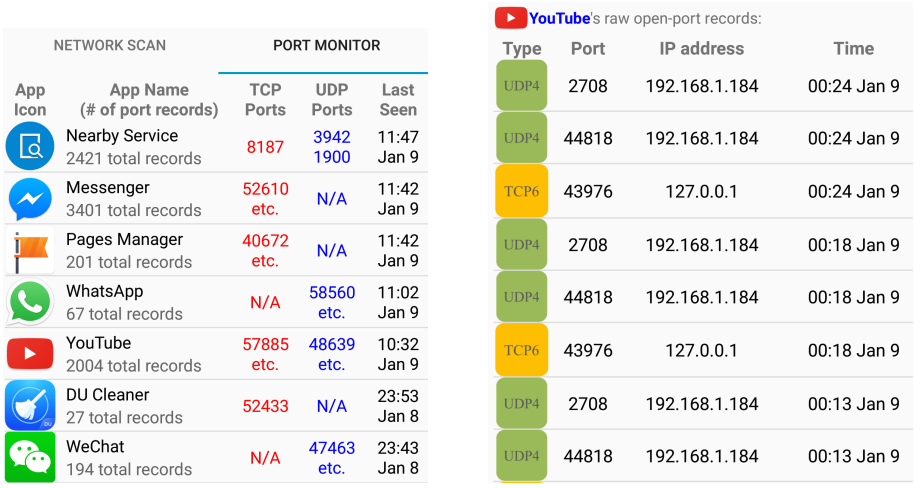
explore on-device crowdsourcing for monitoring other open-port apps in real user devices. Moreover, compared to the typical app-based crowdsourcing (e.g., Netalyzer [130], MopEye [140], and Haystack [122]), our open-port crowdsourcing is unique in that the collected raw records cannot be directly analyzed 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.

4.3.1 On-device Open Port Monitoring

Different from ZMap [96] and Nmap [44] 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. Figure 4.2 shows two NetMon user interfaces for port monitoring. Figure 4.2(a) shows a partial list of apps running with open ports, while Figure 4.2(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.

Port monitoring mechanism. NetMon leverages a public interface in the `proc` file system [47] 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 [56], such as LISTEN and ESTABLISHED.
- *The app UID.* Using the `PackageManager` APIs, NetMon obtains the app's



NETWORK SCAN | PORT MONITOR

App Icon	App Name (# of port records)	TCP Ports	UDP Ports	Last Seen
	Nearby Service 2421 total records	8187	3942 1900	11:47 Jan 9
	Messenger 3401 total records	52610 etc.	N/A	11:42 Jan 9
	Pages Manager 201 total records	40672 etc.	N/A	11:42 Jan 9
	WhatsApp 67 total records	N/A	58560 etc.	11:02 Jan 9
	YouTube 2004 total records	57885 etc.	48639 etc.	10:32 Jan 9
	DU Cleaner 27 total records	52433	N/A	23:53 Jan 8
	WeChat 194 total records	N/A	47463 etc.	23:43 Jan 8

(a) A sample of open-port apps.

YouTube's raw open-port records:

Type	Port	IP address	Time
UDP4	2708	192.168.1.184	00:24 Jan 9
UDP4	44818	192.168.1.184	00:24 Jan 9
TCP6	43976	127.0.0.1	00:24 Jan 9
UDP4	2708	192.168.1.184	00:18 Jan 9
UDP4	44818	192.168.1.184	00:18 Jan 9
TCP6	43976	127.0.0.1	00:18 Jan 9
UDP4	2708	192.168.1.184	00:13 Jan 9
UDP4	44818	192.168.1.184	00:13 Jan 9

(b) Detailed records for YouTube.

Figure 4.2: User interfaces in NetMon showing open ports.

name from its UID (user ID).

According to the definition in Section 4.2, 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 Figure 4.2(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 [3] 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 [4] 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.

4.3.2 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 Figure 4.3, 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 Figure 4.3, we have two groups — TCP4 and UDP4 (both with IP `0.0.0.0`).

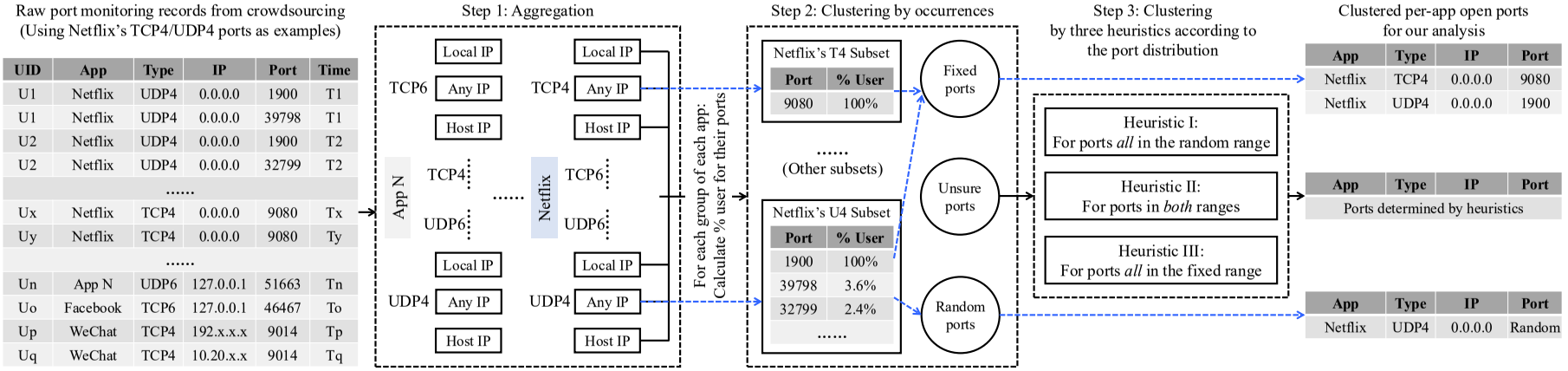


Figure 4.3: An overview of our server-side open-port analytic engine to perform the three-step clustering (using Netflix as an example).

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 Figure 4.3, 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 [25]) 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.

4.3.3 Crowdsourcing Results

We have deployed NetMon to Google Play for an IRB-approved[2] crowdsourcing study since 18 October 2016. In this chapter, 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 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. Figure 4.4 shows the distribution of open-port apps with different types of socket addresses. We find that both TCP and UDP

[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.

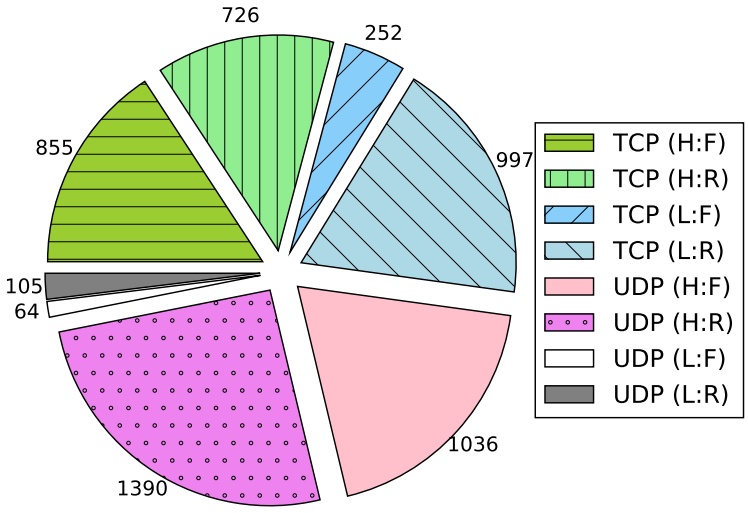


Figure 4.4: Apps with open ports in different types of socket addresses (symbols are "H"/"L": host/local IP; "F"/"R": fixed/random port number), including 1,390 apps with long-lasting client UDP ports.

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.

Open Ports in Popular Apps

With the help of Selenium [51], 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 4.1. 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 4.1, use UDP port 1900 and/or 5353 for the UPnP and mDNS services, respectively. Furthermore, the open-port

Table 4.1: 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 TCP	H L	Random Random	1B - 5B
	VK	TCP TCP	H L	48329 Random	100M - 500M
Communication	Messenger	TCP	L	Random	1B - 5B
	WeChat	TCP	H	9014	100M - 500M
	Skype	TCP TCP	H L	Random Random	500M - 1B
	Chrome	TCP	L	5555	1B - 5B
	Firefox	TCP TCP UDP	H L H	8080 Random 1900	100M - 500M
Video Players or Music & Audio	YouTube	TCP TCP	H L	Random Random	1B - 5B
	GPlay Music	TCP UDP	L H	Random 1900	1B - 5B
	Spotify	TCP	H	Random	100M - 500M
	Amazon Music	TCP TCP	L H	Random Random	100M - 500M
Tools	Google Play Services	UDP UDP	H H	2346 5353	5B - 10B
	Google	TCP	H	20817	1B - 5B
	Clean Master	TCP	L	Random	500M - 1B
	360 Security	TCP	L	Random	100M - 500M
	Avast	TCP TCP	H L	20817 Random	100M - 500M
Productivity	Google Drive	TCP	L	Random	1B - 5B
	Cloud Print	UDP	H	5353	500M - 1B
	ES File Explorer	TCP TCP TCP UDP	H H L H	42135 59777 Random 5353	100M - 500M
Entertainment	GPlay Games	TCP	L	Random	1B - 5B
	Netflix	TCP UDP UDP	H H L	9080 1900 Random	100M - 500M
	Peer Smart Remote	TCP UDP	L H	Random 5353	100M - 500M
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 Section 4.2.

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. [145] has demonstrated that such ports in iOS and OSX apps could suffer from Man-in-the-Middle attacks.

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 Section 4.4.4 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 Figure 4.4). 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 4.1, even anti-virus apps, 360 Security, and Avast, are also affected.

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).

With vendor-specific package keywords, we identify over 20 vendors that include open ports in their built-in apps. Table 4.2 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

Table 4.2: 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	-	-

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 4.2, 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 [16].

Second, about one quarter (175 apps, 23.2%) have TCP/UDP port 5060 open, which is for VoIP SIP connection setup [57]. These built-in apps are from five device vendors: Samsung, LG, Lenovo, Qualcomm, and MediaTek. By inspecting

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

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 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 [49], TCP port 59150 and 59152 for LG's Smart Share [37], and TCP port 5000 and UDP port 1024 for Sony's DLNA technique [54]. We reverse engineer Samsung Accessory and identify a security bug; see Section 4.5.1.

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 [95] (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 [95].

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 [65]. According to the experimental results in Section 4.4.3, only 58.9% of these apps can be detected by typical Android static analysis. In particular, 25.1% of them use dynamic code loading [118] or advanced code obfuscation [137]. They are therefore not possibly detected by a pure static analysis [82,120]. This indicates that crowdsourcing is much more effective than Android static analysis in the context of open port discovery.

4.4 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 (Figure 4.1) includes a diagnosis phase through an enhanced version of BackDroid that is specifically designed 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 [95], since UDP open ports have much more fixed usages (mainly for providing system-

level networking services) as we have seen in Section 4.3.3. 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 (Section 4.4.1), and then present the details of the enhanced version of our BackDroid (Section 4.4.2). Finally, we present the experiments we have performed (Section 4.4.3) and the diagnosis results (Section 4.4.4 and Section 4.4.5).

4.4.1 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 [72], especially the advertisement or analytics SDKs [90, 128], 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 4.1. 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 4.1, 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.

```
// 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 4.1: All `ServerSocket` constructor APIs.

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 isolated constants in SAAF [94]). Last but not the least, it is important for our analysis to be efficient and scalable with a large number of Android apps.

4.4.2 Design and Implementation

We design and implement an enhanced version of BackDroid to specifically handle these challenges. Instead of generating traditional slicing paths, BackDroid 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, BackDroid performs graph traversal and conducts *semantic-aware constant propagation*. We also include a preprocessing step in BackDroid 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 [17]), we use `dexdump` [24] to dump (multiple) app bytecodes into a (combined) plaintext file and then perform the searching. Additionally, for the native code, BackDroid 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 BackDroid 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. BackDroid 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.

A notable challenge for Android backward slicing is to deal with implicit flows and callbacks. BackDroid builds in support of class hierarchy, interface methods, asynchronous execution (e.g., in `Thread`, `AsyncTask`, and `Handler`), and major callbacks in the EdgeMiner list [77]. Furthermore, we support backtracking across (explicit) inter-component communication (ICC) [114], and model Android component lifecycle [69].

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 [84], we handle the following semantics:

Maintaining object semantics. To determine the correct object for each instance field, we perform *points-to* analysis [97] 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 resolv-

ing their parameters.

Removing dead ports and resolving SDK names. An important feature in the enhanced version of BackDroid is the removal of "dead ports" that are never executed. We analyze the port liveness in three steps of BackDroid. 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 [7] of the sink classes which may embed plaintext class names, as demonstrated in Google's official document [7]; and (iii) correlating different apps' open-port parameters and tags, e.g., most Alibaba AMap SDK [5] classes are obfuscated, but we can still find non-obfuscated instances, e.g., `com.amap.api.location.core.SocketService`.

4.4.3 Static Analysis Experiments

As explained in Section 4.3.3, 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 BackDroid, we identify 1,061 apps and their corresponding 1,453 TCP open ports. Figure 4.5 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

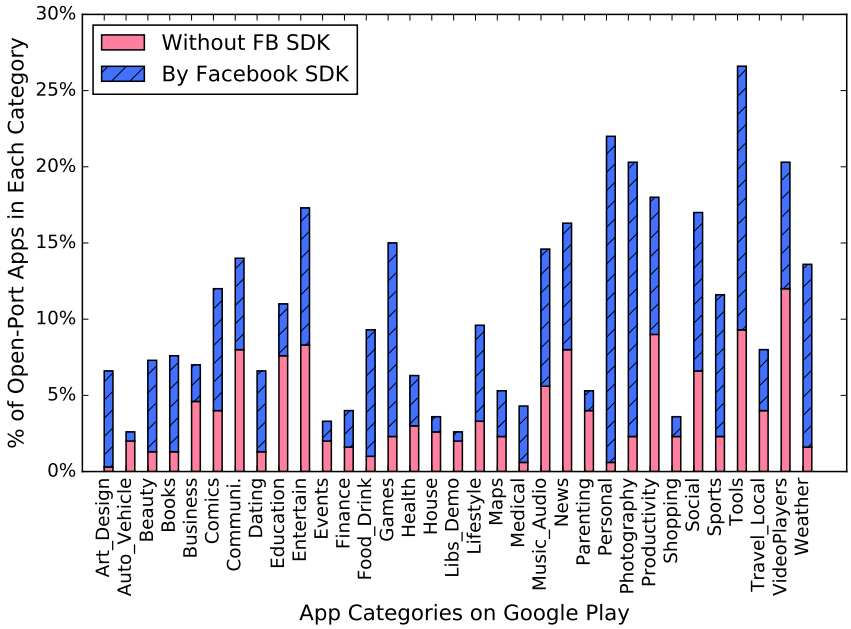


Figure 4.5: Percentage of open-port apps in each Google Play category.

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 Section 4.3.3. Out of the 1,027 open-port apps as ground truth, BackDroid 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 4.1, appear in this set). For the 671 apps analyzed by BackDroid 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., [75, 99]) that BackDroid currently cannot address,

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

even we have adopted the state-of-the-art methods [69, 73, 77]. 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.

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

4.4.4 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 4.3 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 [26] 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 Section 4.6.

Table 4.3: Open-port SDKs detected in our dataset, and the number of apps affected by them.

SDK	Pattern	#
Facebook Audience Network SDK [26]	`Class`='com.facebook.ads.%', `Tag`=ProxyCache, `Ip`=127.0.0.1, `Port`=0, `Backlog`=8	897
Yandex Metrica SDK [64]	`Class`='com.yandex.metrica.%', `Port`=29009|30102	28
CyberGarage UPnP SDK [19]	`Class`=org.cybergarage.http.HTTPServer, `Ip`=getHostAddress(), `Port`=8058|8059	19
MIT App Inventor SDK [40]	`Class`=com.google.appinventor.components.runtime.util.NanoHTTPD, `Port`=8001	19
Tencent XG Push SDK [58]	`Class`=com.tencent.android.tpush.service.XGWatchdog, `Port`=RANDOM+55000	13
Corona Game Engine SDK [18]	`Class`=com.ansca.corona.CoronaVideoView, `Port`=0, `Backlog`=8	11
Alibaba AMap SDK [5]	`Class`='com.amap.%', `Port`=43689	9
Millennial Ad SDK [39]	`Class`='com.millennialmedia.android.%', `Tag`=MillennialMediaAdSDK, `Ip`=null, `Port`=0	8
PhoneGap SDK [46]	`Class`=com.phonegap.CallbackServer, `Port`=0	6
Titanium SDK [60]	`Class`=org.appcelerator.kroll.common.TiFastDev, `Tag`=TiFastDev, `Port`=7999	6
Aol AdTech SDK [9]	`Class`=com.adtech.mobilesdk.publisher.cache.NanoHTTPD, `Port`=RANDOM+9000	6
Apache Cordova SDK [10]	`Class`=org.apache.cordova.CallbackServer, `Port`=0	4
Getui Push SDK [27]	`Class`='com.igexin.push.%', `Port`=48432|51688, `Ip`=0.0.0.0	3

We take a closer look at Table 4.3 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 [19] and two mobile push SDKs [27, 58], are networking related. The others are about advertisements [9, 26, 39, 64] (e.g., Facebook and Yandex), Javascript generation [10, 40, 46, 60] (e.g., App Inventor and PhoneGap), gaming engines [40] and map navigation [5]. Hence, we argue that developers could hardly realize the existence of these open ports by simply examining their functionality.

4.4.5 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 4.3, 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.

4.5 Security Assessment

In the last phase of our pipeline (Figure 4.1), we perform comprehensive security assessment of open ports in three directions: vulnerability analysis in Section 4.5.1, denial-of-service attack evaluation in Section 4.5.2, and inter-device connectivity measurement in Section 4.5.3.

4.5.1 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 Section 4.3.3). 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 [95] that uses predefined 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 Section 4.2, by performing in-depth reverse engineering via the state-of-the-art JEB Android decompiler [35] and extensive dynamic testing.

Table 4.4 summarizes the five vulnerability patterns we have identified. The first two have been reported in [95], while the third is a new variant of the crash vulnerability mentioned in the traditional Android app security research [85]. The last two have not been reported and they are specific to open ports.

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

Table 4.4: 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

at `http://127.0.0.1:1234//filename`, where the name can be leaked by eavesdropping Cloud Mail.Ru's broadcast messages [79]. 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 [79] `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 [49] mentioned in Section 4.3.3. 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 4.3).

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 ex-

ample, MIT App Inventor [40] and Titanium SDK [60] in Table 4.3 use open ports for instant debugging or the so-called living programming [123]. 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 [85]. Now open ports provide a new channel for launching CoS. For example, we 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 Section 4.3.3) 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 [8]. 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

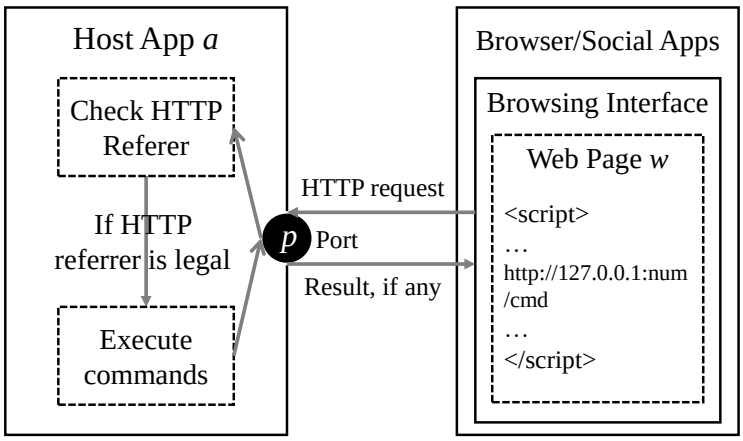


Figure 4.6: The model of using open ports for analytics.

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 Section 4.3.3) and Corona Game Engine SDK (in Table 4.3). 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. Figure 4.6 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 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

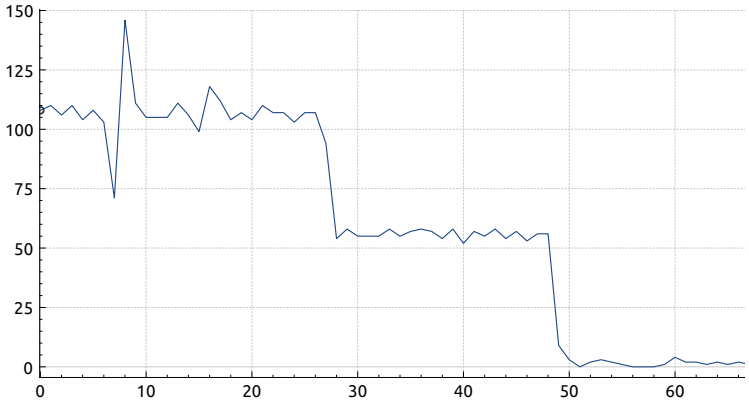
local apps or even a remote adversary (if the open port bound to the host IP address, which is often the case) can set an arbitrary referrer in their HTTP requests to execute privileged commands (e.g., retrieving IMEI and list of installed apps). We uncover this class of vulnerabilities in Sina Weibo, Alibaba AMap SDK, and two Baidu SDKs (which were fixed quite long ago and thus not in Table 4.3). We reported these issues to the vendors in the first half of 2015, much earlier than the subsequent industrial reports (e.g., WormHole [52])[5].

4.5.2 Denial-of-Service Attack Evaluation

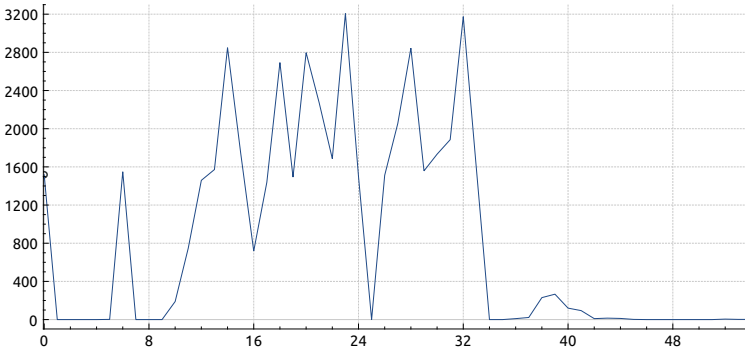
We now evaluate denial-of-service (DoS) attacks against mobile open ports and their effectiveness. Note that this analysis differs from those in Section 4.5.1 in that DoS attacks are typically possible even without exploiting any code-level vulnerabilities. Different from the traditional DoS attacks 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 Section 4.3.3).

Figure 4.7 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` [29] 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. Figure 4.7(a) shows that the throughput of WeChat's

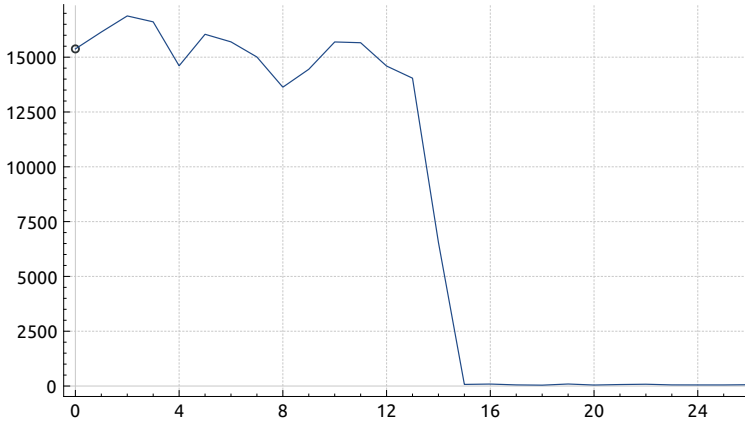
[5]A list of our original reports (in Chinese) can be found at `https://tinyurl.com/opWooyun`, and cached at `https://tinyurl.com/opDropbox`.



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



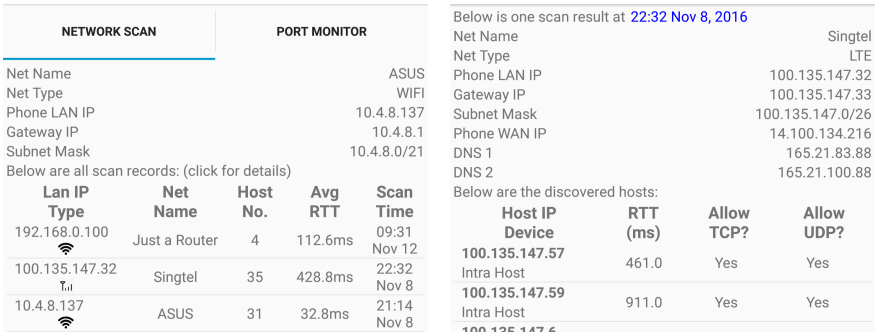
(b) YouTube's video streaming (DoS at ∼32s).



(c) AirDroid's file transmission (DoS at ∼13s).

Figure 4.7: DoS attacks against open ports. The x-axis is the time in seconds, and the y-axis is the victim apps' throughput (packets/sec).

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). Figure 4.7(b) and Figure 4.7(c) 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



(a) A list of networks scanned. (b) Detailed results of one scan.

Figure 4.8: User interfaces in NetMon for network scans.

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 Section 4.5.3). 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, 30].

4.5.3 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. Figure 4.8 presents its two user interfaces, in which Figure 4.8(a) shows a partial list of networks scanned and the detailed results are shown in Figure 4.8(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 mea-

sure 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 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 Section 4.3.3). 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 Section 4.3.3, 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.

4.6 Mitigation Suggestions

To mitigate the threats of open ports, we propose countermeasures 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` [38] 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 Section 4.4.5 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 Section 4.5.1).

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 Section 4.4.1, 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.

4.7 Summary

In this chapter, 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 BackDroid's static analysis enhanced with open-port semantics and SDK identification capability. 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.

Chapter 5

Measuring Declared SDK Versions and Their Inconsistency with API Calls in Android Applications

5.1 Introduction

Along with the fast-evolving Android, its fragmentation problem becomes more and more serious. Although new devices ship with the recent Android versions, there are still huge amounts of existing devices running old Android versions [67]. To better manage the application's compatibility across multiple platform versions, Android allows apps to declare the supported platform SDK versions in their manifest files. We term these declared SDK versions as `DSDK` versions. The `DSDK` mechanism is a modern software mechanism with which, to the best of our knowledge, few systems are equipped until Android. Nevertheless, so far it receives little attention and few understandings are known about the effectiveness of the `DSDK` mechanism.

In this chapter, we aim to conduct a systematic study on the Android `DSDK` mechanism. Specifically, our objective is to measure the current practice of `DSDK` versions in real apps, and the (in)consistency between `DSDK` versions and their host apps' API calls. To make our measurement results representative, we select popu-

lar apps that have at least one million installs each on Google Play as the dataset. More specifically, we have collected a large-scale dataset with 22,687 popular apps (570.8GB in total, with an average app size of 25MB), which covers 90.2% of all such apps (both free and paid ones) available on Google Play. Furthermore, our study utilizes the latest Android API evolution and covers all 28 versions of Android SDKs or API levels[1]. We also find an effective way to accurately map 39,034 APIs to their corresponding SDK versions.

After selecting the dataset and building the API-SDK mapping, we perform a systematic `DSDK` and API call analysis of each individual app. Our design is robust and scalable so that it can be readily deployed by online app markets (e.g., Google Play) to timely notify developers of the `DSDK` inconsistency in their apps. Given this objective, dataflow-based analysis is not very suitable because existing Android dataflow analyses (notably FlowDroid [69] and Amandroid [132]) are expensive even when analyzing medium apps, e.g., requiring $\sim$4 minutes for the 8MB Nextcloud app[2] [93]. Moreover, they need to first transform or decompile Android app bytecode into an intermediate representation (usually Java bytecode), the process of which is not fully accurate [113] and often leaves some apps unanalyzable in many previous studies [149] [71] [108] [116].

In our approach, we thus operate on the original Android bytecode level and employ a lightweight version of BackDroid for app analysis. Specifically, we retrieve `DSDK` versions and API calls *directly* from each app without decoding the manifest file via `apktool` [11] or decompiling app bytecodes via `dex2jar` [22], which enables robust processing of all 22,687 popular apps. We also handle multidex [17], a special Android bytecode format often skipped by prior works but is common in modern apps — 5,008 apps in our dataset split their bytecodes into multiple files. With the correctly extracted app bytecodes, our lightweight BackDroid searches these bytecode texts to obtain valid API calls that are not guarded by

[1]The latest Android version at the time of our writing is Android 9 (API level 28).
[2]`https://f-droid.org/en/packages/com.nextcloud.client/`

VERSION.SDK_INT checking (developers can use such if statements to invoke an API only in certain Android platforms) and also not in uninvoked third-party libraries. In this way, it preserves a scalability suitable for online vetting: the median and average time for analyzing an app in our dataset is only 4.75s and 5.39s, respectively. Moreover, the number of inconsistency warnings per app reported is well manageable for developers to perform a one-time manual check, with fewer than 10 potentially inconsistent API calls in around 80% apps each.

Our study sheds light on the current DSDK practice by app developers and quantitatively measures two side effects caused by the inconsistency between DSDK versions (configured by the app developers in the manifest file) and API calls (made by the app during its execution). Specifically, the compatibility effect occurs when a minimum DSDK version is set too low that certain APIs do not even exist in the corresponding lower versions of Android platforms. The consequence of such compatibility effect can cause runtime crashes. Additionally, the security effect could also happen when a target DSDK version is outdated (i.e., a lower version is used despite device actually running on later versions of Android), causing that a vulnerable API is still rendered by the underlying system even when the app runs on higher versions of Android. Next, we present our three sets of measurement results on DSDK versions and their inconsistency with API calls.

Firstly, our measurement reveals some interesting characteristics of declared SDK versions in the wild. Specifically, nearly all apps define the minSdkVersion attribute, but 4.76% apps still do not claim the targetSdkVersion attribute (in our dataset obtained in 2018), although this percentage has significantly dropped from 16.54% in 2015. This indicates that DSDK attributes nowadays are more widely adopted in modern apps. We further find that the minimal platform version most apps support nowadays is Android 4.1, whereas the most popular targeted platform version is Android 8.0. The median version difference between targetSdkVersion and minSdkVersion also increases from 8 in our last analysis in 2015 to 9 currently in the 2018 dataset.

Secondly, in terms of compatibility inconsistency, we first find that around 50% apps under-set the `minSdkVersion` value, causing them to crash when running on lower versions of Android platforms. Fortunately, only 11.3% apps could crash on Android 6.0 and above. We also show that by employing bytecode search for `SDK_INT` checking, our approach can reduce 17.3% false positives of compatibility inconsistency results. A detailed analysis of Android APIs that incur compatibility inconsistency further reveals that some API classes, such as view, webkit, and system manager related classes, are commonly misused.

Thirdly, our analysis of security inconsistency shows that around 2% apps still set an outdated `targetSdkVersion` attribute when a common WebView API is vulnerable, making them exploitable by remote code execution. In particular, around a half of these vulnerable apps invoke the vulnerable `addJavascriptInterface()` API call because of their embedded third-party libraries. Moreover, our bytecode search of the `addJavascriptInterface()` invocation also helps reduce 12.2% false positives.

To summarize, we highlight the contributions of this chapter as follows:

- (*New problem*) To the best of our knowledge, we are the first to conduct a systematic study on `DSDK`, a modern software mechanism that allows apps to declare the supported platform SDK versions. We also give the first demystification of the `DSDK` mechanism and its two side effects on compatibility and security. In particular, our paper [143] has motivated several recent follow-up works [100] [93] on bug detection.

- (*Novel approach*) We propose a robust and scalable approach that operates on the original bytecode level and leverages lightweight bytecode search in BackDroid to timely notify developers the `DSDK` inconsistency in their apps. The evaluation using 22,687 popular apps (with an average app size as large as 25MB) shows that our approach achieves good performance suitable for online app vetting, requiring only $\sim$5 seconds to process an app on average.

- (*New findings*) Our measurement study obtains three major new findings, including (i) 4.76% apps still do not claim the `targetSdkVersion` attribute, although this percentage has significantly dropped from 2015 to 2018, (ii) around 50% apps under-set the minimum `DSDK` versions and could incur run-time crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above, and (iii) around 2% apps, due to under-claiming the targeted `DSDK` versions, are potentially exploitable by remote code execution, and a half of them actually invoke the vulnerable API via embedded third-party libraries.

5.2 Demystifying Declared SDK Versions and Their Two Side Effects

In this section, we first demystify declared platform SDK versions in Android apps, and then explain their two side effects if inappropriate `DSDK` versions are used. Note that `DSDK` is different from the typical compilation SDK, which is only for compiling apps while `DSDK` is mainly for interpreting run-time API behaviors.

5.2.1 Declared SDK Versions in Android Apps

Listing 5.1 illustrates how to declare supported platform SDK versions in Android apps by defining the `<uses-sdk>` element [62] in apps' manifest files (i.e., `AndroidManifest.xml` [59]). These `DSDK` versions are for the runtime Android system to check apps' compatibility, which is different from the compiling-time SDK for compiling source codes. The value of each `DSDK` version is an integer, which represents the API level of the corresponding SDK. For example, if a developer wants to declare Android SDK version 5.0, she can set its value to 21. Since each API level has a precise mapping of the corresponding SDK version [68], we do not use another term, *declared API level*, to represent the same meaning of `DSDK` throughout this chapter.

```
<uses-sdk android:minSdkVersion="integer"
        android:targetSdkVersion="integer"
        android:maxSdkVersion="integer" />
```

Listing 5.1: The syntax for declaring platform SDK versions in Android apps.

We explain the three `DSDK` attributes as follows:

- The `minSdkVersion` integer specifies the minimum platform API level required for an app to run. The Android system refuses to install an app if its `minSdkVersion` value is greater than the system's API level. Note that if an app does not declare this attribute, the system by default assigns the value of "1", which means that the app can be installed in all versions of Android.

- The `targetSdkVersion` integer designates the platform API level that an app targets at. An important *implication* of this attribute is that Android adopts backward-compatible API behaviors of the declared target SDK version, even when an app is running on a higher version of the Android platform. Android makes such compromised design because it aims to guarantee the same app behaviors as expected by developers, even when apps run on newer platforms. It is worth noting that if this attribute is not set, the default `minSdkVersion` is used.

- The `maxSdkVersion` integer specifies the maximum platform API level on which an app can run. However, this attribute is *not* recommended and already deprecated since Android 2.1 (API level 7). Modern Android no longer checks or enforces this attribute during the app installation or re-validation. The only effect is that Google Play continues to use this attribute as a filter when it presents users a list of applications available for downloading. Note that if this attribute is not set, it implies no restriction on the maximum platform API level.

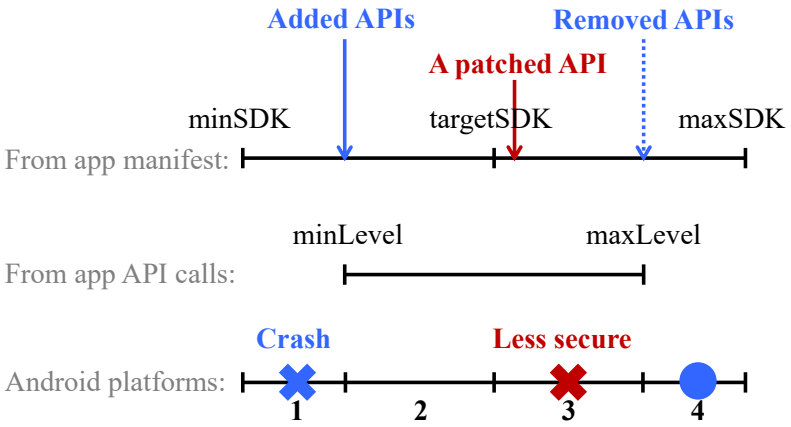


Figure 5.1: Illustrating the two side effects of inappropriate `DSDK` versions.

5.2.2 Two Side Effects of Inappropriate DSDK Versions

Figure 5.1 illustrates two side effects of inappropriate `DSDK` versions. We first explain the symbols used, and then describe the two side effects. As shown in Figure 5.1, we can obtain $minSDK$, $targetSDK$, and $maxSDK$ from an app manifest file. Based on the API calls of an app, we can calculate the minimum and maximum API levels it requires, i.e., $minLevel$ and $maxLevel$. Eventually, the app will be deployed to a range of Android platforms between $minSDK$ and $maxSDK$.

Side Effect I: Causing Runtime Crashes

The blue part of Figure 5.1 shows two scenarios in which inappropriate `DSDK` versions could cause compatibility-related inconsistency. The first scenario is $minLevel > minSDK$, which means a new API is introduced after the $minSDK$. Consequently, when an app runs on Android platforms between $minSDK$ and $minLevel$ (marked as the block 1 in Figure 5.1), it will crash. We verified this case by using `VpnService` class's `addDisallowedApplication()` API, which was introduced on Android 5.0 at API level 21. We invoked this API in the MopEye app [140] and ran it on an Android 4.4 device. When the app executed the `addDisallowedApplication()` API call, it crashed with the `java.lang.NoSuchMethodError` exception.

The second scenario is $maxSDK > maxLevel$, which means an old API is removed at the $maxLevel$. Although it looks like the app would crash when it runs on Android platforms between $maxLevel$ and $maxSDK$, it turns out that Google intentionally keeps the forward compatibility (by keeping those removed APIs in the framework as hidden APIs) so that developers have no concern in over-setting `maxSdkVersion`. As a result, this scenario would not cause runtime method availability errors. Therefore, we measure only the first scenario of compatibility inconsistency that can cause runtime crashes in this chapter.

Side Effect II: Making Apps Vulnerable

The red part of Figure 5.1 shows the scenario in which inappropriate `DSDK` versions cause failure for the app to be patched. Suppose that an app calls an API whose implementation is vulnerable at $targetSDK$, even when the app runs on an updated Android system (with API level $> targetSDK$). In this case, Android still exhibits the compatibility behaviors, i.e., the vulnerable implementation of the API at $targetSDK$ in this case.

Table 5.1: Vulnerable APIs or components on Android and their patched versions.

Vulnerable APIs/Components	Patched SDKs (Android)	Changed Behavior
`file://` scheme in WebView	$targetSDK \geq 16$ (4.1+)	Fix flawed same-origin policy [138]
Content Provider component	$targetSDK \geq 17$ (4.2+)	Do not by default export [12]
`addJavascriptInterface()`	$targetSDK \geq 17$ (4.2+)	Stop Java reflection for RCE [21]
`PreferenceActivity` class	$targetSDK \geq 19$ (4.4+)	Add `isValidFragment()` for apps to prevent Fragment Hijacking [28]
`javascript:` in WebView	$targetSDK \geq 19$ (4.4+)	JavaScript URLs are executed in a separate WebView context [110]
`Context.bindService()`	$targetSDK \geq 21$ (5.0+)	Do not accept Implicit Intents [50]

Table 5.1 summarizes previously reported vulnerable APIs or components on Android and their patched versions. In this chapter, we choose to particularly measure the vulnerable `addJavascriptInterface()` API for two reasons. First, it has a clear API pattern for inconsistency measurement, while other cases in Table 5.1 involve multiple component-level factors that could cause a vulnerability. Second, the `addJavascriptInterface()` API gives rise to the most serious

security issue [81]. By exploiting this API, attackers are able to inject malicious code, which can cause remote code execution (e.g., stealing sensitive information from a victim app or SD card). Google later fixed this weakness on Android 4.2 and above. However, if an app sets the `targetSdkVersion` to be lower than 17 and calls this API, the system will still render the vulnerable API behavior even when running on Android 4.2+. Such vulnerable app examples are available at `https://sites.google.com/site/androidrce/`.

5.3 Methodology

To understand how `DSDK` versions are used in the wild and the pervasiveness of the two side effects in real apps, we propose an automatic approach for a systematic measurement. In this section, we first present an overview of our methodology, and then its two main analysis phases.

5.3.1 Overview

We design our approach with the objective of it being deployed by app markets to timely notify developers the `DSDK` inconsistency in their apps. Figure 5.2 illustrates its overall design, where the app analysis part is conducted in the online phase. Since our app analysis requires the *API-SDK mapping* as an input (for calculating API levels of all valid API calls in an app), we further conduct Android API document analysis to build a mapping between each Android API and their corresponding SDK versions (or API levels). As this step is performed only once, we include it in the offline phase.

The major part of our approach is designed for the online vetting of apps. Specifically, whenever developers upload a new or updated app to app markets, we first unzip this app to obtain its bytecode DEX file(s). We then launch manifest analysis to robustly retrieve an app's declared SDK versions. For bytecode analysis, our novelty is to leverage the lightweight bytecode search in BackDroid, instead

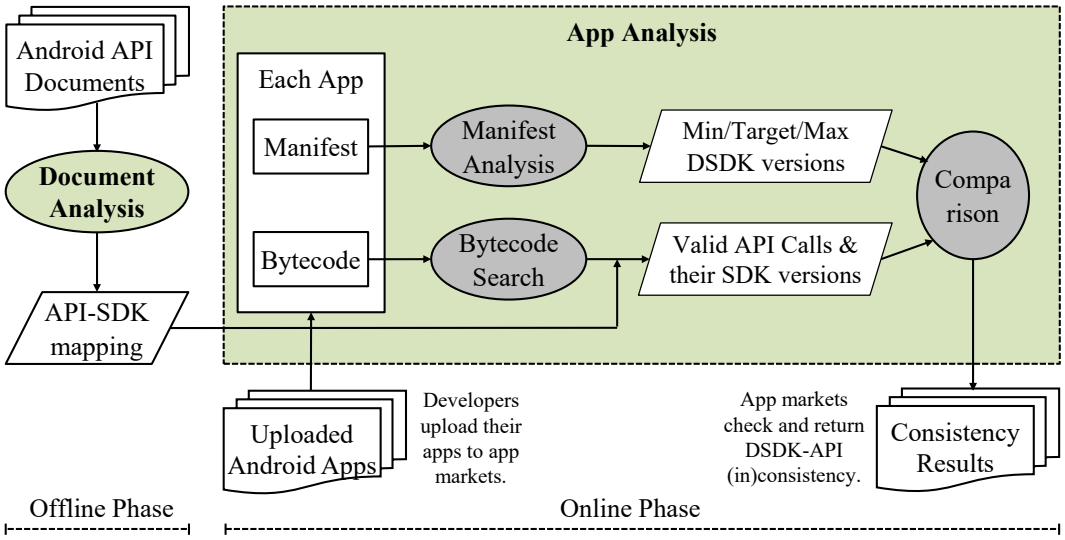


Figure 5.2: The overview of our methodology.

of heavyweight dataflow analysis, to extract valid API calls. Finally, we leverage the API-SDK mapping to calculate the range of the corresponding API levels from API calls, and compare them with the declared SDK versions. The output is the (in)consistency results between declared SDK versions and API calls. It is worth noting that *multiple-apk* analysis [143] is no longer needed in our online analysis, because app markets control all versions of APKs and multiple-apk mechanism is largely used for different hardware configuration [41].

5.3.2 Offline Phase: API Document Analysis

In this subsection, we present our offline phase in detail, including both methodology and results of API document analysis.

Building the API-SDK mapping. There are two potential approaches for building the API-SDK mapping. One is to analyze Android API documents by parsing a SDK document called `api-versions.xml`. A previous API study [109] and our preliminary results reported in [143] leveraged this approach to obtain initial and added APIs, but they did not cover removed and deprecated APIs via the `api-versions.xml` file they analyzed. They thus also needed to analyze the HTML files in the `api_diff` directory, which is, unfortunately, error-prone [143].

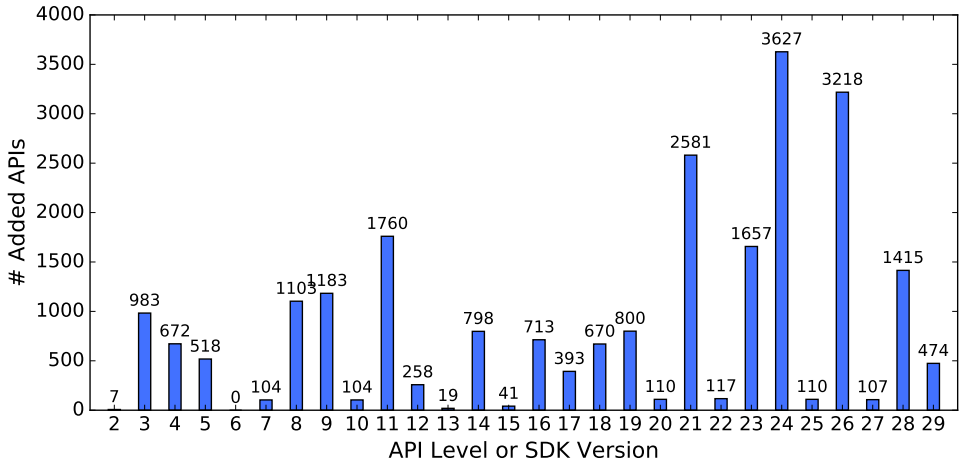


Figure 5.3: Distribution of added Android APIs across different SDK versions.

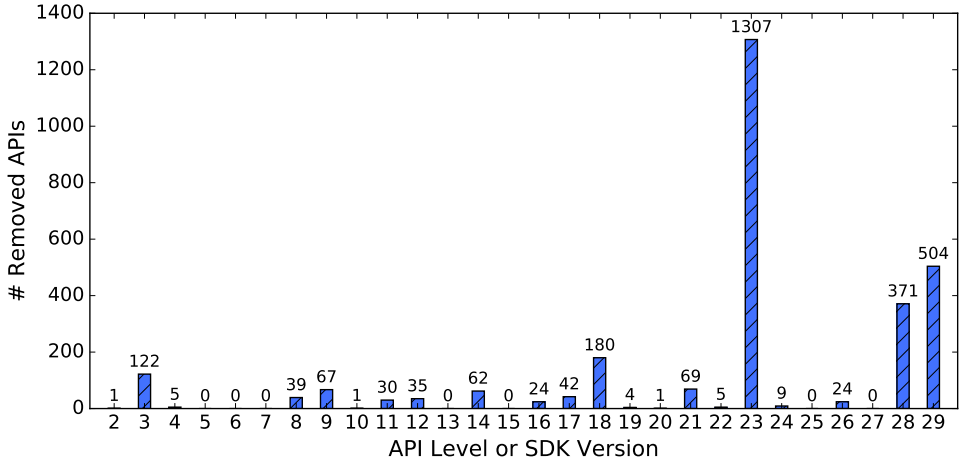


Figure 5.4: Distribution of removed Android APIs across different SDK versions.

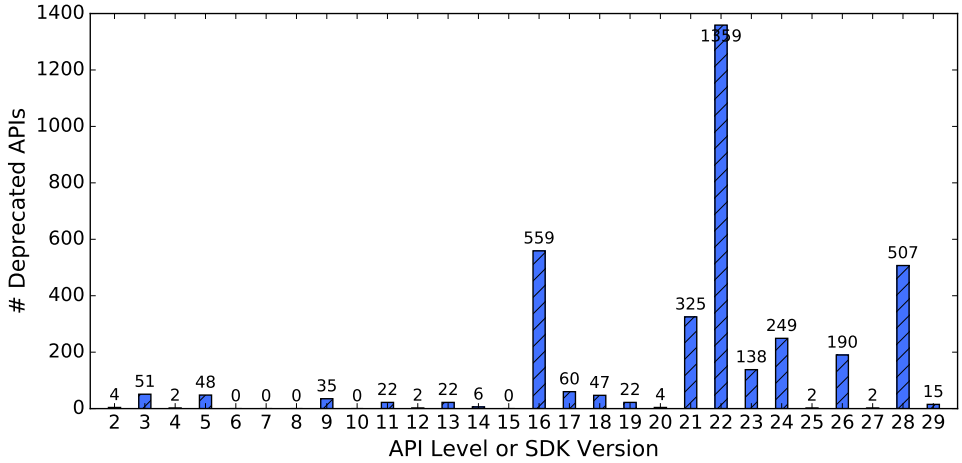


Figure 5.5: Distribution of deprecated Android APIs across different SDK versions.

The other approach is to directly retrieve the API-SDK mapping from each SDK `jar` file. However, different SDK releases under the same API level may have some API differences, and there are over 600 releases[3] for 28 API levels at the time of our writing. As a result, conflicted API mappings could be recorded, e.g., marking the `Gravity.getAbsoluteGravity` API that was removed in SDK version 16 and then added back in version 17 [100].

Fortunately, we find that the first approach provides an accurate mechanism to cover all kinds of Android APIs. Specifically, the latest `api-versions.xml` file released in Android 9 SDK records all added, removed, and deprecated APIs. Therefore, we can simply parse this file to obtain an accurate API-SDK mapping.

Document analysis results. With the accurate API-SDK mapping, we are now able to present a comprehensive evolution of Android APIs across different SDK versions. Figure 5.3, 5.4, and 5.5 plot the distribution of added, removed, and deprecated Android APIs from API level 2 to the very recent API level 29, respectively. Overall, we find that 26,466 (67.8%) out of a total of 39,034 Android APIs are changed. This result shows that Android APIs evolve dramatically during the whole evolution.

The biggest change in the Android API evolution is to add 23,542 APIs since level 2, as shown in Figure 5.3. Specifically, Android 7.0 (API level 24) changed most, with 3,627 new APIs introduced. Android 8.0 (API level 26) and Android 5.0 (API level 21) also introduce a significant number of new APIs, with 3,218 and 2,581 APIs added, respectively. Other versions of platforms with a large number of added APIs are Android 3.0 (API level 11), Android 6.0 (API level 23), and Android 9.0 (API level 28). These new Android APIs bring a huge risk of compatibility inconsistency, causing runtime crashes on lower versions of Android. In particular, we notice that over half (13,306, 56.5%) of all added APIs are introduced since Android 5.0, giving them a higher chance of causing compatibility inconsistency

[3]See tags in `https://android.googlesource.com/platform/frameworks/base.git/+refs`.

than the rest of APIs added.

In contrast, only 4,830 (18.2%) APIs involve the removal change (i.e., removed or deprecated; some of them are also introduced after API level 2), with 3,671 APIs deprecated and 2,902 APIs finally removed. According to Figure 5.4 and 5.5, the biggest removal happens in Android 5.1 and 6.0 (API level 22 and 23), with 1,359 APIs deprecated and 1,307 APIs removed afterwards. Moreover, Android 9.0 (API level 28) deprecates 507 APIs and its next version (API level 29) removes 504 of them, which suggests that Google plans to remove a large number of APIs in the release of Android 9.0. Additionally, although Android 4.1 (API level 16) deprecated 559 APIs, only 222 APIs were removed in the subsequent Android 4.2 and 4.3.

To sum up, 23,542 (60.3%) out of all the 39,034 Android APIs are introduced at a SDK version other than the initial Android SDK version (i.e., API level 1), which brings a high risk for developers to under-set the `minSdkVersion` attribute. On the other hand, much fewer Android APIs, i.e., 7.4% of all APIs, are mapped to a range of SDK versions that have an upper limit.

5.3.3 Online Phase: Android App Analysis

In this subsection, we present three major modules in the online analysis phase, namely manifest analysis, bytecode search, and consistency comparison in Figure 5.2.

Retrieving `DSDK` Versions via Manifest Analysis

To robustly retrieve `DSDK` versions from all apps, we propose a new manifest analysis method that leverages `aapt` (Android Asset Packaging Tool) [2] to retrieve `DSDK` *directly* from each app without extracting and decoding the manifest file. This method is more robust than the traditional `apktool`-based manifest extraction [11] that requires to extract and decode the manifest into a plaintext file. Indeed,

```
$ aapt dump badging example.apk
package: name='com.example' versionCode='1' versionName='1.0'
sdkVersion:'8'
targetSdkVersion:'20'
maxSdkVersion:'22'
```

Figure 5.6: The `aapt` command for retrieving `DSDK` directly from an APK file.

our `aapt`-based approach can successfully analyze all 22,687 apps, whereas a previous work [144] showed that `apktool` failed six times in the analysis of just 1K apps. Specifically, we utilize the `dump baging` command in `aapt` to extract the `DSDK` versions, as shown in Figure 5.6. We can see that `minSdkVersion` is now represented with the "sdkVersion" keyword, whereas `targetSdkVersion` and `maxSdkVersion` still remain the same as in the manifest.

In the course of implementation and evaluation, we observed and handled two kinds of special cases. First, some apps define `minSdkVersion` multiple times, for which we only extract the first value. Second, we apply the default rules (see Section 5.2.1) for apps without `minSdkVersion` and `targetSdkVersion` defined. More specifically, we set the value of `minSdkVersion` to 1 if it is not defined, and set the value of `targetSdkVersion` (if it is not defined) using the `minSdkVersion` value.

Besides retrieving `DSDK`, our manifest analysis also parses all components registered in the manifest to generate a list of valid components and their root (Java) class names. This information will be used in the app analysis module to exclude uninvoked third-party libraries. Specifically, we execute the `dump xmltree` command in `aapt` to output all component information. In the process of parsing these components, we also generate their root class names according to this rule: if the component class does not overlap with the app package or `<application>` name (i.e., this class could be from a third-party library), we record the entire class name as the root class; otherwise, only the leading two or three name portions are treated as the root class.

Extracting Valid API Calls via Bytecode Search

The main module in our app analysis is to extract valid API calls. A valid API call should not be guarded by the `VERSION.SDK_INT` checking (a mechanism developers can use to invoke an API only in certain Android platforms). It also should not be in uninvoked third-party libraries that are essentially dead code. To guarantee the scalability for online vetting, we propose a lightweight version of BackDroid for app analysis, because existing Android dataflow analyses, notably FlowDroid [69] and Amandroid [132], are expensive even when analyzing medium apps, e.g., requiring ∼4 minutes for just an app of size 8MB [93].

Moreover, we operate on the original Android bytecode level without decompiling app bytecodes for minimizing false negatives. This is because the process of transforming or decompiling Android app bytecode into an intermediate representation (usually Java bytecode) is not fully accurate [113]. As a result, many previous studies [149] [71] [108] [116] often failed to handle some apps, causing false negatives in their analysis. In contrast, by directly analyzing app bytecodes, we robustly process all 22,687 popular apps in our dataset. Specifically, we leverage the `dexdump` tool [23] to translate compressed bytecodes into bytecode plaintexts (similar to using `objdump` to generate assembly code texts), upon which we can then launch bytecode search to extract valid API calls. Note that `dexdump`, as an official Android SDK tool, is very robust, and it does not generate intermediate representation. We also dump (multiple) app bytecodes into a (combined) plaintext [142] to handle multidex [17], a special bytecode format often skipped by prior works but indeed common in modern apps — 5,008 apps in our dataset split their bytecodes into multiple files. Hence, we avoid another common source of false negatives.

In the rest of this subsection, we first introduce the basic bytecode search mechanism before describing our bytecode search of `VERSION.SDK_INT` checking and vulnerable API calls in details. We then explain how we exclude uninvoked third-

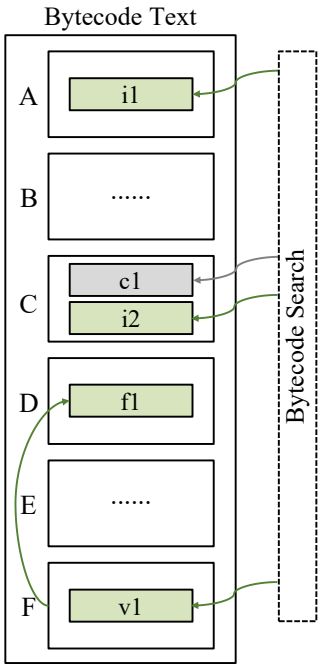


Figure 5.7: A high-level overview of our bytecode search mechanism.

party libraries during the search process.

The basic bytecode search mechanism. Figure 5.7 shows a high-level overview of our bytecode search mechanism. The bytecode text outputted by `dexdump` is a sequence of code statements, hierarchically organized by different class and method bodies. In Figure 5.7, we show six method bodies (from method A to method F), where their corresponding class bodies are omitted for simplicity. As illustrated in the figure, our bytecode search scans these methods to locate inconsistent API calls (e.g., call site i1 and i2 in method A and C, respectively) and vulnerable API calls (e.g., call site v1 in method F). We can perform further search to determine in which class an interested method is invoked, e.g., Figure 5.7 shows that method F (containing vulnerable API call v1) is called by another method D. Besides the method search, we can also launch `if` statement search to locate conditional checking, e.g., statement c1 that surrounds call site i2 in method C.

Searching `VERSION.SDK_INT` checking. As mentioned earlier in this subsection, developers can use `if` statements with `VERSION.SDK_INT` checking to invoke an API only in certain Android platforms, thus avoiding the inconsistency

```
1 VpnService.Builder builder = new VpnService.Builder();
2 if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
3     builder.addDisallowedApplication(Constant.PkgName);
4 }
```

Listing 5.2: An example of `VERSION.SDK_INT` checking.

problem. Listing 5.2 shows an example of `VERSION.SDK_INT` checking, which invokes the `addDisallowedApplication()` API (introduced since API level 21) only on Android 5.0 and above. To avoid such false positives, our approach must handle the `VERSION.SDK_INT` checking.

Our strategy is to make both API call and `VERSION.SDK_INT` checking search and see whether the two search results overlap in the same method. For example, in Figure 5.7, our bytecode search locates both checking statement c1 and API call i2 in method C. Since these two search results overlap and API call i2 is invoked below checking statement c1, we are thus confident that this API call has been guarded with a corresponding `VERSION.SDK_INT` checking.

Searching vulnerable API calls. For a vulnerable API call, we further employ bytecode search to determine whether it is initialized by app's own code or library code. This is particularly important for the vulnerable API studied in this chapter, namely `addJavascriptInterface()`, because a previous study has shown that over 47% of top 40 ad libraries create their JavaScript interfaces [36]. Specifically, after locating vulnerable API call v1 in method F, we further search the invocation(s) of method F to check its origin class.

Excluding uninvoked third-party libraries. An important issue during our bytecode search is to exclude uninvoked third-party libraries. Previous works (e.g., Amandroid [132] and CiD [100]) use a pre-collected white list to skip third-party libraries, but this approach also ignores valid library code. Differently, we consider all components registered in the manifest, including those from third-party libraries. As mentioned in Section 5.3.3, we generate root classes for all registered components via manifest analysis. A class code that does not appear in any root class is

thus from an uninvoked third-party library or dead code. But even for a valid third-party library, only its registered components will be analyzed because not all code in a library will be invoked by the main app.

Calculating API Levels and Comparing Their Consistency with DSDKs

With the extracted API calls, we use the API-SDK mapping to compute the range of their corresponding API levels (i.e., from `minLevel` to `maxLevel`, as explained in Section 5.2.2). The `minLevel` of an app is the maximum of all its valid API calls' corresponding `minLevel` values (i.e., all correspondingly added SDK versions), while the `maxLevel` of an app is the minimum of all valid API calls' corresponding `maxLevel` values (i.e., all correspondingly removed SDK versions). If an API is never removed, we set a large flag value (e.g., 100,000) to represent its `maxLevel` value.

We then compare the extracted `DSDK` values with the calculated API levels to obtain the following two kinds of inconsistency (as previously mentioned in Section 5.2.2):

- `minSdkVersion` $<$ `minLevel`: the `minSdkVersion` is set too low and the app would crash when it runs on platform versions between `minSdkVersion` and `minLevel`.
- `targetSdkVersion` $<$ `maxLevel`: the `targetSdkVersion` is set too low and the app could be updated to the version of `maxLevel`. If the `maxLevel` is infinite, the `targetSdkVersion` could be adjusted to the latest Android version.

5.4 Evaluation

Our evaluation aims to answer the following four research questions:

RQ1 What are the *current `DSDK` characteristics* in popular real-world apps?

RQ2 How pervasive is the *compatibility-related inconsistency* in real-world apps?

RQ3 How pervasive is the *security-related inconsistency* is in real-world apps?

RQ4 How *scalable* is our inconsistency detection approach?

We choose popular real-world apps, instead of randomly selected apps or open-source apps, for evaluation, because they are most likely installed by regular users. Hence, the obtained measurement results can reflect the `DSDK` practice in the wild. In this section, we first describe how we collect such a large dataset in Section 5.4.1. Based on this dataset, we then answer the four research questions from Section 5.4.2 to 5.4.5.

5.4.1 Dataset

We collect popular apps on Google Play via the AndroZoo repository [65], which contains a total of 3,699,731 unique[4] Google Play apps at the time of our crawling on 11 November 2018. However, AndroZoo does not provide the app install information, which is required to determine the popularity of each app. To quickly locate popular apps, we leverage the top app lists available at `https://www.androidrank.org`. Specifically, we crawled top 1,000 app in each Google Play category (49 categories in total, including 17 different game sub-categories), and recorded the package names of apps *with over one million installs*. This allows us to obtain a list of 25,144 popular apps, 22,687 (the rest are either paid apps or not indexed by AndroZoo) of which are available on AndroZoo. We thus downloaded these 22,687 apps as our dataset.

To understand these popular apps' distribution across different app categories, we plot a bar chart in Figure 5.8 that covers all 32 non-game app categories. Additionally, 17 game sub-categories contribute to a total of 10,695 popular apps, which indicates that game apps are commonly installed by real-world Android users. Ac-

[4] An app is unique if its package name, instead of SHA1/256, is different from other apps.

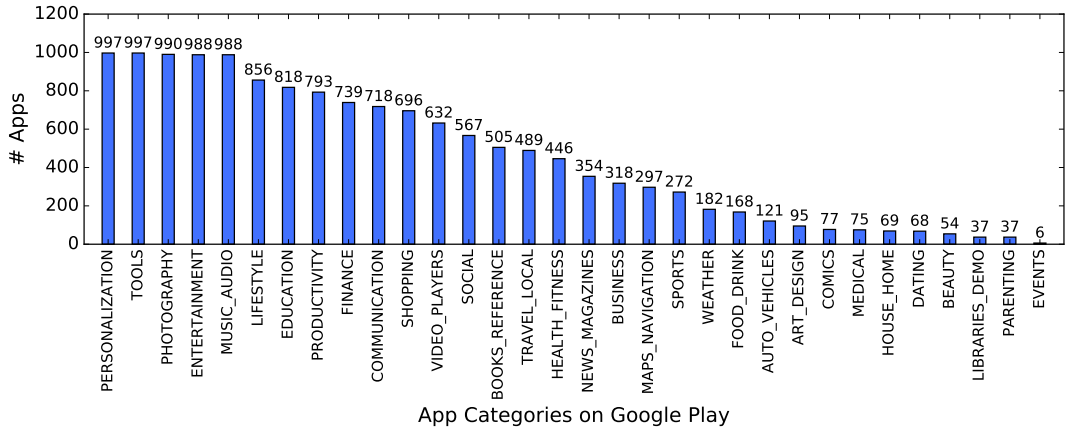


Figure 5.8: The distribution of popular apps across different categories.

cording to Figure 5.8, app categories like "Personalization", "Tools", "Photography", "Entertainment", and "Music" also produce a large number of popular apps, almost 1K popular apps per category. We notice that daily-used categories, such as "Communication" and "Social", however, do not generate an equivalent number of popular apps, with only 600 to 700 popular apps. This is because in these categories, several very popular apps, e.g., WeChat and Facebook, dominate a large portion of the market share. Lastly, it is also reasonable for some unpopular categories, such as "Medical" and "Libraries & Demo", to have a limited number of popular apps.

It is also important to measure the distribution of app size in our dataset. Figure 5.9 plots the CDF (cumulative distribution function) of the APK file size of each app in our dataset. We can see that over 40% apps have a size larger than 20MB, and over 20% apps are even larger than 40MB each. This indicates that a significant portion of modern apps are no longer in small size. Indeed, the average app size in our dataset is 25MB, much larger than the size of apps used in several prior dataflow analysis studies (e.g., apps below 5MB were evaluated in AppContext [149], and the maximum app size in IctApiFinder [93] is 8MB). Therefore, scalability is a key design objective for our approach, and we will evaluate it extensively in Section 5.4.5.

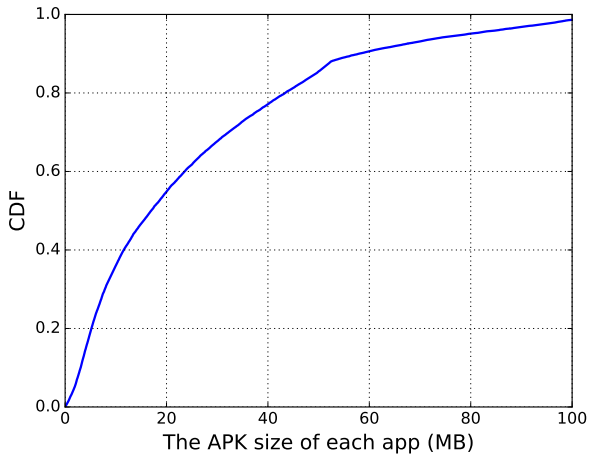


Figure 5.9: CDF plot of the APK file size of each app in our dataset.

5.4.2 RQ1: Characteristics of Declared SDK Versions in the Wild

In this section, we report a total of four findings regarding RQ1. We also compare these new findings with our previous results in [143], which measured a dataset of 22.7K apps crawled in 2015.

Finding 1-1: *Nearly all apps define the* `minSdkVersion` *attribute, but there are still 4.76% apps not claiming the* `targetSdkVersion` *attribute, although this percentage has significantly dropped compared to our prior analysis in 2015.* Table 5.2 shows the number and percentage of non-defined `DSDK` attributes in our dataset. We can see that nearly all apps have defined the `minSdkVersion` attribute while nearly no apps define the `maxSdkVersion` attribute. This result is good because, as we described in Section 5.2.1, defining `minSdkVersion` is necessary while `maxSdkVersion` is *not*. However, we also notice that there are still 1,079 (4.76%) apps not claiming the `targetSdkVersion` attribute, which causes their `targetSdkVersion` values be set to the corresponding `minSdkVersion` values by default.

Fortunately, the percentage of non-defined `targetSdkVersion` has significantly dropped compared to our prior analysis in 2015, from 16.54% to 4.76%. One important factor is the popularity of Android Studio in recent years,

Table 5.2: The number and percentage of non-defined DSDK attributes in our dataset.

	# Non-defined	% Non-defined
minSdkVersion	5	0.02%
targetSdkVersion	1,079	4.76%
maxSdkVersion	22,623	99.72%

which has become the de-facto IDE (integrated development environment) for Android app development. Since Android Studio by default sets and enforces the minSdkVersion and targetSdkVersion attributes, the percentage of non-defined targetSdkVersion naturally drops and we expect that this percentage will further decrease with more apps getting updated.

Finding 1-2: *Some targetSdkVersion attributes are set to outlier values.* We find that a total of 45 apps in our dataset declare their targetSdkVersion attributes as outlier values, a finding close to that in our prior analysis in 2015 when we encountered 55 such cases. There are two classes of outlier values. The first is that targetSdkVersion is set to an API level not in the range of released SDKs. At the time of our analysis, the valid API levels are from 1 to 28 (Android 9.0). However, 12 apps set their targetSdkVersion to larger than 28, namely 29, 30, and 31. In our prior analysis, we are surprised by one app with its targetSdkVersion value set to 10000. This is probably because their developers want to always target at the latest Android SDK.

The other class of outliers is that the targetSdkVersion value is set to a value lower than the minSdkVersion value. Normally, targetSdkVersion should be greater than or equal to minSdkVersion, but 33 apps have negative targetSdkVersion − minSdkVersion values. This number is almost the same as that in our prior analysis in 2015 (34 apps at that time). In particular, there was one app (com.leftover.CoinDozer) which defines its targetSdkVersion as 0, although its minSdkVersion value is 8. We believe that this class of outliers is due to developers' mistakes in declaring DSDK

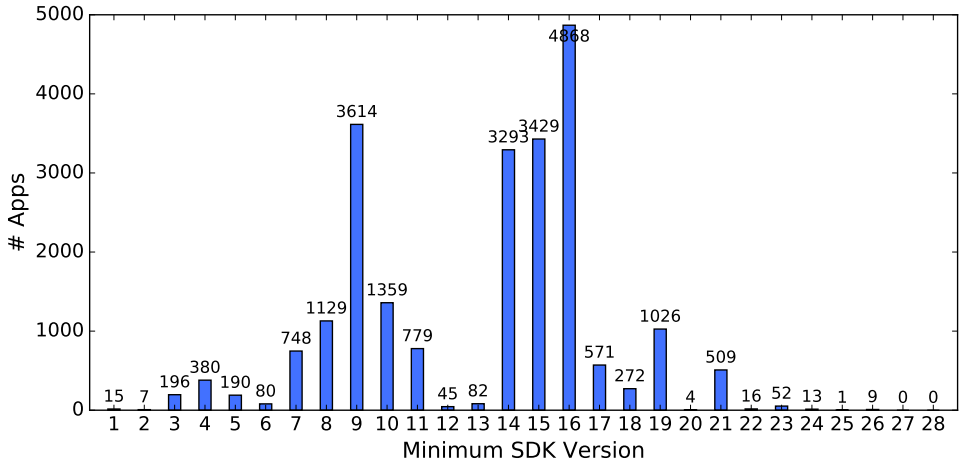


Figure 5.10: The distribution of `minSdkVersion`.

attributes.

Finding 1-3: *The minimal platform version most apps support is Android 4.1, whereas the most targeted platform version is Android 8.0. This has dramatically evolved since our last analysis in 2015.* We first study the distribution of `minSdkVersion`. According to Figure 5.10, the majority (89%) of apps have `minSdkVersion` lower than or equal to level 16 (Android 4.1), which means that they can run on nearly all (99.5%) Android devices in the market nowadays [67]. Specifically, the minimal platform version most apps support is Android 4.1 (level 16), while that in our last analysis in 2015 was only Android 2.3 (level 9). However, Android 2.3 still ranks in the second place, with 3,614 apps' `minSdkVersion` targeted at. Besides Android 4.1 and 2.3, two Android 4.0.x (level 14 and 15) platform versions are also commonly defined as apps' `minSdkVersion`.

On the other hand, Figure 5.11 plots the distribution of `targetSdkVersion`. We can see that 80% apps set their `targetSdkVersion` values to larger than or equal to level 19 (Android 4.4). In particular, the two most targeted platform versions are the most recent Android 8.0 (level 26) and 8.1 (level 27), while those in our last analysis in 2015 were Android 4.4 and 5.0. This suggests that modern apps keep better pace with the evolution of the Android operating system. Besides Android 8, Android 6.0 (level 23) and 4.4 (level 19) still hold a significant portion

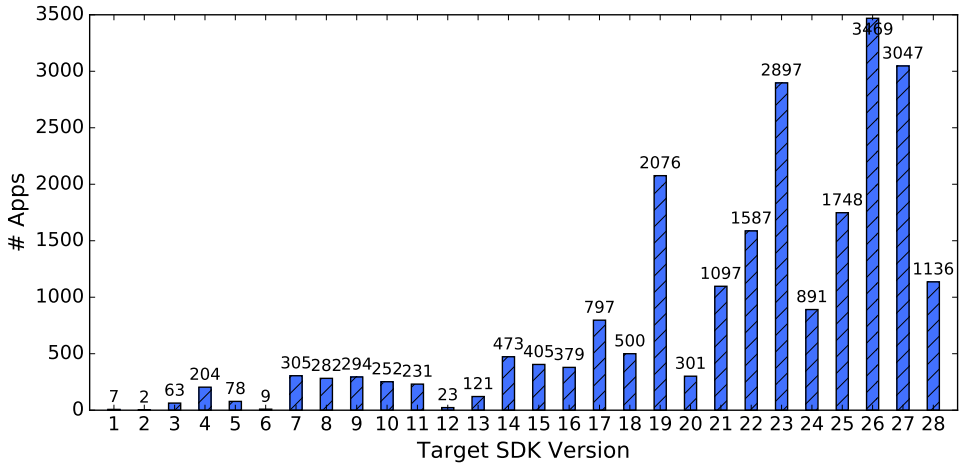


Figure 5.11: The distribution of `targetSdkVersion`.

of apps with the corresponding `targetSdkVersion` setting. Moreover, Android 7.0.x (level 24 and 25) and Android 5.0.x (level 21 and 22) also attract considerable apps being targeted at.

Finding 1-4: *The median version difference between `targetSdkVersion` and `minSdkVersion` is 9, while that of our last analysis was 8. This 11% increase indicates that Android apps nowadays need to support more Android platforms.* We define a new metric called `lagSdkVersion` to measure the version difference between `targetSdkVersion` and `minSdkVersion`, as shown in Equation 5.1.

$$\texttt{lagSdkVersion} = \texttt{targetSdkVersion} - \texttt{minSdkVersion} \quad (5.1)$$

After removing the negative `lagSdkVersion` values (i.e., outliers mentioned in Finding 1-2), we draw the CDF plot of `lagSdkVersion` in Figure 5.12. It indicates that Android apps nowadays need to support more Android platforms. This conclusion can be further supported through measuring the percentage of apps that have a `lagSdkVersion` value greater than 12. Compared to our prior analysis, this percentage has increased from 5% to 20%, which clearly shows that more and more apps nowadays support a wide range of Android platforms. On the other hand, the percentage of apps that have the same value for `targetSdkVersion`

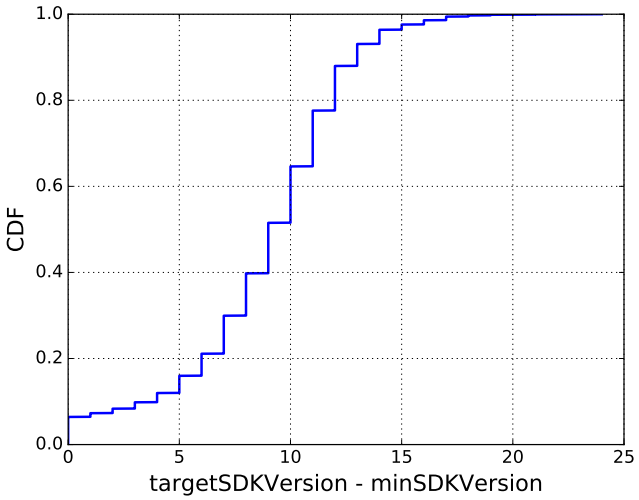


Figure 5.12: CDF plot of `lagSdkVersion`.

and `minSdkVersion` has also dropped from 20% in 2015 to 6.4% in 2018.

5.4.3 RQ2: Inconsistency Results with Compatibility Effect

In this section, we report three important findings regarding RQ2. Besides presenting compatibility results as the major finding, we also summarize the reduced false positives by our bytecode search as compared to the prior conference version, and show in detail the newly added API classes are common sources of compatibility inconsistency.

Finding 2-1: *Around 50% apps under-set the* `minSdkVersion` *value, causing them could crash when running on lower versions of Android platforms. Fortunately, only 11.3% apps could crash on Android 6.0 and above.* As explained in Section 5.3.3, the compatibility inconsistency happens if `minSdkVersion` is less than `minLevel`. In our experiments, we therefore count the number of API calls that have higher API level than `minSdkVersion` in each app, and denote it by `minOverNum`. The higher value an app's `minOverNum` is, the more likely that this app has the compatibility inconsistency.

Figure 5.13 shows the CDF plot of `minOverNum` in each app. We find that 14,363 (63.3%) apps have at least one API call that has higher API level than the corresponding `minSdkVersion`. To further increase the confidence of our anal-

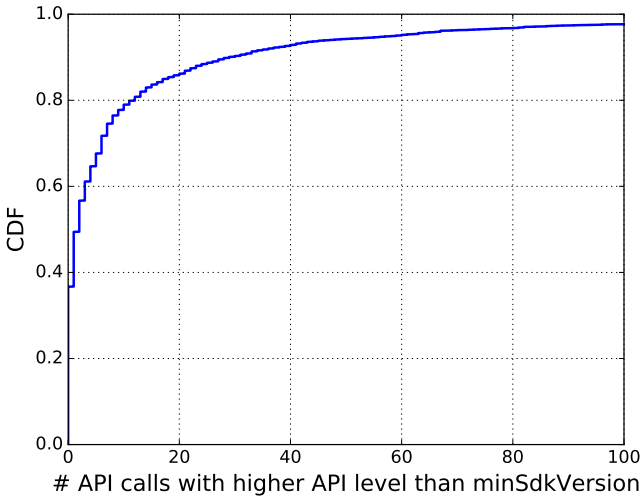


Figure 5.13: CDF plot of `minOverNum` in each app.

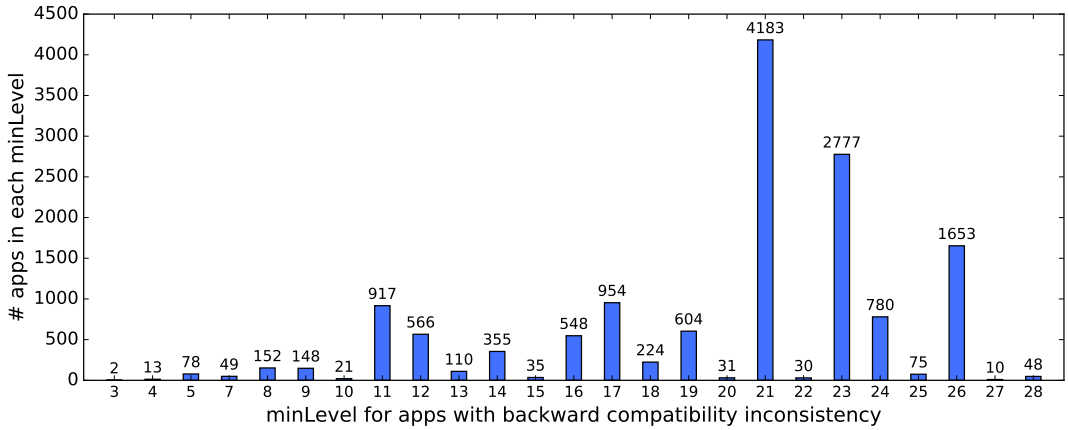


Figure 5.14: Bar chart of the number of apps in each `minLevel`.

ysis, we count that 8,019 (35.4%) apps invoke over five different API calls with higher API levels than corresponding `minSdkVersion`. Therefore, we estimate that around 50% apps could crash when running on lower versions of Android platforms because they under-set the `minSdkVersion` value. Fortunately, we find that the number of inconsistency warnings per app reported by our bytecode search is well manageable for developers — 77.8% of the 14,363 apps have fewer than 10 different inconsistent API calls. It is thus not difficult for developers to perform a one-time manual check.

Fortunately, apps with compatibility inconsistency issues could crash *only* on certain Android platforms. More specifically, they could crash only on versions of platforms between `minSdkVersion` and `minLevel`, as illustrated earlier in

Section 5.2.2. Therefore, it is necessary to study on which Android platforms those buggy apps could crash, because nowadays some lower versions of Android hold a limited market share, e.g., only 11% for Android below 5.0 [67]. As a result, even if some apps are buggy with compatibility inconsistency, they cannot trigger the crash on user phones equipped with recent versions of Android.

Since `minLevel` is the indicator for maximum versions of Android platforms a buggy app could crash on, we plot a bar chart of `minLevel` in Figure 5.14 for 14,363 app that are detected with compatibility inconsistency. We can see that only 2,566 (11.3% of 22,687) apps could crash on Android 6.0 and above (via counting apps with `minLevel` larger than 23). In other words, the majority (11,797 out of 14,363) of buggy apps cannot exhibit their incompatibility bugs on Android devices that are with over 70% market share in January 2019. Furthermore, 8,990 out of 14,363 apps could crash only on Android below 5.0, which significantly limits the consequences of their incompatibility issues.

Finding 2-2: *We find that by employing bytecode search for `SDK_INT` checking, our approach can reduce 17.3% false positives of compatibility inconsistency results.* As mentioned in Section 5.3.3, a false positive of compatibility inconsistency could appear if an API call guarded with `SDK_INT` checking is not detected. Here we measure the number of such false positives that could be excluded by the bytecode search. We find that our search of `SDK_INT` checking avoids 3,003 apps from being mistakenly marked with compatibility inconsistency. Since there are 14,363 apps (i.e., true positives) that could crash when running on lower versions of Android platforms, the percentage of reduced false positives due to bytecode search is 17.3%.

Finding 2-3: *A detailed analysis of Android APIs that incur compatibility inconsistency reveals that some API classes, such as view, webkit, and system manager related classes, are commonly misused.* We further try to understand the common sources of compatibility inconsistency by analyzing the newly added Android APIs that incur compatibility inconsistency in our dataset. We find that 6,454 (27.4% of

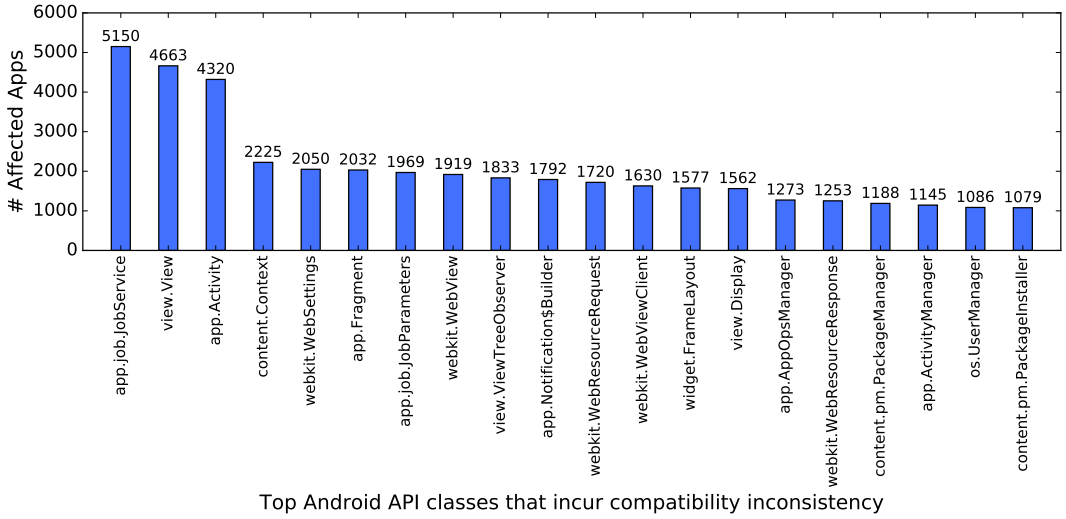


Figure 5.15: Bar chart of the top 20 Android API classes (with "`android.`" prefix omitted) that incur compatibility inconsistency in our dataset.

all 23,542) newly added APIs from 1,138 unique classes cause compatibility inconsistency in at least one app in our dataset. In particular, 232 (20.4%) API classes affect more than 100 different apps each, making them the common sources of compatibility inconsistency. Fortunately, half of API classes only affect fewer than 10 apps each, which suggests that only some portions of API classes are prone to misuses.

We thus take a closer look at the top 20 Android API classes that cause compatibility inconsistency. As shown in Figure 5.15, all of these classes affect over 1K apps each. In particular, the `JobService` class that was introduced in Android 5.0 (level 21) alone could cause compatibility inconsistency in around 5K apps. Other commonly misused API classes include those related to view (e.g., the `View`, `Activity`, `Context`, and `Fragment` classes), webkit (e.g., the `WebSettings` and `WebView` classes), and system manager (e.g., the `AppOpsManager` and `UserManager` classes). These classes nearly occupy all the top 20 misused ones.

5.4.4 RQ3: Inconsistency Results with Security Effect

In this subsection, we present a total of three findings regarding RQ3.

Table 5.3: The top five library classes that introduce `addJavascriptInterface()` API call in vulnerable apps and the number of apps affected.

Library Class	# Vulnerable Apps
Lcom/flurry/android/CatalogActivity;	41
Lcom/openfeint/internal/ui/NativeBrowser;	30
Lcom/doodlemobile/gamecenter/moregames/MoreGamesActivity;	19
Lcom/gau/go/launcherex/theme/classic/FullScreenAdWebPage;	17
Lcom/amazon/ags/html5/overlay/GameCircleUserInterface;	13

Finding 3-1: *Around 2% apps still set an outdated `targetSdkVersion` attribute when a common WebView API is vulnerable, making them exploitable by remote code execution.* As explained in Section 5.2.2, we measure inconsistency results with the security effect by analyzing each app's `addJavascriptInterface()` API call and the declared `targetSdkVersion` attribute. In our dataset, we first find that 2,791 apps invoke the `addJavascriptInterface()` API, which suggests that calling this WebView API is necessary in many apps. However, 484 of them, i.e., 2.1% of the entire dataset of 22,687 apps, still set an outdated `targetSdkVersion` attribute below level 17, making them not only exploitable on Android below 4.2 but also vulnerable on higher versions of Android platforms. This could be avoided if their `targetSdkVersion` values are updated.

Finding 3-2: *Our bytecode search of `addJavascriptInterface()` invocation helps reduce 12.2% false positives.* Recall from Section 5.3.3 that we perform bytecode search to check whether an `addJavascriptInterface()` API call is invoked by a valid class. We find that without such checking, 551 apps can be detected with vulnerable combination of `addJavascriptInterface()` and `targetSdkVersion`. In other words, our search of `addJavascriptInterface()` invocation avoids 67 (551 - 484) apps from being mistakenly marked with security inconsistency. Hence, we conclude that our bytecode search reduces 12.2% false positives in the context of `addJavascriptInterface()`.

Finding 3-3: *Around a half of the vulnerable apps invoke* `addJavascriptInterface()` *because of their embedded third-party libraries.* Our approach can also determine whether the `addJavascriptInterface()` API is invoked by app's own code or embedded by a third-party library. It turns out that 214 (44.2%) of 484 vulnerable apps invoke `addJavascriptInterface()` because of their embedded third-party libraries. In particular, five libraries affect at least 10 vulnerable apps each. Table 5.3 lists their class names and the number of apps affected. We can see that the popular Yahoo Flurry SDK [32] and OpenFeint Game SDK [45] cause some apps with outdated `targetSdkVersion` to become vulnerable.

This finding gives two implications. First, developers must check whether a third-party library invokes some vulnerable APIs before embedding it into apps. Second, library producers also need to ensure certain dangerous APIs are invoked only in safe versions of Android platforms, because a library can be used in any app with all kinds of `targetSdkVersion` values.

5.4.5 RQ4: Performance Metrics of Our Approach

In this section, we evaluate performance metrics of our approach, a lightweight version of BackDroid, to answer RQ4.

Finding 4-1: *Our approach achieves good scalability with an average time of 5.39s and the analysis time of 90% apps in less than 10 seconds. This makes our approach suitable for online vetting.* In Figure 5.16, we present CDF plot of the amount of time required for our approach to analyze each app. We can see that more than 50% apps can be analyzed in less than five seconds each, with the median time of 4.75s. The average analysis time of all the 22,687 apps is only 5.39s. These results indicate that our approach achieves good scalability, therefore suitable for online vetting. Therefore, app markets can deploy our approach to timely notify developers the `DSDK` inconsistency in their apps.

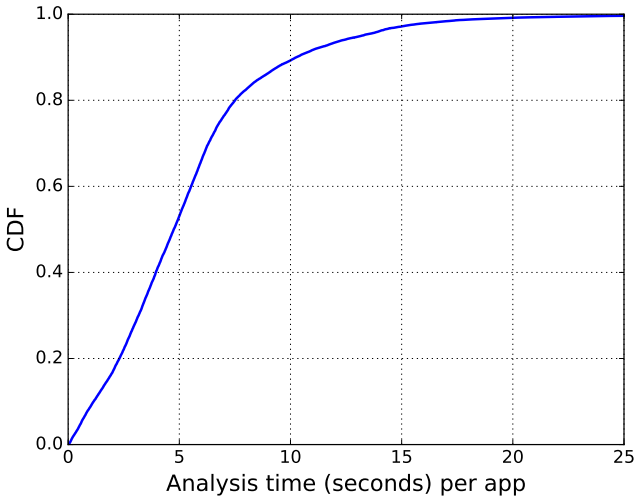


Figure 5.16: CDF plot of the amount of time required for our lightweight BackDroid to analyze each app.

In contrast, dataflow-based approaches [100] [93] suffer from the scalability problem. Specifically, CiD [100] failed to analyze 387 apps (out of a dataset of 2,000 apps) due to timeouts and bugs. This 19.4% timeout or failure rate makes it infeasible for online vetting, let alone performance statistics were also not clear for those successfully analyzed. On the other hand, IctApiFinder [93] takes 3 minutes and 45 seconds to analyze only an app of 8MB (the app is available via historical versions on `https://f-droid.org/en/packages/com.nextcloud.client/`), a size much smaller than the average size (25MB) of our dataset. This suggests that IctApiFinder is impractical to perform online vetting of a modern app dataset from Google Play (all apps evaluated by IctApiFinder were open-source apps from the F-Droid website).

Finding 4-2: *A further correlation analysis between analysis time and app size shows that the performance of our approach is approximately in a linear relationship with DEX file size of the app.* We further statistically demonstrate that the performance of our approach is always under control regardless of the app size. This can be evaluated by performing a correlation analysis between analysis time and app size. In Figure 5.17, we draw a scatter plot of the relationship between analysis time and the size of DEX file of the app (APK file contains both bytecode

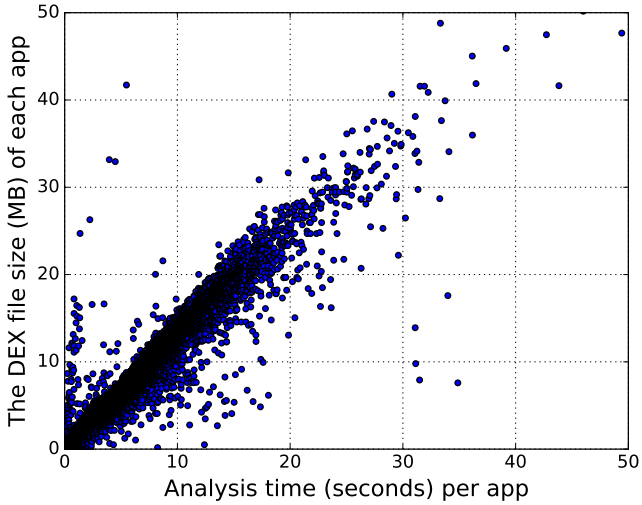


Figure 5.17: Scatter plot of the relationship between analysis time and DEX size.

and resource files while DEX file is only for bytecode). According to this figure, the analysis time and DEX file size are approximately in a linear relationship at the rate of around 30 seconds for a 40MB DEX file (note that we count the file size of multiple DEX files if any). There are some outliers of small apps with more analysis time (e.g., five apps under 20MB exceeding 30s), which is largely because these apps involve much more vulnerable API calls to search. On the other hand, the outliers of large apps with less analysis time is due to unused third-party libraries embedded. Overall, the linear relationship between analysis time and app size indicates that our approach can achieve good performance even with large apps.

5.5 Threats To Validity

In this section, we discuss some threats to the validity of our study.

First, same as typical Android static analysis, our approach does not handle Java reflection, dynamic code loading, native code, and complicated code obfuscation. However, some apps may employ these mechanisms to access certain Android APIs. If a such API call has inconsistency issues, a false negative would appear. Since these code protection mechanisms are usually used in malware, our statistical results of popular apps will be less affected and we will consider these mechanisms to our

future work.

Second, although our bytecode search in Section 5.3.3 has minimized false positives caused by `VERSION.SDK_INT` checking and uninvoked third-party libraries, it is theoretically less accurate than dataflow-based approaches. Therefore, in our deployment model, developers are required to manually check and correct inconsistency reported by our approach. Fortunately, as evidenced in Section 5.4.3, around 80% apps are reported with fewer than 10 inconsistent API calls each, which is well manageable for developers to perform a one-time manual check.

Third, the consistency detection in this chapter focuses on changed APIs, but there are also added and removed Java/Android fields during the SDK evolution. To build the mapping between fields and SDK versions, we found that we can leverage the same document analysis method in Section 5.3.2, because the `api-versions.xml` file also records `added`, `removed`, and `deprecated` fields in all Android classes. By inputting this mapping to our app analysis, we can extend our consistency detection to evolved Android fields as well in our future work.

5.6 Summary

In this chapter, we conducted a systematic study of declared SDK versions in Android apps, a modern software mechanism that has received little attention. We measured the current practice of declared SDK versions or `DSDK` versions in a large set of 22,687 modern apps, and the inconsistency between `DSDK` versions and their host apps' API calls. To facilitate the analysis that can be readily deployed by app markets for online vetting, we proposed a robust and scalable approach that operates on the Android bytecode level and employs the lightweight bytecode search in BackDroid for app analysis. We have obtained some interesting new findings, including (i) 4.76% apps do not claim the targeted `DSDK` versions, although this percentage has significantly dropped over recent three years, (ii) around 50% apps

under-set the minimum DSDK versions and could incur runtime crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above, and (iii) around 2% apps, due to under-claiming the targeted DSDK versions, are potentially exploitable by remote code execution, and a half of them invoke the vulnerable API via embedded third-party libraries.

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

In this dissertation, we made a first attempt to explore a novel on-demand Android static analysis that does not generate a whole-app call graph but creatively leverages bytecode search to guide inter-procedural analysis on the fly or just in time. We developed such on-the-fly static analysis into a novel tool, called BackDroid, for efficient and effective targeted security vetting of Android apps. Notably, BackDroid employed a novel backward search technique to search over Java polymorphism, threads, implicit callback flows, and Android inter-component communication. We further explored how the core technique of on-the-fly static analysis in BackDroid can enable different vulnerability studies and their corresponding new findings. To this end, we performed three representative vulnerability studies as follows:

- First, we applied BackDroid to detect crypto and SSL/TLS misconfigurations in modern Android apps. We also used this study as an evaluation of BackDroid and compared it with the state-of-the-art Amandroid tool. The results showed that BackDroid achieved a much better performance than Amandroid, around ten times faster on average, and at the same time, maintained similar detection effectiveness as Amandroid for the apps detected by Amandroid. Moreover, BackDroid discovered 18 additional vulnerable apps (out of the

144 apps with the targeted sink APIs) that were missed by Amandroid.

- Second, we explored how BackDroid can facilitate a systematic security study of open ports in Android apps. To this end, we first discovered open-port apps using crowdsourcing, and then enhanced BackDroid to identify insecure open ports and open-port SDKs in the discovered open-port apps. Specifically, the crowdsourcing allowed us to observe the actual execution of open ports in 925 popular apps and 725 built-in system apps, and the enhanced BackDroid diagnosed that 61.8% of the open-port apps are solely due to embedded SDKs and 20.7% suffer from insecure API usages. We further performed three security assessments to reveal five vulnerability patterns in open ports of popular apps, to measure the feasibility of remote open-port attacks, and to demonstrate the effectiveness of denial-of-service attacks against mobile open ports.

- Third, we customized a lightweight version of BackDroid that operated on the original bytecode level and leveraged lightweight bytecode search to measure the inconsistency between declared SDK versions and their API calls in modern Android apps. By focusing on the control-flow information of searched sink APIs, our lightweight BackDroid preserved a scalability suitable for online vetting. We then employed this custom BackDroid to analyze the SDK-API inconsistency for 22,687 modern popular apps, and found that (i) $\sim$50% apps under-set the minimum DSDK versions and could incur runtime crashes, but fortunately, only 11.3% apps could crash on Android 6.0 and above; and (ii) $\sim$2% apps, due to under-claiming the targeted DSDK versions, are potentially exploitable by remote code execution.

To conclude, this dissertation made this core contribution: *On-the-fly Android static analysis guided by bytecode search can efficiently and effectively analyze the security of modern apps. It enables us to perform vulnerability studies with different kinds of sink analysis requirements, and to obtain new findings on crypto and SSL/TLS misconfigurations, insecure open ports, and SDK-API inconsistency.*

6.2 Future Research Directions

Given this dissertation, it will be interesting and valuable to further work on the following three major research directions:

1. *Searching over Java reflection and native code*. As discussed in Section 3.5, BackDroid currently does not handle Java reflection and native code. Although it is possible to integrate existing dedicated works (e.g., DroidRA [99] and JN-SAF [131]) into BackDroid, a desirable and long-term approach is to propose new backtracking capability that can be still performed in an on-the-fly manner.

2. *Hybrid analysis with both static and dynamic techniques*. BackDroid also does not support dynamically loaded and packed code, a common limitation in typical Android static analysis. Addressing them will need the support of dynamic analysis. In Chapter 4, we have shown that by combining BackDroid and crowdsourcing-based dynamic analysis, we can build an effective open-port analysis pipeline. It will be interesting to see more hybrid analysis using both static and dynamic techniques.

3. *Exploring more applications of BackDroid*. To realize the full potentials of BackDroid, we will open source BackDroid to the community and add more analysis capabilities together with them. Such a diverse BackDroid platform can help explore more security problems, e.g., not only limited to vulnerability discovery but also for malware analysis. Moreover, researchers in the software engineering community can also develop their custom tools on top of BackDroid, as we have demonstrated in Chapter 5.

Bibliography

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

[2] aapt: Android Asset Packaging Tool. http://elinux.org/Android_aapt.

[3] AlarmManager. https://developer.android.com/reference/android/app/AlarmManager.html.

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

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

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

[7] Android Logcat. https://developer.android.com/reference/android/util/Log.html.

[8] AndroidVideoCache. https://github.com/danikula/AndroidVideoCache.

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

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

[11] apktool. http://ibotpeaches.github.io/Apktool/.

[12] App security best practices - Android developers. https://developer.android.com/topic/security/best-practices.

[13] AssignStmt (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/AssignStmt.html.

[14] BinopExpr (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/BinopExpr.html.

[15] BluetoothSocket. https://developer.android.com/reference/android/bluetooth/BluetoothSocket.html.

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

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

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

[19] CyberGarage UPnP SDK. https://github.com/cybergarage/cybergarage-upnp.

[20] DefinitionStmt (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/DefinitionStmt.html.

[21] Detecting remote code execution vulnerabilities in Android apps. https://sites.google.com/site/androidrce/.

[22] dex2jar. https://github.com/pxb1988/dex2jar/.

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

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

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

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

[27] Getui Push SDK. http://docs.getui.com/mobile/android/overview/.

[28] How to fix Fragment Injection vulnerability. https://support.google.com/faqs/answer/7188427.

[29] hping3. http://linux.die.net/man/8/hping3.

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

[31] IDA Support: Download Center. https://www.hex-rays.com/products/ida/support/download.shtml.

[32] Integrate Flurry SDK for Android. https://developer.yahoo.com/flurry/docs/integrateflurry/android/.

[33] InvokeExpr (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/InvokeExpr.html.

[34] InvokeStmt (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/InvokeStmt.html.

[35] JEB Decompiler for Android. https://www.pnfsoftware.com/jeb/android.

[36] JS-Binding-Over-HTTP Vulnerability and JavaScript Sidedoor: Security Risks Affecting Billions of Android App Downloads. https://www.fireeye.com/blog/threat-research/2014/01/js-binding-over-http-vulnerability-and-javascript-sidedoor.html.

[37] LG Smartshare. http://www.lg.com/support/smart-share.

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

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

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

[41] Multiple APK support - Android Developers. https://developer.android.com/google/play/publishing/multiple-apks.

[42] NewArrayExpr (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/NewArrayExpr.html.

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

[44] Nmap: the network mapper. https://nmap.org/.

[45] Openfeint is the largest mobile social gaming network in the world. http://www.openfeint.com/.

[46] PhoneGap SDK. https://phonegap.com/.

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

[48] ReturnStmt (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/jimple/ReturnStmt.html.

[49] Samsung Accessory SDK. http://developer.samsung.com/galaxy/accessory.

[50] Security tips - Android developers. https://developer.android.com/training/articles/security-tips.

[51] Selenium - web browser automation. http://docs.seleniumhq.org/.

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

[53] smali/baksmali. https://bitbucket.org/JesusFreke/smali/overview.

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

[55] [soot-list] flowdroid: call graph doesn't look context sensitive. https://mailman.cs.mcgill.ca/pipermail/soot-list/2016-March/008410.html.

[56] 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.

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

[58] Tencent XG Push SDK. http://docs.developer.qq.com/xg/.

[59] The AndroidManifest.xml File. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

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

[61] Unit (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/Unit.html.

[62] The uses-sdk manifest element. http://developer.android.com/guide/topics/manifest/uses-sdk-element.html.

[63] What is Soot? http://sable.github.io/soot/.

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

[65] 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.

[66] M. Almeida, M. Bilal, J. Blackburn, and K. Papagiannaki. An empirical study of Android alarm usage for application scheduling. In *Proc. Springer PAM*, 2016.

[67] Android. Dashboards. https://developer.android.com/about/dashboards/.

[68] Android. Platform codenames, versions, and API levels. https://source.android.com/source/build-numbers.html.

[69] 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.

[70] K. Au, Y. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proc. ACM CCS*, 2012.

[71] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. ACM ICSE*, 2015.

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

[73] 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.

[74] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proc. ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012.

[75] 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.

[76] G. Brito, A. Hora, M. T. Valente, and R. Robbes. Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In *Proc. IEEE SANER*, 2016.

[77] 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.

[78] P. P. Chan, L. C. Hui, and S.-M. Yiu. A Privilege Escalation Vulnerability Checking System for Android Applications. In *Proc. IEEE International Conference on Communication Technology*, 2011.

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

[80] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. Springer Information Security Conference (ISC)*, 2010.

[81] J. Drake. On the WebView addJavascriptInterface saga. `http://www.droidsec.org/news/2014/02/26/on-the-webview-addjsif-saga.html`, 2014.

[82] 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.

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

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

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

[86] S. Fahl, M. Harbach, T. Muders, L. Baumgrtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proc. ACM CCS*, 2012.

[87] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. ACM CCS*, 2011.

[88] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards detecting logic bombs in Android applications. In *Proc. IEEE Symposium on Security and Privacy*, 2016.

[89] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proc. ACM CCS*, 2012.

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

[91] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proc. ISOC NDSS*, 2012.

[92] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proc. ACM MobiSys*, 2012.

[93] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proc. ACM ASE*, 2018.

[94] 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.

[95] 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.

[96] M. Johns, S. Lekies, and B. Stock. ZMap: Fast Internet-wide scanning and its security applications. In *Proc. Usenix Security*, 2013.

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

[98] 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 *Proc. ACM ICSE*, 2015.

[99] 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.

[100] L. Li, T. F. Bissyandé, H. Wang, and J. Klein. CiD: Automating the detection of API-related compatibility issues in Android apps. In *Proc. ACM ISSTA*, 2018.

[101] L. Li, T. F. Bissyand, Y. L. Traon, and J. Klein. Accessing inaccessible Android APIs: An empirical study. In *Proc. IEEE ICSME*, 2016.

[102] 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.

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

[104] M. Linares-Vsquez, G. Bavota, C. Bernal-Crdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proc. ACM FSE*, 2013.

[105] 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.

[106] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. ACM CCS*, 2012.

[107] S. Ma, D. Lo, T. Li, and R. H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proc. ACM AsiaCCS*, 2016.

[108] 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.

[109] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proc. IEEE ICSM*, 2013.

[110] P. Mutchler, Y. Safaei, A. Doupe, and J. Mitchell. Target fragmentation in Android apps. In *Proc. IEEE Mobile Security Technologies (MoST)*, 2016.

[111] 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.

[112] 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.

[113] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *Proc. ACM FSE*, 2012.

[114] 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.

[115] 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.

[116] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps. In *Proc. ISOC NDSS*, 2017.

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

[118] 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.

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

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[126] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proc. ISOC NDSS*, 2014.

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

[128] 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.

[129] 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.

[130] 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.

[131] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang. JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *Proc. ACM CCS*, 2018.

[132] 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.

[133] 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. *ACM Transactions on Privacy and Security (TOPS)*, Volume 21, Issue 3, 2018.

[134] L. Wei, Y. Liu, and S.-C. Cheung. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proc. ACM ASE*, 2016.

[135] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proc. ACM ACSAC*, 2012.

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

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

[138] D. Wu and R. K. C. Chang. Analyzing Android Browser Apps for file:// Vulnerabilities. In *Proc. Springer Information Security Conference (ISC)*, 2014.

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

[140] 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.

[141] D. Wu, Y. Cheng, D. Gao, Y. Li, and R. H. Deng. SCLib: A practical and lightweight defense against component hijacking in Android applications. In *Proc. ACM CODASPY*, 2018.

[142] D. Wu, D. Gao, R. K. C. Chang, E. He, E. K. T. Cheng, and R. H. Deng. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *Proc. ISOC NDSS*, 2019.

[143] 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. Springer International Conference on Wireless Algorithms, Systems, and Applications (WASA)*, 2017.

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

[145] 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.

[146] K. Xu, Y. Li, R. H. Deng, and K. Chen. DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In *Proc. IEEE EuroS&P*, 2018.

[147] 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.

[148] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in Android hybrid apps. In *Proc. Springer RAID*, 2017.

[149] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. ACM ICSE*, 2015.

[150] 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.

[151] M. Zhang and H. Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. ISOC NDSS*, 2014.

[152] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. In *Proc. ACM CODASPY*, 2015.

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

[154] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in Android applications. In *Proc. ISOC NDSS*, 2013.

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

[156] 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.

[157] C. Zuo, J. Wu, and S. Guo. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In *Proc. ACM AsiaCCS*, 2015.