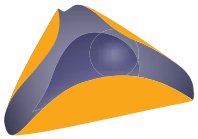


MASTER IN COMPUTER SCIENCE

Web Communication Analysis of Android Applications

Master Thesis

Marc-Andrea Tarnutzer

from

Basel BS, Switzerland

Faculty of Science
at the University of Bern

26 April 2019

Prof. Dr. Oscar Nierstrasz

Pascal Gadient

Software Composition Group
Institute for Computer Science

University of Bern, Switzerland

b
UNIVERSITÄT
BERN

UNIVERSITÉ DE
NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

The use of internet communication channels in Android applications has become omnipresent; mobile apps exchange various kinds of information and it is not always clear which data and for what purpose. Privacy leaks are on the rise and unlock the door for even more severe threats such as data spoofing and hijacking. To gather an understanding of the current web API use in Android apps, we have investigated nine major web communication frameworks, and searched for their design patterns in 413 open-source projects available on *GitHub* and 834 Android apps published in *Google*'s *Play* store. With this knowledge, we recovered JSON data schemes understood by their public web API endpoints, considering type information and used parameter values. Based on these data schemes we built and verified request URLs that leverage potentially sensitive data, and moreover, we explored the differences in web API use between open-source and closed-source apps. Finally, we reason about the collected HTTP response header information received from the various API endpoints. We present a static code analysis tool on top of the *JADX* DEX decompiler and the *JavaParser* framework, which dissects and assesses the decompiled byte code of Android apps. With the help of this tool we successfully analyzed web APIs in 36% of all tested apps, out of which we were able to rebuild 2 154 API endpoint URLs. Furthermore, we manually validated the tool's results for 25 open-source and for 25 closed-source apps. More than 90% of the endpoints did successfully reply to our requests. We found that many potential data leaks emerge from web APIs that have been implemented exclusively for individual apps and thus are out of focus for many security investigators due to their scarce deployment and the implementers' reluctance to provide any public documentation.

Contents

1 Introduction

The changes introduced by recent advances in cellular network technology (3G/4G/5G) have heavily influenced the architecture of mobile apps. Whereas ten years ago apps mainly focused on local operations and rarely consumed any services from the internet, apps now shift major functionality to remote servers due to the hosts' always connected policy, ultimately leading to heavily increased wireless data traffic.[1] Even more important, instead of processing information on in-house servers a trend has emerged that apps offload information to external cloud services, which have benefitted across the board from massive growth over the past years.[2]

However, this change of paradigm does not only save computational resources on mobile handsets, it also introduces a never before seen complexity in implementing and maintaining such services due to the large software stacks involved in client-server architectures. Such architectures generally are multi-tier, geographically distributed, and thus collaboratively maintained by globally scattered teams in which developers cooperate with other team members. Moreover, the use of sensitive sensory data is omnipresent on smart mobile devices, and therefore, developers are in need of common secure programming practice guidelines.

[1] https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html

[2] https://www.salesforce.com/content/dam/web/en_us/www/academic-alliance/datasheets/IDC-salesforce-economy-study-2016.pdf

The Android platform suffers in particular from security hazards due to the industry-lead proliferation of, and the unrestricted access to apps. The Android *APK* file format not only provides access to each app's Java byte code offering deep insights into the inner workings of an app, the Android ecosystem also dominates the smartphone operating systems with a market share of about 88% and just its official app store contains more than 2.6 million apps.[3,4] Low annual developer fees and the use of the Java programming language further attract first time programmers with only little background in computer science. As a result, these communication-oriented apps represent an irresistible target for attackers and present a serious threat to their users, as revealed by recent reports of well-known publicly available security incident databases.[5]

As history has shown, attackers prefer the path with least resistance and since web APIs are seldom sufficiently protected they represent a severe threat to billions of mobile app users, paving the way for many new problems related to data security, *e.g.*, cloud data theft and leakage from public servers. This threat gains even more importance by considering the fact that many mobile devices are taking over prior desktop computer functionality, such as health care monitoring, e-banking, and social networking.

Researchers are aware of this major threat. In fact, Android security has been progressively studied from various aspects over the previous years. Countless security tools [11], operating system changes [4, 12], and guidelines have been proposed [9, 20]. Nevertheless, many entry-level developers seem to be less concentrated on the programming task at hand while working in comprehensive IDEs, *e.g.*, Android's official IDE called *Android Studio*, instead relying on the tool and framework support even for rather simple tasks [10]. In addition, many developers do not feel personally responsible for the implementation of software security, despite being aware of its importance and having a general understanding about the topic [23]. As a result, user data-related incidents on web applications remain one of the major threats to the security of mobile apps.[6]

We continue to promote the automated adoption of secure programming practices and show in this thesis nine major web communication frameworks along with their characteristics. Moreover, we evaluate web communication in open-source and closed-source apps, and we discuss the found issues.

We present a static source code analysis framework on top of the Android DEX decompiler *JADX*[7] and the Java source code parsing framework *JavaParser*[8] to establish API specifications, *i.e.*, data schemes, based on the apps' source code. With the help of this tool we were able to reconstruct 2 154 different request URLs.

We address the following four research questions:

- **RQ**$_1$: *Which are the prevalent web communication frameworks used in mobile apps?* We assessed the communication facilities used in 160 open-source projects from *F-Droid* hosted on *GitHub* by

[3] `https://www.statista.com/statistics/266136`
[4] `https://www.statista.com/statistics/266210`
[5] `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android`
[6] `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf`
[7] `https://github.com/skylot/jadx`
[8] `https://javaparser.org/`

manually reviewing the code blocks responsible for web communication and found nine different used web communication frameworks. Prioritized by their popularity we automated the detection of sink methods for the major frameworks. The corresponding tool allowed us to analyze 413 open-source projects from *GitHub* and 834 (partially) decompiled closed-source apps available on the *Google Play* store. We found in our analysis that web communication channels are primarily built on *java.net* standard library classes such as `java.net.URLConnection`, `java.net.Socket`, and `java.net.HttpURLConnection`. Moreover, the most used third party libraries were *OkHttp* and *Retrofit*.

- **RQ**$_2$: *What data do mobile apps transmit through web communication channels?* In addition, our tool extracts relevant code snippets, URLs, and JSON schemes from the applications' source code. We applied that tool to our repository of open/closed-source apps and found that the most used data types in JSON schemes for both, open-source and closed-source applications, are the type `STRING`, followed by `NUMBER`, `BOOLEAN`, and `NULL`. Furthermore, the most commonly transmitted JSON and URL query keys were identifiers (*e.g.*, for resources, sessions), categories, application settings, and (list) boundaries. We found the data transmitted through web communication channels to be very diverse, ranging from very specific (potentially sensitive) resource requests to commands and selection criteria requesting database information from servers.

- **RQ**$_3$: *What are the differences between open-source and closed-source apps in regard to web communication?* We compared our findings from the analysis of open-source and closed-source applications. We found only minor differences when comparing the choice of web communication frameworks, however, we encountered substantially more advertisement services in closed-source applications. More interestingly, we discovered that the use of the secure HTTPS protocol is much more prevalent in open-source projects than closed-source applications, in which the majority still relies on the insecure HTTP protocol. At the same time, the code responsible for web communication functionality in closed-source applications is spread across more files and the endpoints tend to have more complicated endpoint paths. Obvious security flaws such as hard-coded API keys, email addresses, and outdated server configurations have been found in both open-source and closed-source applications.

- **RQ**$_4$: *What configurations apply to API endpoint servers found in the wild?* Finally, the tool is able to validate the collected URL and JSON scheme data by issuing requests to web servers using different HTTP methods. During a manual analysis of the results from our tool we found multiple security issues concerning web server configurations including unnecessary disclosure of server configurations, outdated web servers and programming language support with known security vulnerabilities, leaking of the server's internal error messages, issuing shell commands inside the request body, and (private) APIs with no authentication/authorization mechanisms at all.

In summary, this work investigates the use of, and the potential risk caused by web APIs to sensitize API developers for taking care of sensitive user data to reduce the corresponding attack vectors. We believe that this study is a first step towards better tool support based on our found API use patterns, which will

ultimately improve the state of information security[9] for every mobile app user in the Android ecosystem.

1.1 Contributions

Most existing network communication analysis tools are technology-driven rather than data-driven which inevitably leads to a subtle shift in the result's expressiveness. In this work we want to leverage data-driven approaches to aggregate an overview of the current state of web communication. These approaches favorably support developers in fixing issues at the architectural level, rather than the app level, due to the nature of the feedback, which is based on high-level API specifications. This thesis provides (i) an overview of major web communication frameworks, (ii) insights into the use of web communication in mobile apps, (iii) information regarding the communication preferences of open-source and closed-source apps, (iv) configuration details and culprits of API endpoints, and, (v) access to the open-sourced tool that we developed for the web API extraction of Android apps.

1.2 Outline

The remainder of this thesis is organized as follows. We provide a brief overview of the state-of-the-art in web communication analysis in chapter 2, followed by the necessary background, including the Android ecosystem and its communication facilities in chapter 3. We reveal our methodology and implementation details in chapter 4 and chapter 5, accordingly. Our empirical study is presented in chapter 6, and the corresponding threats to validity are shown in chapter 7. We provide an outlook in chapter 8, before concluding the thesis in chapter 9.

[9] Information security covers all aspects of data confidentiality, integrity, and availability.

2

Related Work

In this chapter we shed light on the different aspects considered in prior work with respect to the use of third party web services. We first present significant work targeting the field of web communication, *i.e.*, covering different analysis techniques, and server response evaluations. Afterwards, we highlight some publications investigating the use of hard-coded credentials in apps' source code, before we briefly focus on soft factors.

2.1 Web communication

Web communication analysis techniques can be roughly split into three different categories: code-based analyses, network traffic-based analyses, and variations of both. Whereas code-based analysis techniques are very often performed in a static manner, *e.g.*, by statically resolving method and field names in the source code, network traffic-based analysis mechanisms show rather a dynamic characteristic as researchers generally try to analyse network requests initiated by the apps at run time. Ultimately, hybrid variations of both exist that rely on static and dynamic analyses.

2.1.1 Code-based analyses

One major interest in code-based analyses belongs to data flow reasoning, which assesses the prevalent communication channels used in mobile apps. Data flow analysis tries to (re)construct data flows from sensitive data sources to data sinks, *e.g.*, from locally stored private information to network sockets, or within inter-app communication in general. Various methods based on pre-processed source and sink lists and deep (machine) learning have been proposed.

For instance, Arzt *et al.* achieved with *FlowDroid* remarkable analysis results by performing a reachability analysis starting from sources marked in pre-defined lists [2]. Whenever the tool finds a valid source in the code it conducts a reachability analysis based on the apps' augmented inter-procedural control-flow graph. If the reachability analysis can reach a sink from such a source it considers the combination of the source/sink method pair as a potential data leak that is reported. Zhu *et al.* augmented *FlowDroid* by adding deep learning algorithms to leverage a more precise classification of the reported data flows. According to them their tool *DeepFlow* achieves a superior performance in distinguishing between data flows of benign and malign apps [26]. Besides tools based on the Java optimization framework *Soot*[1], Wei *et al.* implemented *Amandroid*, which is able to discover data flows across Android apps' components, and which greatly increases the tool's performance [22]. Unlike our work these tools do not focus specifically on the actual data structures sent over network sinks, but only focus on specific low-level method call traces that lack high-level context. Furthermore, they only consider data flows within apps on the device but neglect the server at the end of the line. Other more efficient approaches are based on decompiled byte code of Android apps where, amongst others, various security code smells have been discussed [8, 9]. In summary, while prior works on security code smells addressed only the Android app security, we are going with this study one step further: we consider the transmitted data and the corresponding server infrastructures that respond to mobile app requests.

Another major interest can be found in the static analysis of data transmission channel configurations. Fahl *et al.* and Sounthiraraj *et al.* both verified the protection of *Hypertext Transfer Protocol Secure* (HTTPS) data transmission channels by validating the correctness of the apps' SSL implementations and found that 6% and 8% of all tested apps were vulnerable, as described in their respective papers [7, 18]. Fahl *et al.*, for instance, conducted a manual inspection of 100 apps and discovered that 41 of those apps were vulnerable to man-in-the-middle attacks. These results underline the importance of our work as it is important to be aware of the data sent to servers, as in many cases the data transmission could be easily intercepted by attackers.

The final interest can be found in the static analysis of apps' network libraries and their use in Android apps. For example, Stevens *et al.* were especially interested in libraries providing ad services and found excessive app permission requests as well as the use of identifiers and sensitive user data [19]. They also discovered that this data can then be exploited to track users across ad providers by analyzing the network traffic.

[1] `https://github.com/Sable/soot`

2.1.2 Network traffic-based analyses

Analyses based on network traffic most frequently rely on observing the apps' network communication while it is being executed in a virtualized environment. This kind of dynamic analysis has been very prominently used for malware detection. For instance, Arora *et al.* closely monitored several network traffic properties, *e.g.*, average packet size, time interval between packets sent, and relative amount of TCP traffic to identify 13 different Android malware types running on mobile devices [1]. They could correctly determine the malware based on the observed network traffic in more than 90% of their studies. One step further was taken by Conti *et al.* who investigated the identifiability of user actions in apps running on Android mobile devices, using encrypted network communication, by intercepting the network traffic directly on the connected wireless routers [5]. Even though the traffic remained encrypted, they could still identify most user actions with an accuracy of better than 95%. Following the trend of offloading code to the cloud, Shen *et al.* implemented the malware detection service *DroidDetector* on top of a virtual private network (VPN) service that closely monitors all connection requests made by Android apps and classifies the traffic based on the Support-Vector Machine (SVM) machine learning algorithm [16]. As a result of offloading the analysis to the cloud, the process is sparing the resources of the mobile devices while conducting a comprehensive server-side analysis with a detection confidence of 95.68%. According to such research fingerprinting of app traffic is not an issue at all, but the opposite is true for the reverse: cloaking user actions is more of a challenge.

In contrast to the overwhelming number of papers proposing fingerprinting mechanisms we used dynamic analysis techniques primarily for evaluating the server responses. We did not execute the app as such but tested the apps' reconstructed API URL calls and payloads in a distinct environment, which led to greater control over the request and at the same time simplified our workflow. We further found for network traffic-based analysis techniques a similar problem to code-based analysis methods: researchers heavily focus on low-level app communication and carelessly neglect relevant server configurations.

2.1.3 Combined approaches

Malware detection using code- and network-level measures has been implemented in hybrid approaches in order to maximize the individual gains achieved by static and dynamic analyses. One typical representative of this category is the tool called *MARVIN* by Lindorfer *et al.*, which utilizes machine learning techniques combined with static and dynamic code analysis to classify apps into several categories ranging from benign to malicious [13]. This tool not only relies on static features, *e.g.*, method calls and manifest meta information, it also leverages runtime network traffic monitoring. This combined approach performed exceptionally well. They evaluated the classification capabilities of their tool on a large Android malware dataset of over 135 000 Android apps and 15 000 malware samples with a success rate of 98.24% and less than 0.04% false positives.

Another approach targeting specifically the communication between mobile apps and the web API servers

has been investigated by Mendoza *et al.* [15]. They tried to reconstruct with their tool *WARDroid* the data schemes used for web communication and generated web API URLs with random field values based on the extracted type information. Their goal was to find inconsistencies in input validation logic between apps and their respective web API services. The gathered results were intriguing. The evaluation of their system uncovered problematic validation logic in APIs used by over 40% of the tested 10 000 popular free apps from the *Google Play* store and also revealed unencrypted client-server communication over the *Hypertext Transfer Protocol* (HTTP) in 1 743 apps. They validated their results by sending invalid requests to selected APIs used in 1 000 apps potentially vulnerable to API hijacking and discovered that over 90% of them were indeed affected.

In comparison to our work, we collect more thorough information such as provided value type samples to generate more reasonable request strings, and the complete server response headers. Moreover, we augment our dataset with *Google Play* store meta data, and perform analyses on open-source and closed-source apps. This allows us to draw further and more in-depth conclusions regarding the use and prevalence of web communication and its security in Android apps.

Besides these two use cases, hybrid approaches are commonly used to confine static analysis results, because static analyses are prone to false-positives due to the high complexity of code analyses and their indispensable need for simplified assumptions.

2.2 Credentials

Credentials extracted from source code are likewise of large interest for security researchers, because they potentially allow adversaries to gain full control over the apps' server-side infrastructure and its enclosed data store.

Zhou *et al.* harvested cloud service credentials such as email and passwords of developers with their tool *CredMiner* from more than 36 500 apps from various Android markets [25]. More specifically, they were interested in the usage of free email services and Amazon AWS by Android app developers and alarmingly found that more than every second app using such a service leaked the developers' credentials in the apps' source code. Making matters worse, they discovered that more than 77% of those collected credentials could still be used to access the developers' accounts. The work of Zhou *et al.* shows the massive threat such security issues pose, as many of those credentials cannot be easily replaced without temporarily abating the experience of millions of users, but in the meantime they can be easily exploited by attackers.

Consequently, researchers also focused on countermeasures against the leak of credentials. Cox *et al.* proposed *SpanDex*, a secure password tracking runtime that allows the execution of untrusted code in a secure sandbox where the initiated data flows can be analyzed in more detail [6]. Their tool provides Android applications with password protection against attacks in which user passwords are compared to a list of known passwords. They tested their tool using 50 popular applications and found that 84% of them executed without issues while drastically improving their protection against password guessing attacks.

In our work hard-coded developer or user credentials are extracted implicitly during the API endpoint and data scheme extraction.

2.3 Soft factors

Some researchers investigated soft factors, *e.g.*, the user's understanding and perception of network communication, rather than technical aspects.

Benton *et al.* conducted user studies on *Amazon Mechanical Turk* in which they asked people questions related to the understanding of app permissions, including permissions regarding web communication [3]. They found permissions were ineffective, even with the addition of an additional text warning only 57.8% correctly understood the threat. Conversely, they found that an app's download count had a strong effect on app installations. This work shows that people are highly influenced by popularity counters, *i.e.*, users trust the decisions of other people, rather than technical details.

In contrast to the understanding of permissions, Shklovski *et al.* evaluated the perceptions of privacy and mobile app use [17]. They discuss studies focusing on the assessment of users' reactions when confronted with mobile apps accessing and sharing sensitive data. In their reported studies users described these privacy intrusions as "creepy" and as a violation of their personal space. But despite the strong disapproval by research participants of user data sharing such as sensory data and other personally identifiable information, they continued to use the apps in question after being informed about them. The paper concluded that users are starting to feel uncomfortable with excessive data collection by apps and they suspect consequences for businesses that rely on such practices in the future.

It appears that the human perception of threats in the appified world is seldom reasonable and users are in need of additional technical measures to protect themselves from bad decisions.

For that reason Wang *et al.* introduces more comprehensive permission popup windows containing detailed information about specific permissions of the app and ad services as well as allowing the user granular control over them before installing the app [21]. The findings of their study suggest that allowing such granular control over app permissions and ads has a positive effect on users' perceptions of the app as well as app installations while leading users to disclose less information about themselves. Unfortunately, such a comprehensive mechanism has not yet been provided by the Android operating system.

3

Background

In this chapter we introduce the terms and technologies relevant for this work. The focus lies on the network communication aspect within the Android ecosystem. This includes the ways Android applications are able to communicate with external web services, the architectural styles commonly used for such web interfaces and the format in which data is transmitted. We also discuss the topic of code analysis for open-source and compiled applications including the decompilation process, the Abstract Syntax Tree (AST) construction and the symbol resolution. Finally, we will have a look at the tools used for the validation of collected data.

3.1 Android ecosystem

The Android ecosystem consists of the Android Operating System (OS), the devices capable of running any version of the Android OS, the provided services, and the applications developed for Android.

Android was initially an OS designed for cameras that has later been acquired from *Google* who repurposed the OS for (smart)phones. Since its introduction to the market back in 2008 it then quickly expanded into other categories like tablets, smartwatches (Wear OS), televisions (Android TV) and even cars (Android Auto). In addition to the services and applications developed by *Google* themselves, a core element of Android is the corresponding application store called *Google Play* store. The *Google Play* store allows third party developers to distribute their own apps on the Android platform. These applications are

primarily programmed in Java or Kotlin and then compiled and packaged into Android PacKage (APK) files. Packaged APK files are either distributed through the *Google Play* store after undergoing a screening by *Google* for malicious code and other violations of *Google*'s programming guidelines, or through an unofficial third party distribution platform.

3.2 Network-based communication facilities

A majority of Android applications require an internet connection to work correctly, *e.g.*, to display web content inside a web view, accessing resources from web servers, or saving data to back-end servers for synchronization. In order to acquire system features such as internet access and location data access, the application is required to request the relevant permissions from the user first. The user can then review those specific requests and either grant or deny them individually. All of these permissions for a specific Android project must be specified inside the app's manifest file `AndroidManifest.xml` which is located inside the root folder of the source set. The manifest file must follow a certain structure based on the eXtensible Markup Language (XML) as shown in Listing 1. For example, with the `android.permission.INTERNET` permission set the application is then allowed to send and receive data over the network, respectively the internet.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="your.package">

    <uses-permission android:name="android.permission.INTERNET"></uses-permission>

    <application ... >
        <activity ... >
            ...
        </activity>
        ...
    </application>

</manifest>
```

Listing 1: Android manifest file structure shown with a declared "INTERNET" permission

To simplify the implementation of network features in apps, developers have the choice between network-based communication facilities provided by Android or popular third party libraries.

3.2.1 Plain Java methods

Android contains the *java.net* network communication package, which provides developers network-related classes and methods that are identical to the ones found in regular Java applications designed

for desktop systems (*e.g.*, Linux, Apple macOS, and Microsoft Windows). The package contains low-level classes that provide the custom implementation of network sockets, and high-level classes, *e.g.*, `HttpURLConnection`, which abstract from sockets to ease the connection handling. The code block in Figure 3.1a provides a sample implementation of how an HTTP `GET` request could be made using low-level sockets. To emphasize the differences between low- and high-level approaches, Figure 3.1b implements the same functionality as before but using a high-level approach based on the class `HttpURLConnection`.

Ready-to-use Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols are provided by the classes `SSLSocket` and `HttpsURLConnection`.

3.2.2 REpresentational State Transfer

Most mobile applications communicating with a web Application Programming Interface (API) follow the REpresentational State Transfer (REST) architectural style when making HTTP requests to a web service. APIs respecting the REST architectural constraints are called RESTful web services (RWS). An alternative to web services using REST are Simple Object Access Protocol (SOAP) web services. However, in mobile apps RESTful APIs dominate over SOAP APIs, *i.e.*, most popular apps rely on RESTful communication, *e.g.*, Twitter, YouTube, and Instagram.

The commonly agreed six constraints RESTful web services should comply with are [14]:

- **Uniform interface:** A resource on the server is always identified by a unique resource identifier that can be used in requests to access that specific resource. A response to a request contains a representation of the requested resource in a format that may differ from its internal representation and includes all the necessary information to process, modify or delete the resource. A representation of the resource can provide links to related resources if relevant.
- **Client/server architecture:** The client (mobile application) is independent of the server. To access a resource on the server, the client only needs to know the identifier of a resource but nothing about the implementation of the server.
- **Statelessness:** The server does not save any state information of prior requests; the client is responsible for maintaining a state if required. As a result, any state information (*e.g.*, user authentication details) must be attached to each request.
- **Cacheablility:** To improve response times and to reduce the server load, responses include information about their cacheability.
- **Layered system:** The client is unaware of the layers behind the API implementation it is talking to.
- **Code-on-demand (optional):** The server is able to supply the client with executable code to extend the client's functionality.

In RESTful web APIs the resources on the server are specified by a base URL and an API endpoint, *e.g.*,

```
URL url = new URL("http://www.google.ch");
Socket socket = new Socket(url.getHost(), 80);
OutputStream outputStream = socket.getOutputStream();
PrintWriter printWriter = new PrintWriter(outputStream, false);
printWriter.print("GET / HTTP/1.1\r\n");
printWriter.print("Host: www.google.ch\r\n");
printWriter.print("Connection: Close\r\n");
printWriter.print("\r\n");
printWriter.flush();
InputStream inputStream = socket.getInputStream();
InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
BufferedReader bufferedReader = new BufferedReader((inputStreamReader));
int in;
while ((in = bufferedReader.read()) != -1) {
    System.out.print((char) in);
}
bufferedReader.close();
```

(a) Low-level `Socket` implementation

```
URL url = new URL("http://www.google.ch");
HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
InputStream inputStream = httpURLConnection.getInputStream();
InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
int in;
while ((in = bufferedReader.read()) != -1) {
    System.out.print((char) in);
}
bufferedReader.close();
httpURLConnection.disconnect();
```

(b) High-level `HttpURLConnection` implementation

Figure 3.1: Identical HTTP connection examples implemented with low- and high-level Java features

the URL `https://example.com/api/user/bob` uses the base URL `https://example.com` and the endpoint `/api/user/bob` which identifies the resource `bob` on the server. The syntax of such URLs is defined in RFC 3986.[1] To indicate the retrieval, modification, or deletion of resources some standardized HTTP methods are used in the request header:

- **GET:** Indicates that the client wants to retrieve a representation of a specified resource.
- **POST:** Indicates that the client wants to create a new entry on the server of the specified resource in the resource representation sent with the request.
- **PUT:** Indicates that the client wants to replace an existing entry on the server with the resource specified in the resource representation sent with the request.
- **PATCH:** Indicates that the client wants to modify an existing entry on the server according to the information sent with the request.
- **DELETE:** Indicates that the client wants to delete the specified resource on the server.

Consequently, requests using the HTTP methods `POST`, `PUT`, and `PATCH` may include a request body that includes information to create or modify the resource.

3.2.2.1 Data schemes

Resources on a web server can be represented in countless different data formats, *e.g.*, plain text or *JavaScript Object Notation* (JSON). Hence, the HTTP `Content-Type` header in requests to RESTful web APIs is used to specify the format of the resource representation sent with the request. This header is also included in the response and indicates similarly the format of the resource representation sent with the response. In addition, the HTTP `Accept` header can be used to inform to the server in what format the resource representation in the response to the client should be.

Listing 2 illustrates an example of a response header from an API using a JSON-formatted resource representation. In more detail, the `Content-Type` header is specified as `application/json; charset=utf-8` and encompasses the media type `application/json`. The media type consists of the generic `application` type and the `json` subtype. The optional parameter `charset=utf-8` specifies the character encoding as *Unicode Transformation Format 8* (UTF-8). Common other generic types are `image` (*e.g.*, png, jpeg, gif), `text` (*e.g.*, html, csv, css), `audio` (*e.g.*, mpeg, ogg), and `application` (*e.g.*, javascript, xml, sql, pdf, zip). The most common media types used to represent resources when communicating with RESTful APIs are `application/json` and `application/xml`.

JSON is based on a subset of the JavaScript programming language. The JSON protocol is very lightweight and only supports the data types object, array, number, string, boolean, and null. A JSON object always consists of key-value pairs where the key is of type `string`, and the value is of any valid JSON data type. Moreover, a JSON array is basically a list of values of any valid JSON data type. Since JSON can

[1] `https://tools.ietf.org/html/rfc3986`

be used in *any* programming language to easily format serializable data objects as text, it is generally a straightforward choice for any client-server communication as clients and servers can remain completely independent from a technical perspective. Figure 3.2a represents a typical object in the JSON format. The example shows that JSON objects are always surrounded by curly brackets (*e.g.*, the value of the `address` key) and JSON arrays are surrounded by square brackets (*e.g.*, the value of the `badges` key).

```
HTTP 200 No Error

Server: cloudflare
Content-Type: application/json; charset=utf-8
X-Powered-By: Express
Set-Cookie: __cfduid=de87d4e837435504704532491 87c2f5a91550244627;
        expires=Sat, 15-Feb-20 15:30:27 GMT; path=/; domain=.reqres.in; HttpOnly
Access-Control-Allow-Origin: *
CF-RAY: 4a98d0db4b4b3e86-ZRH
Date: Fri, 15 Feb 2019 15:30:28 GMT
Connection: keep-alive
Content-Length: 443
Expect-CT: max-age=604800,
        report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
Etag: W/"1bb-D+c3sZ5g5u/nmLPQRl1uVo2heAo"
```

Listing 2: Response header example

XML on the other hand is based on a subset of the complex Standard Generalized Markup Language (SGML). Identically to JSON it is also programming language agnostic and it is used to format serializable data objects as text. Moreover, XML is widely used as a file format, *e.g.*, for persisting designed application user interfaces in Android app development, for configuration/property files, and as a foundation for other document formats. XML is not as lightweight as JSON, which is presumably one of the main reasons that JSON remains usually the preferred choice amongst the majority of web API developers. In XML data is structured using nested tags. Figure 3.2b shows the same object as in Figure 3.2a, but now formatted in XML. The example shows the XML declaration `<?xml version="1.0" encoding="UTF-8" ?>`, which entails the XML version and the character encoding. In XML every value is always surrounded by tags. For example, the whole object is surrounded by the tags `<root>` and `</root>`, while a simple element like `id` is surrounded by the tags `<id>` and `</id>`.

3.2.2.2 Libraries

Writing network-related code as described in subsection 3.2.1 introduces a lot of boilerplate code in projects as it requires the application programmer to handle repetitive complex workflows using input and output streams and asynchronous/concurrent requests. As a result, numerous third party network libraries emerged that make the implementation of network-related code more efficient for Android developers. At the same time libraries for the (de)serialization of JSON and XML data have also become available. We

```json
{
    "id": 3948292,
    "name": "Miller",
    "first_name": "Bob",
    "address": {
        "street": "Sample Street 1",
        "number": 10
    },
    "married": true,
    "badges": [
        "badge1",
        "badge2"
    ],
    "social_media": null
}
```

(a) JSON object example

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <id>3948292</id>
  <name>Miller</name>
  <first_name>Bob</first_name>
  <address>
    <street>Sample Street 1</street>
    <number>10</number>
  </address>
  <married>true</married>
  <badges>badge1</badges>
  <badges>badge2</badges>
  <social_media/>
</root>
```

(b) XML object example

Figure 3.2: Content-wise identical data structures implemented in JSON and XML

found the following libraries in apps during the manual review of 160 random open-source projects and 10 closed-source apps:

- **Apache HttpClient** is a network library developed by the Apache Software Foundation.[2] It is a direct competitor to newer HTTP client libraries like *OkHttp* as its features include handling of HTTP requests, encrypted communication using Transport Layer Security (TLS), and response caching. Asynchronous requests are supported using `HttpAsyncClient`. The library is mainly used by older projects.
- **glide** is an open-source library specifically designed to manage image downloads over a network.[3] It handles caching of the downloaded images either in memory or on disk to prevent Android applications from downloading the same images more than once. Furthermore, applications with a need to download many remote images can use *glide* on top of other web API HTTP libraries.
- **Gson** is an open-source data conversion library developed by *Google*.[4] It is commonly used alongside an HTTP client library to handle the transformation of Java objects into JSON formatted text and vice versa. The library uses annotations to provide developers more control over the data stored in the JSON representations.
- **ion** is an open-source network library for Android applications.[5] The library is less popular than its alternatives, but it supports an extraordinary number of features: It not only handles the usual suspects such as HTTP requests, asynchronous requests, caching, gzip compression, HTTP/2 support, and encrypted communication using TLS, it also supports asynchronous file or image downloads and JSON (de)serialization using the *Gson* library.
- **Moshi** is a data conversion library developed by Square.[6] It has the same feature set as *Gson* and is also used for the forward and backward transformation of Java objects into JSON formatted text. For more control over how Java objects should be transformed, *i.e.*, the amount of information that should be preserved, *Moshi* uses custom annotations. *Moshi* supports data types like array, collection, enum, string, and other primitive types. *Moshi* and *Gson* are equally popular and thus it is often the result of a personal preference or the accompanying HTTP client which library will be selected for a new Android project.
- **OkHttp** is a very popular open-source network library developed by Square.[7] It allows the effortless implementation of (a)synchronous HTTP requests for web APIs, including automated concurrent request management (*i.e.*, connection pooling). Its contributors regularly update the code against security threats. Security features like Application-Layer Protocol Negotiation (ALPN) and certificate pinning are supported by the *OkHttp* library. Furthermore, responses are cached to achieve shorter

[2] `https://hc.apache.org/httpcomponents-client-ga/`
[3] `https://github.com/bumptech/glide`
[4] `https://github.com/google/gson`
[5] `https://github.com/koush/ion`
[6] `https://github.com/square/moshi`
[7] `https://square.github.io/okhttp/`

response times and to eliminate unnecessary network traffic. It also supports the gzip algorithm for (de)compression of request bodies to reduce network traffic. By supporting HTTP/2 it is possible to send multiple requests over a single TCP connection, which further increases the efficiency in data communication.

- **org.json** is the default JSON library of Android.[8] It supports the construction of JSON objects and the conversion of JSON formatted text into JSON objects and vice versa. However, this library does not provide support for data conversion from and to Java objects unlike *Gson* and *Moshi*.
- **Retrofit** seems at the first glance very similar to *OkHttp* because is also marketed as an "HTTP client for Android and Java" by its makers at Square.[9] Nevertheless, *Retrofit* is neither a replacement for nor a competitor to *OkHttp*, but rather an extension that builds on top of *OkHttp*. In fact, *Retrofit* internally uses *OkHttp* and it is, like *OkHttp*, among the most popular HTTP clients in Android development. *Retrofit*'s features include interfaces for API endpoint definition, handling of synchronous and asynchronous HTTP requests and HTTP body transcoding using converters. API endpoints and request headers are defined using Java interfaces, and annotations are used to indicate the HTTP methods for a specific endpoint. For the deserialization of request bodies *Retrofit* provides converters, which are used to (de)serialize JSON and XML text to and from Java objects. These converters support other popular third party data conversion libraries, *e.g.*, *Gson*, *Moshi*, and *Simple*.
- **Simple** is a framework explicitly designed for XML serialization.[10] It is one of only few third party options for XML serialization on the Android platform, and like most other XML frameworks or libraries it is not actively maintained anymore, *i.e.*, the last release dates back to 2013. The serialization of Java objects works similarly to *Gson* and *Moshi*, using annotations for XML elements and attributes.
- **Volley** is an open-source HTTP library officially supported by *Google*.[11] It provides a similar feature set to *OkHttp* and *Retrofit*, but it further includes support for image downloads. *Volley*'s primary use cases are the eased use of HTTP requests to web APIs and the handling of image downloads and JSON formatted text. Apparently because of the similar feature sets it is an ongoing debate if *Volley* is better or worse than its alternatives and it ultimately comes down to personal preference. *Retrofit* and *OkHttp* seem to have a more active community support and seem to be preferred by developers over *Volley*.

3.3 Code analysis

A program can be analyzed using either dynamic or static program analysis. With dynamic program analysis the analysis is performed during run time of the program, while static analysis does not need

[8] `https://developer.android.com/reference/org/json/package-summary`
[9] `https://square.github.io/retrofit/`
[10] `http://simple.sourceforge.net/`
[11] `https://developer.android.com/training/volley/`

to execute the program. When, for example, analyzing the network code in an Android application, the dynamic program analysis approach would include running the application and triggering certain functionality in the application while monitoring the network traffic to test for functional or security issues. On the other hand, static program analysis works directly on the source code. Examples are IDEs and linting tools that are able to display syntax errors, unused functions, and undeclared variables before compiling the program. Another popular use case for static analysis is code optimization, which is applied by compilers during the compilation process. Popular tools which support static code analysis functionality for Java include *IntelliJ IDEA*, *Eclipse*, *Checkstyle*, and *Soot*.

3.3.1 Decompilation

Decompilation is only required if the originating (byte code) files are not available as human readable source code. Before we discuss the decompilation process it is important to have a quick look at the compilation process as it is conceptually the reverse.

Android Java development is usually done using either *Android Studio* or another IDE/text editor to write Java source code. To make this source code executable on any Android device it is necessary to compile the Java source code and then to package it into an APK file that contains all the code and resources required to run the application.

The APK compilation process is performed in multiple steps. First the standard Java programming language compiler `javac` is used to compile the source files including any libraries into Java byte code (*i.e.*, Java `.class` files). Next, some developers use a tool named *ProGuard* to obfuscate class, field, and method names to make it more difficult for outsiders to decompile and reason about the app. These Java byte code files are then recompiled again into Dalvik byte code which outputs a single `.dex` file. Using the "Android Asset Packaging Tool" the `.dex` file and the application resource files are then packaged into one APK file. Finally, this generated APK file must be signed as the signing ensures that the contents of the APK file cannot be tampered with without invalidating the checksum.

For decompilation, each compilation step has to be reversed which is a non-trivial problem as some steps are irreversible and must be approximated, *e.g.*, obfuscated variable names cannot be successfully restored without informed decisions based on identical non-obfuscated code. Hence, decompiled source code suffers often from flaws as it is very rare for decompilers to reconstruct everything accurately. This is especially the case for code that has been obfuscated during the compilation process. For Android, applications tools like *JADX* and *Apktool* can be used for decompilation and reverse engineering to turn APK files back into Java source code.

3.3.2 Lexing and parsing

When analyzing source code the first step is to read the characters in the source code file and to structure them into tokens according to the lexical grammar of the corresponding programming language. This process is called "lexing". Examples of such tokens are the literal expression `true` and the plus operator `+`.

The resulting token sequence can then be turned by a parser into a more abstract tree-like data structure to enable syntax validation. In general, a parser outputs an AST which then can be traversed for further analysis of the source code's syntactic structure. This process is called "parsing". For instance, the tokens `([3],[+],[3])` would then be contained by a `BinaryExpression` node to ease their evaluation.

Besides IDEs like *IntelliJ* and *Eclipse* that both expose their constructed ASTs there also exists a popular open-source library called *JavaParser* which is able to build ASTs for Java code compliant with the Java language specifications 1.0 to 12.0.[12]

Such ASTs mostly consist of nodes and their relationships as illustrated with the code snippet in Figure 3.3a that leads to the AST shown in Figure 3.3b. An AST node can have multiple children but only one parent, and by traversing the tree the whole source code can be reconstructed. In *JavaParser* the `CompilationUnit` is always the root node of a tree, and the whole tree represents one `.java` source file. The *CompilationUnit* has children for the package information, imports if there are any (which is not the case in Figure 3.3b), and types which contain the class declarations, *i.e.*, represented by the class nodes. Class nodes often have children for their own names and members, *e.g.*, the method declaration nodes. The method declaration nodes comprise children with information about their own names, the name of the method return type, and a body node that is the root node of all the AST nodes representing the source code inside the `example()` method. However, in Figure 3.3b the class node maintains only one method declaration.

3.3.3 Symbol resolution

It is important to see that in Figure 3.3b the AST only provides type names for certain nodes, *e.g.*, for the variable `helloWorld` in the return statement no type information is provided at all. Moreover, there is also no direct relation in the AST between the variable `helloWorld` in the return statement and its declaration in the previous line. The reason for this intended behavior is that this information is irrelevant for code syntax. In order to resolve this name expression to find its declaration or to find other relations between nodes the *JavaParser* includes the `SymbolSolver` tool. `SymbolSolver` can be applied on variables to find declaration nodes, but also on method calls to find method declarations, or on type nodes to get their fully qualified names. This is conceptually similar to features found in popular Java IDEs. For example, *IntelliJ* enables the programmer to jump to a specific symbol declaration by just clicking on it.

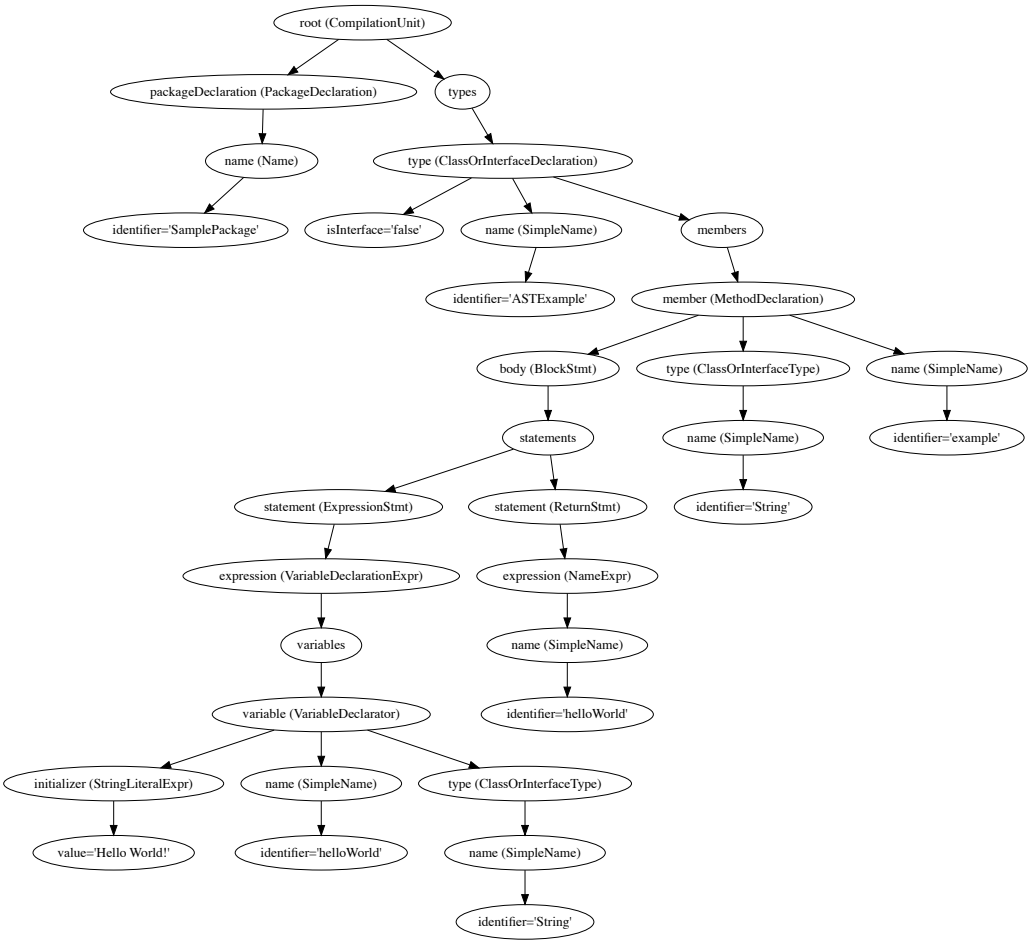
[12] `https://github.com/javaparser/javaparser`

```java
package SamplePackage;

public class ASTExample {

    public String example() {
        String helloWorld = "Hello World!";
        return helloWorld;
    }
}
```

(a) “Hello World” Java code snippet



(b) Corresponding AST of the “Hello World” code snippet

Figure 3.3: Transformation of source code into an AST

3.4 Validation

Code analysis can be used to extract web API related data from the source code of Android applications. After the successful extraction of such data it is recommended to further evaluate the found API endpoints by sending requests to them, because false positives are (almost) always a concern in static analyses. The validation additionally leverages potential configuration and architectural data about web services. Fortunately, tools like *Postman* and *CocoaRestClient* that originally were developed to aid the construction of web APIs can be repurposed to test existing web services. More precisely, these tools can be used out-of-the-box to send request messages by specifying, for example, the endpoint, the HTTP method, the query parameters, the request headers, and the request body which yield a server response containing, for instance, a response body and multiple headers including transmitted cookies.

4 Methodology

In this chapter we describe the methodology used to analyze the network-related functionality of open-source and closed-source Android applications and their respective web endpoints. First, we discuss our dataset on which our experiments are based on before we explain the preliminary manual analysis we performed. Finally, we elaborate on the analyses, *e.g.*, parametrization of the automated analyses, and discuss how the preliminary results from the manual analyses have been validated.

4.1 Dataset

The open-source dataset for our analysis consists of 3 114 open-source projects from *F-Droid* with source code available on public code sharing platforms like *GitHub*. The collection entails applications from different categories, sizes, with or without third party libraries, and supports various Android API levels, *i.e.*, Android releases. The downloaded packaged projects of these open-source projects contain all the necessary files, scripts and resources to build and compile the application. Different versions of the same application are included in this collection, however, we always strived for the most recent release. Furthermore, we rejected all apps that did not use any of Android's internet communication facilities. After that cleanup 413 apps were left for our analysis.

The closed-source dataset contains 2 500 random closed-source applications that originate from the *Google*

Play store. The collection similarly entails applications from different categories, sizes, with or without third party libraries, and supports various Android API levels. Every closed-source application contained in this collection is represented by the APK file distributed by the *Google Play* store. We only considered apps for the analysis where we could fetch the relevant meta information from the *Google Play* store web page, and removed all the apps that were unavailable in the store. Furthermore, we rejected all apps that did not use any of Android's internet communication facilities. After that clean up 834 apps were left for our analysis.

4.2 Preliminary work

In this section we discuss the feature detection and the file conversion strategies performed prior to the comprehensive analysis. To determine and extract relevant features we gathered information on how Android applications interact with web APIs and on how this process could be automated by manually analyzing subsets of the dataset.

4.2.1 Open-source projects

For open-source projects the source code and build configuration files are available by definition. Most of the applications can be built using Android's default build automation system called *Gradle*, which automatically resolves any third party library dependencies. The typical Android project structure and build configuration is depicted in Figure 4.1. The relevant files for assessing network-related functionality of open-source Android applications are the `AndroidManifest.xml` file, the `gradle.build` files, and the source code files either written in the Java or Kotlin programming language.

The `AndroidManifest.xml` file must be stored within the main source code folder and contains valuable information, amongst others because it specifies which permissions the application intends to use. Hence, we automatically excluded all applications that did not explicitly include the manifest permission `android.permission.INTERNET`, *i.e.*, the capability to use network socket communication. The required tool for this selection process has been open-sourced.[1]

The project depicted in Figure 4.1 specifies two different `gradle.build` files: One project-level (top-level) Gradle file and one additional module-level (low-level) Gradle file for the module "app". The project-level Gradle file always applies to the whole project (*e.g.*, specifying the required dependencies and repositories), whereas each subordinate build configuration only applies to a single module within a project (*e.g.*,specifying the required Android version, or requesting code obfuscation).[2] Moreover, in Figure 4.1 the module-level `build.gradle` file includes a `dependencies` block that specifies the necessary imports of multiple network and data conversion libraries, *i.e.*, *OkHttp*, *Retrofit* and *Moshi*. If projects

[1] `https://github.com/wozuo/aipchecker`
[2] `https://developer.android.com/studio/build/`

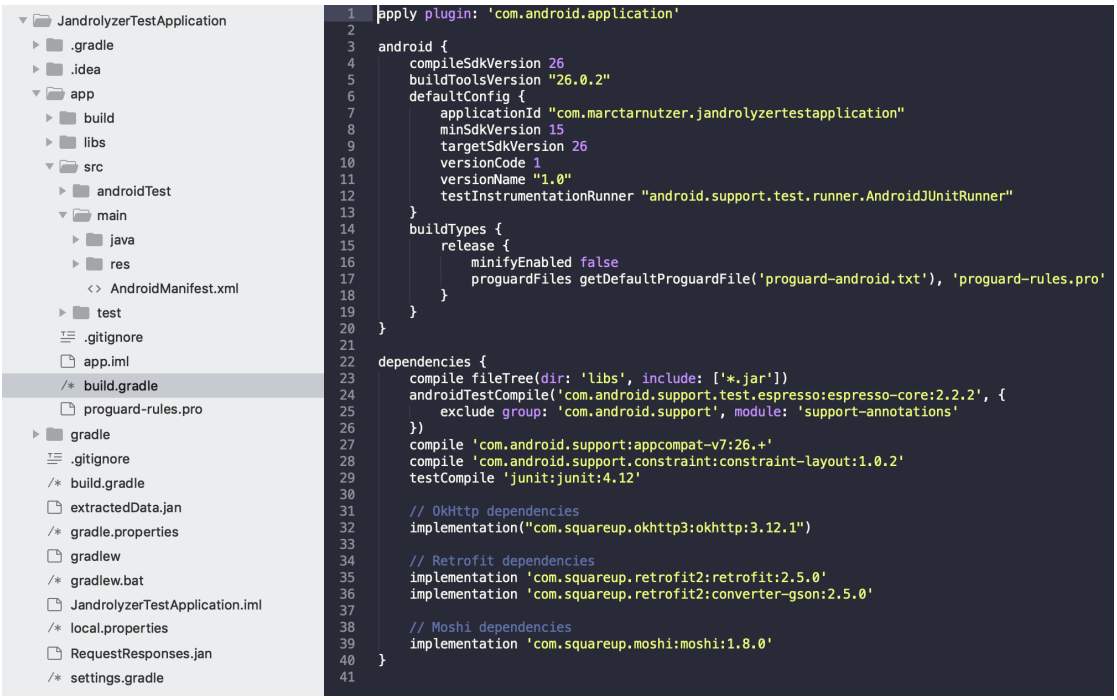


Figure 4.1: Typical Android project structure and build configuration

use third party libraries they generally can be found as declarations in the `gradle.build` files or in the `libs` folder packaged as a `.jar` file. Consequently, we investigate all `gradle.build` files and the `jar` folder to discover which third party network libraries are used in a specific open-source project.

4.2.2 Closed-source applications

Closed-source applications are always packaged as `.apk` files. An APK file includes the `classes.dex` file which represents the compiled VM byte code. This `classes.dex` file must be decompiled before any analysis based on Java source code can take place.

We used the *JADX* tool for decompilation, which can be used either from the command line or through a graphical user interface. Figure 4.2 shows a screenshot of the *JADX* tool applied to a compiled Android application that we developed for testing purposes. The resulting decompiled source code is syntactically correct and names of classes, variables, and methods have been well preserved. We can further see that *JADX* decompiles the source code of the referenced libraries and their dependencies. If code has been obfuscated during the compilation phase, *e.g.*, using the ProGuard tool, *JADX* would not be able to recover most original names used for classes, variables, and methods, but the major logic would remain the same. In other words, the decompiled obfuscated source code would still be syntactically correct, but would have short nonsensical names instead of those chosen by developers. For instance,

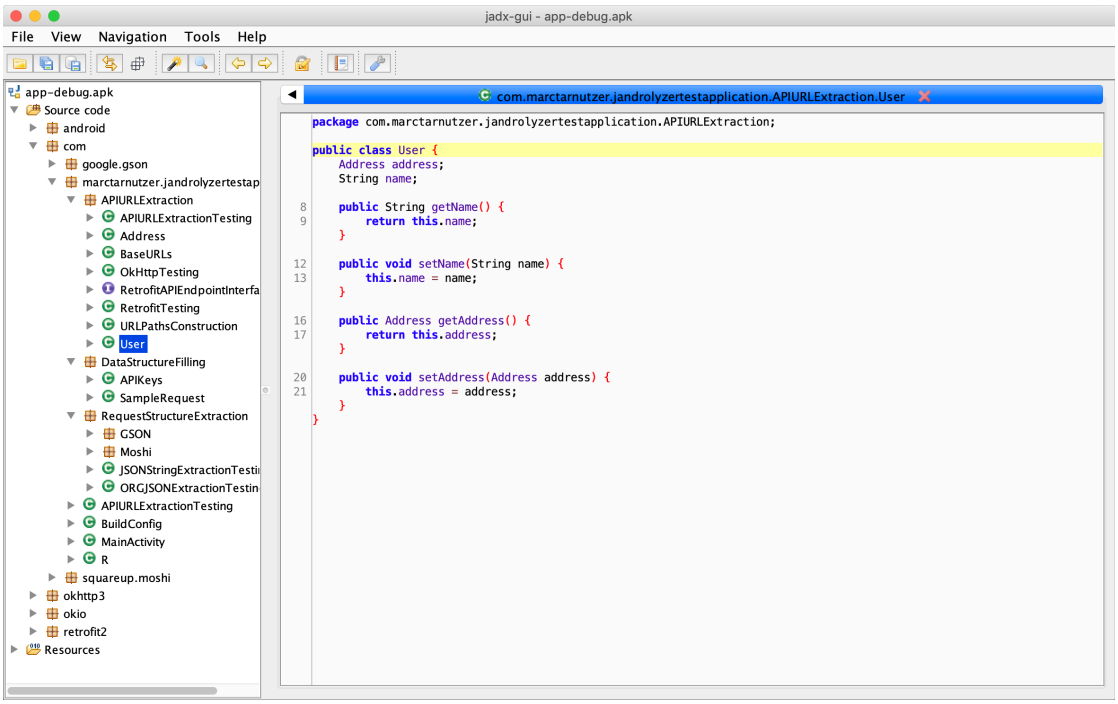


Figure 4.2: Screenshot of the *JADX* tool applied to a compiled Android application

`someObject.someMethod()` in the original source code would then become `a.b()`. During our manual tests we encountered many commercial apps that either obfuscated the code only partially, or even completely abandoned any obfuscation techniques, which immensely eased our reverse engineering tasks. Nevertheless, for larger apps *JADX* is barely able to properly decompile the whole source code. Every time it encounters such an incompatibility it includes the error message as a comment in the decompiled source code, but still continues to decompile the remaining byte code. If *JADX*, for example, would be unable to decompile the method `hwDecodeImage(Image2D)`, this method would still be included in the decompiled source code, however, it would only contain a comment containing a dump of the irrecoverable byte code instructions supplemented with the parser's error encountered error and an `UnsupportedOperationException` exception throwing the message "Method not decompiled". After the successful (partial) decompilation we continued with the analyses as with the open-source projects.

4.3 Analyses

During the initial manual analysis of 160 apps the feature extraction, lexing, and parsing were not automated, but done by manual intervention. After we manually identified network and data conversion sink method calls we started to manually reverse engineer the applications' network functionality in

depth. Information about API endpoints or payloads attached to requests, *e.g.*, in the form of JSON or XML strings, were of special interest. This information eventually led to the creation of the analysis tool *Jandrolyzer*, which is based on *JavaParser*.

4.3.1 Manual analysis

We conducted a manual code review of the source code contained in all the `.java` files for a subset of the open-source projects in our dataset. During this code review we compiled a list of different network-related code snippets and the found libraries. We identified sink MCEs, *e.g.*, calls to web APIs, and checked for each if there is any additional hard-coded information available (*e.g.*, URLs, JSON, XML data, *etc.*) by manually resolving their variables, respectively their symbols, across different methods and classes. Such an accurate symbol resolution was required to limit the search scope by type (*e.g.*, "method call"), instead of just by name. An example sink MCE `openConnection()` is presented in Listing 3. In this example, if we resolve the `url` variable to its declaration it can be seen that a `GET` request to the endpoint `http://www.example.com/api/getUsers` will be performed that could potentially lead to a data leak of registered users. As a reminder, this is a rather trivial code example and in the wild such method calls, *e.g.*, the instantiation of the `URL` object and `url.openConnection()`, are often spread across several different methods and classes.

```
1 URL url = new URL("http://www.example.com/api/getUsers");
2 HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
3 InputStream inputStream = httpURLConnection.getInputStream();
```

Listing 3: Sink method call example

4.3.2 Automated analysis

Based on the manually collected data we then built another tool named *Droidalyzer*, which scans all `gradle.build` files of a project for known network- or data conversion-related third party libraries and conducts a keyword search over the source files with the aim to find even more network-related code that supports us in finding relevant features.[3] Ultimately, the automated analyses have been performed by *Droidalyzer*'s successor *Jandrolyzer* in which we implemented rules according to our findings in the manual analysis and *Droidalyzer*'s output. We executed *Jandrolyzer* on a dual octa-core processor setup with 128 GBytes of memory. We restricted each analysis to 30 minutes of CPU time (one core) and to a recursion depth of 15. We ran eight analysis jobs in parallel together with, if necessary, two decompilation jobs.

[3] `https://github.com/wozuo/droidalyzer`

4.4 Validation

To validate the results we found during our manual analysis we used the *CocoaRestClient* tool, which simplifies the testing of HTTP/Restful endpoints.[4] In some cases during the manual analysis of the source code it was not always clear how the found data correlates. For instance, we found a base URL and an endpoint for a specific web API, but lacked the used HTTP method. In such cases we had to try different HTTP methods while validating the web server responses. Moreover, data could be incomplete due to missing user input as illustrated by the API call `http://example.com/api/search?keyword=`. We similarly observed incomplete data with JSON objects in which the keys of the key value pairs have been hard coded but not their corresponding (user) values. We had to fill those blanks with values that have a high probability of triggering a successful response from the server (*e.g.*, number values to indicate the start and end of a selection). Based on the HTTP status code, the HTTP headers, and the response message it is then possible to (manually) validate whether the data found during the static code analysis is correct.

To validate the results we found during our automated analysis we implemented a validation facility in *Jandrolyzer* that reassembles and executes the found API calls. The server responses of those calls are logged for further analyses and for the tool improvement.

[4] `https://github.com/mmattozzi/cocoa-rest-client`

5

Implementation

In this chapter we introduce *Jandrolyzer*, the tool we developed to automate the analysis of network communication functionality of Android applications.[1] First, we discuss the concept of the tool, followed by a close look at its decompilation and data extraction stages. In the end, we discuss how the tool verifies the collected data.

Jandrolyzer is intended to automate the individual static analysis steps we performed during the manual analysis. The tool can be used to analyze open-source Android projects or closed-source applications as APK files. If it is applied to an open-source project, the tool searches the `build.gradle` files for known network and data conversion libraries. The collected data is then used to improve the reverse engineering of network and data conversion code using code parsing and symbol resolution techniques. Based on the results of the *Droidalyzer* tool which emerged from the manual analysis' needs, we added library support for the two most common network libraries, *i.e.*, *OkHttp* and *Retrofit*, and the three most prevalent data conversion tools, *i.e.*, *org.json*, *Gson*, and *Moshi*. On the other hand, if *Jandrolyzer* is applied to a closed-source app it first attempts to decompile the APK file before it starts to analyze the (partially) decompiled application in the same manner as with the open-source projects. After the data extraction step is completed, the tool will fill in missing values based on the variables extracted from the source code using the similarity distance, also known as Jaro-Winkler distance, between the extracted variable names and the key name in question. Next, *Jandrolyzer* sends requests to the resulting API endpoints using the

[1] `https://github.com/pgadient/jandrolyzer`

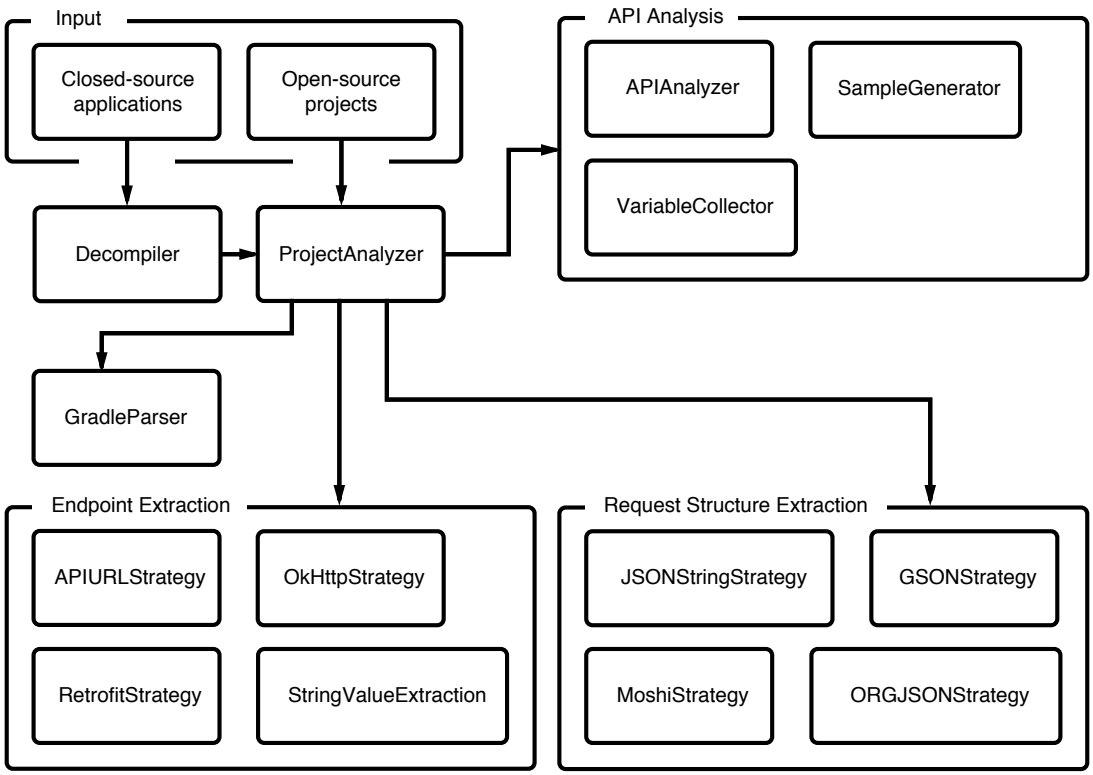


Figure 5.1: Internal structure of *Jandrolyzer*

extracted data structures as payload and saves the retrieved data from the web server responses. This data can then be used to assess the presence of security issues, *e.g.*, hard coded API keys, plain text sensitive data transmissions, data leakages, or outdated server infrastructure with known security issues.

The purpose of *Jandrolyzer* is to provide Android application developers, security researchers, and application reviewers a tool that facilitates the efficient identification network-related source code or data, and that enables out-of-the-box the security auditing of apps and their API endpoints. *Jandrolyzer* can not only be used to identify potential security issues, it can also help uncover malicious application functionality.

5.1 Concept

The diagram in Figure 5.1 illustrates the internal structure of *Jandrolyzer*. The analysis process starts with the classes in the top left “Input” box. The application has been designed with maximal flexibility in mind; it allows one to independently run the decompilation, the app analysis, and the API evaluation. The results of each step are saved to files inside the corresponding (decompiled) project folder. Moreover, *Jandrolyzer*

is easily extendible with new strategies for future library or expression type support. *Jandrolyzer* can be started with various arguments providing options to start the analysis of one or more closed-source apps in the form of `.apk` files or of one or more open-source projects by providing the corresponding project folders. In case the input is a closed-source application, *Jandrolyzer* will first decompile the APK file and create a Gradle project folder. This newly created project folder or the open-source project folder are then processed by the `ProjectAnalyzer` class. `ProjectAnalyzer` will first scan all the `build.gradle` files included in the project to detect if any known third party network or data conversion libraries are used. This information is then used to accurately initialize the type resolver with dependency injections before starting the actual analysis. During the analysis, the *JavaParser* library creates an abstract syntax tree for every `.java` file within the project.

After the ASTs have been built, *Jandrolyzer* traverses the nodes of the AST and identifies network code snippets and other predefined expression types for which it triggers the appropriate extraction strategies. Every "interesting" node will then be passed to either one of the endpoints or the request structure extraction classes. The endpoint extraction classes in the "Endpoint Extraction" box include strategies for how expressions must be handled that potentially contain base URLs or endpoint information. We implemented such strategies for the *OkHttp* and the *Retrofit* libraries in the `OkHttpStrategy` and the `RetrofitStrategy` classes, respectively, and also for `java.net.URL` objects, strings constructed with `StringBuilder`, the `concat()` method, binary expressions, and any combination thereof. The request structure extraction classes in the "Request Structure Extraction" box provide similar capabilities for JSON formatted text constructed using strings, for JSON objects constructed or converted using the *Gson* and the *Moshi* third party libraries, as well as for the *org.json* library which is built into the Android system.

The found network snippet data, endpoints, and JSON structures are collected and forwarded to the API analysis classes in the "API Analysis" box of the *Jandrolyzer* structure. The `APIAnalysis` package contains classes like `SampleGenerator`, `APIAnalyzer`, and `VariableCollector`. The `SampleGenerator` is invoked in the `APIAnalyzer` class and is used to prepare the endpoint and JSON data by filling in missing values previously collected by the `VariableCollector` class. `APIAnalyzer` will then send requests to all of the supplied endpoints using the specified HTTP methods and request bodies. The server response messages are collected and saved to a file within the project folder for later analysis.

5.2 Decompilation

Jandrolyzer first forwards the data to the `Decompiler` class if the input is a closed-source application. The `Decompiler` class performs the decompilation by invoking the command line interface version of *JADX* using the Java class `java.lang.ProcessBuilder`. Since *JADX* is used for decompilation, the path to the *JADX* binary must be specified when launching *Jandrolyzer* for any decompilation tasks. The

output of the decompilation process is an Android app project folder that includes Gradle build scripts and information about the Android SDK version the application is targeting. *Jandrolyzer* uses this information to accurately inject the code of specific library versions into the follow up symbol resolver.

During the decompilation process the console output of *JADX* is monitored using a `BufferedReader`. If the *JADX* process terminates without any errors, a file with the name "noJadxErrors" is created inside the newly created projects folder. On the contrary, if the *JADX* process finishes with errors, a file with the name "hasJadxErrors" is created inside the newly created project's folder. Errors are very common during the decompilation process. However, the decompilation routines of *JADX* are very robust and continue even if errors occur: The generated code still remains syntactically correct with comments in places where the decompilation failed. *Jandrolyzer* contains logic to act accordingly when encountering incomplete source code.

5.3 Detection and extraction

The main feature of *Jandrolyzer* is the automated detection and extraction of sink method calls and network-related information like base URLs, API endpoints, and JSON data. *Jandrolyzer* is able to trace variables and method calls back to their declarations across classes, and to check if they contain hard-coded information by using recursive methods. Before any analysis, a path to the folder containing different versions of third party network and data conversion libraries together with the Android core system library has to be specified as a command line argument when starting *Jandrolyzer*. Furthermore, the project's class path and recursion depth argument is represented by an integer that is also required in the initial command line argument; the recursion depth specifies the upper bound on the number of recursive symbol resolution iterations that can be applied to one specific expression.

After successfully starting *Jandrolyzer* on a source code project, the `ProjectAnalyzer` object is created and its `run()` method is invoked. During its initialization an object of the `Project` model class is created that represents the Android project and is used throughout the analysis process to collect any extracted data. `ProjectAnalyzer` uses the *JavaParser* library for parsing and type resolution. In *Jandrolyzer* a `SymbolSolverCollectionStrategy` class is used to add all the classes contained in `.java` files to the type resolver. `SymbolSolverCollectionStrategy` also injects classes from the Java system library, *e.g.*, `java.net.Socket`, into the type resolver using a `ReflectionTypeSolver`. *Jandrolyzer* will then scan all `build.gradle` files for the required Android SDK version and known third party network or data conversion libraries. The resulting information is used to add the relevant source code of the Android SDK and of the found third party libraries to the symbol resolver to let *Jandrolyzer* successfully resolve their classes. For that purpose, we assembled an Android SDK source code and third party library collection of all the releases we could find online. If the desired version is unavailable in the collection, the most recent version of the source code is added instead. Fortunately, decompiled projects do not suffer from this problem as they benefit from the comprehensive decompilation process: The third

party libraries are decompiled as well which makes their sources available to the symbol resolver without any additional effort.

After the symbol resolution has been set up, the *JavaParser*'s `ParserCollectionStrategy` is used to create a `SourceRoot` collection, which is then parsed to transform all `.java` files into `CompilationUnit` objects. As a result, the `SourceRoot` collection fosters transparent access to all source code. Each `CompilationUnit` is also the root node of the AST created by *JavaParser* for the corresponding `.java` file. *Jandrolyzer* first analyzes the import declarations of a `CompilationUnit` and then starts to iterate through all of the nodes recursively. If the current node is an instance of `ObjectCreationExpr` (OCE), `CastExpr` (CE), `MethodCallExpr` (MCE), `StringLiteralExpr` (SLE), `BinaryExpr` (BE), or `VariableDeclarator` (VD) it is further evaluated for sink method, URL, or JSON data.

5.3.1 Sink methods

The detection of network code snippets is based on AST nodes of the types "object creation expression" (OCE), "cast expression" (CE), or "method call expression" (MCE).

The code snippet extraction is currently implemented for the Java packages `com.android.volley`, `com.mcxiaoke.volley`, `com.squareup.okhttp`, `com.squareup.retrofit`, `java.net.Socket`, `javax.net.ssl.SSLSocket`, `java.net.URLConnection`, `java.net.HttpURLConnection`, `javax.net.ssl.HttpsURLConnection`, `org.apache.httpcomponents`, `android.net.http.AndroidHttpClient`, `com.loopj.android.http.AsyncHttpClient`, `com.koushikdutta.ion`, and `com.github.bumptech.glide`. This list can be extended without much effort by implementing snippet detection strategies for more package names followed by the registration of relevant source code files for the symbol resolver.

```
String url = "http://www.example.com";
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
        new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        System.out.print("Response: " + response);
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        System.out.println("Error: " + error);
    }
});
```

Listing 4: Extracted *Volley* `StringRequest` code snippet

An extracted network code snippet based on an OCE node is shown in Listing 4. The example illustrates the code required for performing an HTTP `GET` request using the *Volley* network library and its `StringRequest` class. For its detection *Jandrolyzer* must first visit each node of the corresponding AST to check if it represents an instance of the class `com.android.volley.toolbox.StringRequest`. Next, in order to successfully resolve the `stringRequest`'s OCE node in the AST, *JavaParser* needs to resolve all of the OCE node's arguments, *i.e.*, the request method, the URL, and the response listener. However, the symbol resolution process of *JavaParser* is not guaranteed to be successful, because the tool is still in early development and some call chains might break the resolution as would be the case for this example.

Hence, we developed a unique node fingerprinting technique which is able to circumvent these limitations of *JavaParser*. The fingerprinting does not solely rely on the name of an expression; it additionally takes into account the number of arguments, their names and their types. For example, in Listing 4 we see the method's three arguments `Request.Method.Get`, `url`, and `Response.Listener`. This technique ensures that even if the original node cannot be resolved successfully, we are still able to detect code snippets with a high accuracy. Besides the `StringRequest` class *Jandrolyzer* also detects and extracts instances of *Volley*'s `JsonArrayRequest`, `JsonObjectRequest`, and `ImageRequest` classes.

```
1 URL url = new URL("http://www.webservicename.com/api/getUsers");
2 HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
3 httpURLConnection.setRequestMethod("GET");
4 InputStream inputStream = httpURLConnection.getInputStream();
```

Listing 5: `HttpURLConnection` code snippet

When analyzing CE nodes of ASTs an identification technique based on string matching is used. An extracted network code snippet that emerges from a CE node is shown in Listing 5, where a `java.net.HttpURLConnection` instance is used to send a request to a web server using the HTTP `GET` method. *Jandrolyzer* visits every CE node of the corresponding AST and if one is found with the name "HttpURLConnection" the AST is consulted one more time to find the originating expression which is being cast. In our case, if the originating expression is an MCE node with the name "openConnection" it will be resolved one more time, and in case the final node is of type `java.net.URLConnection` the snippet is saved to disk.

An extracted network code snippet based on an MCE node is shown in Listing 6, where an API call to `http://example.com` is performed using the HTTP `GET` method to retrieve a JSON server response. In this example, *Jandrolyzer* would traverse the AST for MCE nodes with the name "with". Because this is rather a common method name, *i.e.*, even used more than once in our own library dataset, symbol resolution is required to prevent false positives. In our example, if the type is equal to `com.koushikdutta.ion.builder.LoadBuilder` the snippet is being saved.

```
1 Ion.with(this)
2     .load("http://example.com")
```

```
    .asJsonObject()
    .setCallback(new FutureCallback<JsonObject>() {
        @Override
        public void onCompleted(Exception e, JsonObject result) {
            System.out.print("Result: " + result);
        }
    });
```

Listing 6: *ion* code snippet

After the successful detection of any code snippet more snippet-specific information is collected before it is saved to disk. Of particular interest are the name of the involved `.java` file, its path, the library name, the start and end line numbers where the snippet has been found, the type information, and the relevant source code of the class, method, and the snippet itself. An output generated by *Jandrolyzer* with omitted class and method code is shown in Listing 7.

```
Name:
OkHttpTesting.java
Path:
.../OkHttpTesting.java
Library:
com.squareup.okhttp3
Networking code:
new Request.Builder()
Lines:
[210, 210]
Type:
okhttp3.Request.Builder
```

Listing 7: *Jandrolyzer* network snippet report

5.3.2 Unified Request Locators (URLs)

We explained the structure of RESTful URLs in subsection 3.2.2, nevertheless we briefly show the components of an API URL on a more concrete example in Figure 5.2 to recall the key elements. In this example, `https` is the *scheme* that ensures the network traffic between the client and the server is protected using an SSL/TLS encryption. The *scheme* in combination with the *authority* represents the *base URL* of the API. The *path* extends the *base URL* and both together represent the *API endpoint*. *Queries* and *fragments* are used to request individual resources on a server. The *query* consists of key-value pairs that are separated by a delimiter symbol (*e.g.*, "&"), and at the end, the fragment identifier is used to further select a specific section of a given resource, *e.g.*, the second page of a document resource.

Jandrolyzer inspects AST nodes of the types OCE, MCE, SLE, BE, and VD, to detect URLs in the source code. The package `EndpointExtraction` includes classes with strategies to analyze those AST nodes and to extract URLs from them. *Jandrolyzer* supports the extraction of URLs from `java.net.URL`

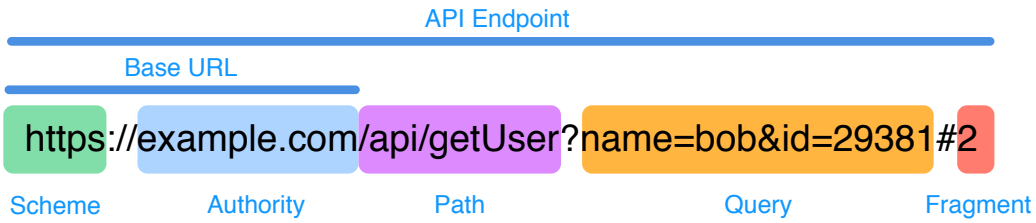


Figure 5.2: Key components of a RESTful API URL

objects, from third party libraries such as *OkHttp* and *Retrofit*, and from reassembled strings. Listing 8 shows a sample output from *Jandrolyzer* after successful URL extraction.

```
Path:
/Users/marc/...
Library:
com.squareup.retrofit
Scheme:
http://
Authority:
retrofiturl.com
Base URL:
http://retrofiturl.com
Endpoints:
    Path: api/loadUsers
    Queries:
        Query key: position, query value: <String>
        Query key: order, query value: <String>
    Fragments:
    HTTP Methods:
        HTTP Method: GET
```

Listing 8: *Jandrolyzer* sample output of a collected URL

5.3.2.1 Strings

A source code snippet whose variable value will be parsed to a "string literal expression" (SLE) is shown in Listing 9. Extracting a URL from an SLE node is straightforward: Whenever an SLE node is found in the AST the value of the SLE is passed to an instance of the `APIURLStrategy` class which contains the logic for verifying URLs. If this class reports a valid URL it is immediately saved for later use.

```
String baseURL = "https://example.com";
```

Listing 9: Example of a "string literal expression"

However, it might occur that the URL we want to detect is composed of multiple other expressions by using a `BinaryExpression` (BE) node, *e.g.*, when developers use the "+" string concatenation operator. A BE example is displayed in Listing 10, in which the base URL of the API is specified in a variable and the endpoint to a specific resource is subsequently added to the base URL by using a BE. In these cases the BE is first passed to an instance of `StringValueExtraction` which uses the `ExpressionValueExtraction` class to reconstruct the contained string value. This class traverses the subtree spanned by the BE and collects all its relevant nodes, *e.g.*, containing variables, literal expressions (integer, string, boolean, null), nested BEs, MCEs, and OCEs. Second, *Jandrolyzer* iteratively resolves all collected nodes and searches for hard coded values associated with them. Finally, all the resolved values are put together.

Back to Listing 10, *Jandrolyzer* would try to reconstruct all possible values that originate from the variable `baseURL` (line 8). For that reason, *Jandrolyzer* would first resolve that variable in the AST and identify it as a parameter. Second, it would look up the parameter's associated method declaration in the AST (line 7) and search for all MCEs that refer to the found method declaration (line 4), before it would analyze the arguments that are passed to those MCEs to ensure the methods match and to extract potential hard coded values. Next, the argument variable in the discovered method declaration would be resolved leading to the base URL (line 3). Finally, the found values would be merged and lead to the fully functional API endpoint `https://example.com/path2`.

The example used only one MCE, *i.e.*, `addPath()`. If multiple MCEs exist *Jandrolyzer* creates a list with all the collected nodes before it inspects them one by one. The MCE search works over class boundaries thus it is not constrained to the AST of one specific class. In case *Jandrolyzer* is not able to find any hard coded values associated with an expression, it tries to resolve its type and attaches the found type information to the string in the format <typeinfo>, *e.g.*, `<STRING>`. The value extraction process from expressions is performed in a best effort manner and does not yet appropriately consider certain constructs available in the Java programming language, *e.g.*, conditional statements and collections.

```
1  public class TestClass {
2      public void createEndpoint() {
3          String baseURL = "https://example.com";
4          String completeURL = addPath(baseURL);
5      }
6
7      public String addPath(String baseURL) {
8          return baseURL + "/path2";
9      }
10 }
```

Listing 10: Example of a "binary expression"

Furthermore, *Jandrolyzer* not only extracts concatenated strings using BEs, but also using the `java.lang.StringBuilder` class, the `java.lang.String.concat()` method, and the `java.net.URL` constructors.

When searching for `StringBuilder` values, *Jandrolyzer* examines if any VD nodes initialize new objects of the type `java.lang.StringBuilder`. The value of the `StringBuilder` object is then extracted with the help of the `ExpressionValueExtraction` class: *Jandrolyzer* would first search the AST for the method or class node containing the `StringBuilder` object in question and determine if the object was created within a method or as a field of a class, *i.e.*, by an instance variable. Depending on where and how the objects are declared, *Jandrolyzer* would scan the AST subtrees of those nodes for MCEs with a `StringBuilder` object as argument, or for their contained `StringBuilder.append()` MCEs. Finally, *Jandrolyzer* would try to resolve and extract the argument expression value for each matched node similar to the BE-based extraction before. Listing 11 shows a code snippet in which two different `StringBuilder` instances are used to construct two different API URLs. *Jandrolyzer* is able to extract both URLs correctly, *i.e.*, `https://exampleone.com/path1/path2` and `https://exampletwo.com/path1/path2`.

```
1  public class TestClass {
2      private StringBuilder stringBuilder = new StringBuilder();
3      private String baseURL = "https://exampleone.com";
4      public void createEndpoint() {
5          StringBuilder stringBuilder = new StringBuilder();
6          stringBuilder.append(baseURL);
7          stringBuilder.append("/path1").append("/path2");
8
9          this.stringBuilder.append("https://exampletwo.com");
10         this.stringBuilder.append("/path1");
11         addPath(this.stringBuilder);
12     }
13
14     public void addPath(StringBuilder stringBuilder) {
15         stringBuilder.append("/path2");
16     }
17 }
```

Listing 11: Code snippet using the `StringBuilder` class

```
1 String baseURL = "http://example.com";
2 String cp1 = baseURL.concat("/p1");
3 String cp2 = cp1.concat("/p2").concat("/p3");
```

Listing 12: Code snippet using string concatenation

Furthermore, *Jandrolyzer* supports the extraction of concatenated strings using the `java.lang.String.concat()` method. The detection technique for the `concat()` method is very similar to the one used for the `StringBuilder` method `append()`, except that the search is performed implicitly, *i.e.*, whenever a node in an AST is traversed for any reason it will be scanned for the `concat()` method at the same time. Listing 12 illustrates the use of the `concat()` method which would be correctly detected as `http://example.com/p1` for line 2 and `http://example.com/p1/p2/p3` for line 3.

Finally, *Jandrolyzer* can extract information from objects of the type `java.net.URL` whose constructors require an URL string.[2] In other words, *Jandrolyzer* is able to detect and extract `java.net.URL` objects instantiated by the following constructors:

- `URL(String spec)`
- `URL(String protocol, String host, int port, String file)`
- `URL(String protocol, String host, String file)`
- `URL(URL context, String spec)`

The expression values used as arguments for the creation of `java.net.URL` objects are extracted as for the previously mentioned `concat()`, `StringBuilder`, and BE-based approaches.

5.3.2.2 Retrofit library

In *Retrofit* implementations the *Jandrolyzer* tool is able to extract HTTP request body data for the *Gson* and *Moshi* JSON converters. *Retrofit* enforces users to define API URLs with Java interfaces (as shown in Listing 13) and the executable implementation of these interfaces is created using a `Retrofit` object as illustrated in Listing 14.

```
1  public interface RetrofitAPIEndpointInterface {
2      @GET("http://example2.com/api/loadNews")
3      Call<Article> loadNews();
4
5      @GET("api/loadUser/{id}")
6      Call<User> loadUser(@Path("id") int userId);
7
8      @GET("api/loadUsers")
9      Call<List<User>> loadUsers(@Query("order") String order,
10             @Query("position") String position);
11
12     @POST("api/createUser")
13     Call<User> createUser(@Body User user);
14 }
```

Listing 13: *Retrofit* API interface example

The interface annotations are used to define and configure base URLs and their related endpoints. For instance, the supported HTTP methods are defined using the annotations `@GET`, `@POST`, `@PUT`, `@DELETE` `@PATCH`, *etc.* Moreover, the provided Java method protocols allow *Retrofit* to map Java objects to specific server responses. The corresponding methods specify the content of the request bodies, specify key-value query pairs, or dynamically rewrite URLs. On the other hand, when instantiating a new `Retrofit`

[2] `https://docs.oracle.com/javase/7/docs/api/java/net/URL.html`

object the base URL and a converter factory must be specified. The converter factory decides about the deserialization strategy applied to HTTP response bodies received from servers.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();

RetrofitAPIEndpointInterface rApiInt
        = retrofit.create(RetrofitAPIEndpointInterface.class);
```

Listing 14: *Retrofit* API implementation example

In our implementation, the `RetrofitStrategy` class performs the extraction of URLs from *Retrofit* code snippets. If MCEs containing the name "create" exist in the AST, an instance of the `Retrofit-Strategy` will start the extraction. As there might exist other MCEs with the same name it will first try to resolve the scope of the found MCEs to assure they represent a valid `Retrofit` type. After a successful resolution and evaluation the `Retrofit` object declaration node, its base URL, and the referenced converter factory are extracted. Next, the interface where the API endpoints are defined is being resolved which leads to the creation of an AST for that interface. The resulting AST is then used to extract the individual endpoints. *Jandrolyzer* supports the extraction of HTTP methods, endpoints, base URL replacements, partial dynamic URL replacements, key-value query pairs, and request bodies.

5.3.2.3 OkHttp library

The *OkHttp* library provides similar functionality to *Retrofit*, however, it uses a different approach. Instead of using interfaces to define the various endpoints it relies on `Builder` methods.

Consequently, HTTP requests are constructed by calling the static method `Builder()` on the `Request` instance. Numerous customization features are available; two major features are provided by the method `url(String url)` which specifies the base URL, and MCEs for the available HTTP methods, *e.g.*, `post(RequestBody body)` which attaches a body to a HTTP `POST` request as exemplified in Listing 15. The argument of the `url()` MCE may be of the type `java.lang.String`, `java.net.URL`, or `okhttp3.HttpUrl`. While the first two types are processed as described in subsubsection 5.3.2.1 the extraction of `okhttp3.HttpUrl` objects is completed using the `OkHttpStrategy` class.

```
MediaType jsonMediaType = MediaType.parse("application/json; charset=utf-8");
RequestBody body = RequestBody.create(jsonMediaType, "{\"name\":\"test\"}");

Request request = new Request.Builder()
    .url(httpUrl)
    .post(body)
    .build();
```

Listing 15: *OkHttp* HTTP `POST` request code snippet

Similarly, assembling URLs with the `HttpUrl` class depends on the `HttpUrl.Builder()` method. Unlike for the data in HTTP header fields, HTTP URLs must bear additional conversion steps to comply with the American Standard Code for Information Interchange (ASCII) character set used in web communication, *i.e.*, control, reserved, some special-, and all non-ASCII characters must be encoded by a preceding "*%*" symbol followed by the corresponding numerical ASCII code represented as two hexadecimal digits. *OkHttp* provides many different set up methods for configuring the scheme, the host, the port, single path segments (for both, encoded and not encoded), multiple path segments (for both, encoded and not encoded), key-value query pairs (for both, encoded and not encoded), fragments, *etc.* Listing 16 shows the construction of a URL by separately adding the scheme, the host, the query pairs, and the path segments denoting the endpoint. Developers using *OkHttp* can deliberately choose to encode only certain segments of the URL, or even to not use any of its URL encoding capabilities. For example, if they would decide to apply the URL encoding by *OkHttp* to the path segment `path/1` containing the reserved character "/", the library would encode that path segment as `path%2F1` containing only valid URL ASCII characters.

```
1 HttpUrl httpUrl = new HttpUrl.Builder()
2         .scheme("http")
3         .host("example.com")
4         .addPathSegment("api")
5         .addPathSegment("path1/1")
6         .addPathSegment("path2")
7         .addQueryParameter("queryname1", "queryvalue1")
8         .build();
```

Listing 16: *OkHttp* `HttpUrl` request code snippet

Jandrolyzer's `OkHttpStrategy` extraction process is triggered when nodes of the type VD or MCEs with the name "build" are detected. Every found `build()` MCE is resolved and verified if it is either one of the types `okhttp3.HttpUrl` or `okhttp3.Request`. *Jandrolyzer*'s feature detection processes the extraction of `okhttp3.HttpUrl` and `okhttp3.Request` snippets independently, because the link between the request and the (URL) definition could be broken due to resolution errors or missing code from incomplete decompilations. Hence, if a `Request.Builder` has been detected the HTTP method, the request body, and the URL are extracted, and if on the other hand a `HttpUrl.Builder` has been detected `OkHttpStrategy` will reconstruct the URL from the supplied nodes. Developers using *OkHttp* further can use an `HttpUrl.Builder` object to create a URL using multiple statements. *Jandrolyzer* detects this configuration by checking VD nodes if they declare `OkHttp.Builder` objects, and if such an object is found it will be extracted using the `OkHttpStrategy` class. The expressions supplied as arguments to the `Builder` MCEs are then extracted using the `ExpressionValueExtraction` class as described in subsubsection 5.3.2.1. As a result, if the scheme is defined using an MCE of type `scheme(scheme)` and the argument is a variable pointing towards a hard-coded string, it will still be accurately extracted.

Jandrolyzer supports the extraction of the URLs, HTTP methods, and JSON bodies from *OkHttp* requests, and it uses the *OkHttp* library to handle all necessary URL encodings. Moreover, *Jandrolyzer* supports the

extraction of `HttpUrl` and `HttpUrl.Builder` objects including schemes, hosts, ports, any form of one or more path segments, any form of query key-value pairs, and fragments.

5.3.3 Data scheme reasoning

For data scheme reasoning, *i.e.*, reasoning about the actual values transmitted in JSON constructs, *Jandrolyzer* supports the most prevalent data conversion libraries based on our manual analysis outlined in chapter 4, *i.e.*, *org.json*, *Gson*, and *Moshi*. Nevertheless, some developers refrained from using any libraries and developed data conversion algorithms themselves, *i.e.*, they programmatically assemble JSON-compliant strings for HTTP request bodies.

The extraction of JSON data requires the inspection of OCE, MCE, SLE, BE, and VD AST nodes; the relevant Java classes are located in *Jandrolyzer*'s `RequestStructureExtraction` package. Whenever *Jandrolyzer* finds a match the following information is collected: the involved library, the path of the corresponding `.java` file, and the JSON data itself. The model classes, *i.e.*, approximated JSON data structure representations, are designed to provide hints for later verification: For every missing value in a reconstructed JSON object, *e.g.*, in a key-value pair, the value's type hint is stored to foster later generation of sound URLs. Listing 17 shows *Jandrolyzer*'s output after a successful JSON object extraction of a Java model class converted to JSON using *Moshi* in line 6.

```
1 Path:
2 .../User.java
3 Library:
4 com.squareup.moshi
5 JSON Object:
6 {"address":{"street":"<STRING>","number":"<NUMBER_INT>"},"name":"Bob"}
```

Listing 17: *Jandrolyzer*'s output for a successful JSON object extraction

Frequently, complex expressions can be only partially traced back to the hard-coded values. Possible reasons are that expressions depend on user input and therefore are unavailable in the source code, or because expressions are declared in an external `.jar` file, respectively another unreachable location. Besides that, one more possibility is that the source code is incomplete due to decompilation errors which break *Jandrolyzer*'s algorithms.

In the next four subsections we explain in more detail the different extraction algorithms for each supported method, *i.e.*, *org.json*, *Gson*, *Moshi*, and *Json strings*.

5.3.3.1 org.json library

The *org.json* library is included in Android and is used to construct JSON objects, therefore the library provides classes such as `JSONArray` and `JSONObject`. The *org.json* example shown in Listing 18

creates a JSON object consisting of four key-value pairs. The `put()` MCE is used to add a key-value pair to the JSON object. The first `String` argument defines the key, and the second argument defines the corresponding value which can be of type `String`, a primitive number, a `boolean`, a `JSONArray` object, `JSONObject.NULL`, or any `JSONObject` object. Furthermore, it is shown how multiple `put()` MCEs can be chained.

```
1 JSONObject jsonObject = new JSONObject();
2 try {
3     jsonObject.put("ojk7", "ojv7");
4     jsonObject.put("ojk8", JSONObject.NULL);
5     jsonObject.put("ojk9", 9).put("ojk9_1", 91);
6 } catch (JSONException e) {
7     e.printStackTrace();
8 }
```

Listing 18: *org.json* code snippet

During the analysis of AST nodes in the `ProjectAnalyzer` class VD nodes will trigger the related extraction strategy defined in the `ORGJSONStrategy` class, which starts the extraction if the declared object is an `org.json.JSONArray` or `org.json.JSONObject`. The `ORGJSONStrategy` class internally uses the *org.json* library to reconstruct the JSON objects found in the source code. After extracting the initialization value of the JSON object or array *Jandrolyzer* starts with its reconstruction. If a `JSONObject` was declared inside a method, *Jandrolyzer* will traverse the subtree of that method's AST node for any operations on that object, *e.g.*, assignment operations, `put()` operations, return statements, and object passings to other methods. On the other hand, if the object was declared in a field, *Jandrolyzer* will search the AST of every reachable method for operations. With the found operations `ORGJSONStrategy` creates for each a result string, *e.g.*, {`"oik7":"ojv7"`}. The expression values used in a `put()` MCE are extracted using the `ExpressionValueExtraction` class as described in subsubsection 5.3.2.1. *Jandrolyzer* further supports the extraction of two implementation peculiarities: i) nested JSON objects, *i.e.*, when objects of the type `JSONObject` or `JSONArray` are set as a key-value pair's value, and ii) objects initialized using a JSON formatted string. After successfully parsing a method a list of strings is returned containing the generated JSON strings. These strings are then one by one validated using the `JSONStringStrategy` class as described in subsubsection 5.3.3.4.

The JSON extraction is performed using a best effort approach and thus neither guaranteed to be correct, nor complete. In more detail, we introduced a threshold value which specifies how many look up iterations are performed at most which reduces *Jandrolyzer*'s memory and CPU load at the expense of incomplete results when analysing exceptionally complex structures. As a result, JSON objects created inside a method have a higher chance of being extracted accurately than in fields.

5.3.3.2 Gson library

The *Gson* library is more flexible to use than *org.json* and provides an additional approach for the JSON object creation: Instead of only creating JSON objects from scratch it enables the conversion of Java objects to JSON objects. The classes of such Java objects are called "model classes" which describe the resources sent and received from and to the web APIs as shown in Listing 19. They are regular Java classes (getters and setters have been omitted for improved readability) with fields for desired information such as for `name`, `id`, `isVip`, `address`, and `secretNote`.

```
public class User {
    private String nameGSON;
    private int id;
    @SerializedName("is_vip_GSON")
    public boolean isVip;
    private Address address;
    private List<Address> otherAddresses;
    public transient String password;
    ...
}

public class Address {
    @Expose
    String name;
    @Expose
    int number;
    String secretNote;
    ...
}
```

Listing 19: Typical *Gson* model classes for a "User" and an "Address" resource

Converting a `User` object into JSON syntax using the *Gson* library is straightforward as shown in Listing 20 where the three required steps are shown: First, a `com.google.gson.Gson` and a model class object, *e.g.*, a `User` object, must be created before the MCE `toJson()` can be called which converts the `User` object to JSON syntax. During this conversion process Java types will be converted to valid JSON types. Accordingly, in Listing 19 the value of `nameGSON` will be converted to a JSON string, the value of `id` to a JSON number and the value of `isVip` to a JSON boolean. Moreover, nested model classes will be converted to nested JSON objects, *e.g.*, `Address`, and collection types to JSON arrays, *e.g.*, `otherAddresses`. Transient fields such as `password` are not included in the JSON representation. However, this can also be achieved with the `@Expose` annotation as seen in the `Address` model class, which indicates fields that should be exposed to the JSON structure while omitting the fields without that annotation. By default, *Gson* uses the variable names for the JSON key-value pair's key names, *e.g.*, `nameGSON`. Nevertheless, to change a specific key name additional annotations can be used. For example, the variable name `isVip` is changed to `is_vip_GSON` in the JSON representation using the `SerializedName` annotation.

```
Gson gson = new Gson();
User user = new User();
gson.toJson(user);
```

Listing 20: *Gson* object creation

Jandrolyzer scans the AST for MCEs with the name “toJson” to detect the use of *Gson* in the source code, which are analyzed by the `MoshiGSONStrategy` class due to the many similarities between the *Gson* and the *Moshi* library. First, the scope of such MCEs is resolved to verify whether it is a valid `com.google.gson.Gson` type before the argument of the MCE itself is analyzed. After detecting the corresponding model class its AST is used to recursively analyze the fields.

```
{
    "address":{
        "name":"<STRING>",
        "number":"<NUMBER_INT>"
    },
    "id":"<NUMBER_INT>",
    "otherAddresses":[
        {
            "name":"<STRING>"
        },
        {
            "number":"<NUMBER_INT>"
        }
    ],
    "is_vip_GSON":"<BOOLEAN>",
    "nameGSON":"<STRING>"
}
```

Listing 21: JSON structure rebuilt from a *Gson* class

Jandrolyzer is able to extract *Gson* model classes by considering `SerializedName` annotations, `Expose` annotations, transient fields, JSON arrays, and nested JSON objects. The successful JSON reconstruction for the `User` object depicted in Listing 19 can be seen in Listing 21.

5.3.3.3 Moshi library

The *Moshi* and the *Gson* library are very similar in terms of features and use, *i.e.*, both are data conversion libraries with comparable syntax which are able to convert JSON formatted text to Java objects and vice versa. Listing 22 shows two typical *Moshi* model classes: A `User` and an `Address` model class representing a `User` resource with the corresponding address resource. The `UserKind` indicator used in the model class `Address` is defined by a Java enum type.

In *Moshi*, a `JsonAdapter` for each model class needs to be created to convert a Java model class object to JSON syntax as shown in Listing 23. As for *Gson*, transient fields are omitted in the JSON representation. Among the very few differences to *Gson* is the use of different annotations. The `@Json` annotation is used to specify a custom JSON key name, *e.g.*, the `isVip` field's key name is changed from `isVip` to `is_vip` for the JSON representation of that object. Furthermore, the by *Moshi* supported enum type values are ultimately transformed into JSON string values.

```
public class User {
    public String name;
    public int score;
    @Json(name = "is_vip")
    public boolean isVip;
    public UserKind userKind;
    public Address address;
    public List<Address> altAddresses;
    private transient String password;
}

public class Address {
    public String street;
    public int number;
}

public enum UserKind {
    PROUSER,
    NONPROUSER
}
```

Listing 22: Typical *Moshi* model classes for a "User" and an "Address" resource

During the AST scan *Jandrolyzer* will search for MCEs named "adapter" to extract the scheme of *Moshi* model classes. The extraction of such model classes is achieved with the same *MoshiGSONStrategy* class as for the *Gson* extraction due to the aforementioned analogies. After detecting such an MCE, its scope is resolved to validate if it is a valid `com.squareup.moshi.Moshi` type. The argument of the MCE directly leads to the model class for which an AST is created. This newly created AST is then traversed to analyze the fields of the model class. *Jandrolyzer* supports the extraction of *Moshi* model classes by considering `@Json` annotations, transient fields, JSON arrays, nested JSON objects, and enum types. The successful JSON reconstruction for the `User` object depicted in Listing 22 can be seen in Listing 24.

```
Moshi moshi = new Moshi.Builder().build();
JsonAdapter<User> jsonAdapter = moshi.adapter(User.class);
User user = new User();
String json = jsonAdapter.toJson(user);
```

Listing 23: *Moshi* object creation

```
{
    "address":{
        "number":"<NUMBER_INT>",
        "street":"<STRING>"
    },
    "score":"<NUMBER_INT>",
    "altAddresses":[
        {
            "street":"<STRING>"
        },
        {
            "number":"<NUMBER_INT>"
        }
    ],
    "name":"<STRING>",
    "userKind":"<STRING>",
    "is_vip":"<BOOLEAN>"
}
```

Listing 24: JSON structure rebuilt from a *Moshi* class

5.3.3.4 JSON strings

JSON encoded text can be created, manipulated, and stored with plain `String` expressions, *i.e.*, an SLE node in the AST. We mostly encountered this so called "primitive obsession" code smell for short HTTP request bodies where developers were not eager enough to use a library.

During the analysis of AST nodes in the `ProjectAnalyzer` class SLEs are first searched for valid URL data. If `APIURLStrategy` was unable to extract any URL data, the very same node is also checked for potential JSON data using the `JSONStringStrategy` class which also removes the found string's escape sequences. Next, the `JSONDeserializer` class is used to deserialize the found strings into `JSONRoot` objects. The process is aborted, if the strings do not contain any valid JSON data. On the other hand, if `JSONDeserializer` is able to deserialize the string it creates a new `JSONRoot` object which is assigned to the corresponding project.

For JSON formatted text composed of multiple expressions within a BE the extraction strategy it is the same as for the extraction of URLs from BEs explained in subsubsection 5.3.2.1. In fact, the BE is only analyzed once using the `StringValueExtraction` class. The extracted strings are first scanned for URLs using the `APIUrlStrategy` class, and if no URLs are found the very same strings are processed by the `JSONStringStrategy` class to search for valid JSON strings. Similarly, JSON strings constructed with the `StringBuilder` class or the `java.lang.String.concat()` method are searched for valid URLs, and if none is found they are searched for valid JSON data.

```
http://example.com/api/registerDevice&api_key=<STRING>
```

(a) partially reconstructed URL

```
{
    "accessKey":"<STRING>"
}
```

(b) partially reconstructed JSON scheme

Figure 5.3: Missing query values in reconstructed data

5.4 URL validation

After analyzing all the `.java` files of a project, the extracted URLs and JSON data are saved inside the corresponding `Project` object. *Jandrolyzer* provides functionality to test URLs with different HTTP method and request body combinations to validate the collected data. The code for the data preparation and validation is located in the `APIAnalysis` package of the project.

5.4.1 Data preparation

Before conducting any validation, the collected data must be prepared first, because query pairs in URLs and JSON data are often incomplete as illustrated in Figure 5.3a for partially extracted URLs and in Figure 5.3b for partially extracted JSON objects.

With the help of the collected type information *Jandrolyzer* will fill in type-sensitive values, *e.g.*, numbers and boolean values are set to `0` and `true`, respectively. The processing of missing `String` values is more complex; they are populated with actual `String` values found in the application. More precisely, *Jandrolyzer* detects string variable declarations and collects the variable names and the assigned values with the help of the `VariableCollector` class. Next, this class compares the key name of each incomplete key-value pair to the collected variable names. The value of the variable name that is most similar to the key name is used as source for the missing value. We used the *Jaro-Winkler* string similarity distance metric to compute the distance between a key and a variable name which has been implemented in the *java-string-similarity* library.[3] This algorithm is intended for comparisons of human entered short text strings such as variable names, because it is exceptionally robust against letter transpositions and it is assigning more weight to the similarity at the beginning of the strings. The resulting similarity value is in the range `[0, 1]`; 1 indicates the closest match of two strings, and 0 indicates that the two strings have no similarities at all. For instance, the `Jaro-Winkler` similarity between the two string variable names “apiKey” and “api_key” is 0.8944444715976715. Consequently, *Jandrolyzer* would insert the value of the

[3] `https://github.com/tdebatty/java-string-similarity`

"apiKey" variable into the JSON data if no better match has been found. Finally, the completed URL and the JSON data will be saved to disk.

5.4.2 Queries

The code for the testing procedure is located in the `APIAnalyzer` class and supports besides the non-altering `GET` the HTTP methods `PUT`, `PATCH`, and `DELETE` that per definition are supposed to alter or delete remote resources located on web servers. Depending on how *Jandrolyzer* was configured only specific HTTP methods are tested. Users must proceed with caution; they are advised to disable the altering HTTP methods for tests with third party applications to avoid any potential harm to app developers.

First, the files containing the prepared data are parsed and the URLs and JSON data are collected. Next, *Jandrolyzer* tests each endpoint based on the preset HTTP methods and includes the JSON data in the HTTP request body. The requests are executed using the *OkHttp* library.

5.4.3 HTTP response headers

Jandrolyzer yields the HTTP server responses from the tests outlined in subsection 5.4.2. The collected data is saved to a file for later analysis and includes information like the *URL*, the *JSON string*, the *HTTP method*, the *status code*, a *success indicator*, the *success or error message*, the *response headers*, and the *response content*.

More precisely, *Jandrolyzer* stores the following components: i) The used *URL* which describes the API endpoint. It is stored for later replayability of the results. ii) The used *JSON string* which contains the reconstructed JSON key-value data pairs. It is stored for later replayability of the results. iii) The used *HTTP method* which states how the request was treated. It is of the type `GET`, `PUT`, `POST`, `PATCH`, or `DELETE`. This method in combination with the request URL and the JSON string allows researchers to resubmit the request for further analyses. iv) The received *status code* which informs the client about the current server state. It is an integer, *e.g.*, 200, and serves as an indicator whether the server understood the request, *i.e.*, the request has succeeded. v) The *success indicator* is represented by a boolean value and shows if the corresponding request was successful. This value includes client-side errors (*e.g.*, in case of a malformed URL) as well as the response from the server. vi) The received *success or error message*, *e.g.*, "OK", which relates to the status code. Based on this message *Jandrolyzer* decides if the recreated URL is valid. For example, if the error message "Not Found" is replied or a time out occurs, *Jandrolyzer* assumes that either the URL is invalid or the server offline. vii) The received *response headers* which are a set of key-value pairs. They provide additional information regarding the server configuration. A few standardized response headers exist, *e.g.*, `Allow`, `Cache-Control`, `Content-Type`, `Date`, `Content-Length`, and `Server`. However, the server is not required to send any of these headers. Headers could be used for further security assessments, *e.g.*, to check if the endpoint server software is outdated and suffering from known exploitable security vulnerabilities. viii) The received *response content*

which contains the requested (user) resource from the server. When sending a request to an API endpoint it generally can be assumed that the response uses the same encoding (*e.g.*, UTF-8) and language (*e.g.*, JSON). The response content could disclose potential web API data leaks (*i.e.*, by containing variables with sensitive information), denial-of-service attack vectors (*e.g.*, by containing range arguments for data retrieval), and other security issues.

6 Analysis

In this chapter we present the findings from our analysis of 1 247 applications using the *Jandrolyzer* tool. First, we characterize the two datasets we used, before we explain our findings regarding the apps' use of web communication and the differences in web communication between open-source and closed-source apps. Finally, we deal with server responses received from our requests and highlight a few security risks we have encountered. More concretely, section 6.2 responds to $\mathbf{RQ}_1$, section 6.3 to $\mathbf{RQ}_2$, subsection 6.3.3 to $\mathbf{RQ}_3$, and finally, section 6.4 to $\mathbf{RQ}_4$.

6.1 Datasets

In this section we characterize the two datasets we have used in our analysis, one containing open-source projects from *F-Droid*, hosted on *GitHub*, and one containing closed-source applications from *Google*'s *Play* store.

6.1.1 Open-source apps

Our open-source dataset relies on *F-Droid* software repository which hosts open-source Android projects whose source code is available on code sharing platforms like *GitHub*. The corpus initially contained all

the projects found in *F-Droid*'s index file,[1] but we removed all duplicates (same package identifier, but different version numbers) by striving for the most recent version, and by removing all projects that did not specify the use of the `android.permission.INTERNET` permission in the manifest file which is required for any web communication. In the end, we performed the analysis on a corpus of 413 open-source projects. From these 413 apps 68.5% (283 apps) could be successfully analyzed by *Jandrolyzer*, whereas the remaining apps either exceeded the time constraints or crashed the tool, thus yielded no results. Only about 29.5% of the (deduplicated) apps require the `android.permission.INTERNET` permission.

6.1.2 Closed-source apps

Our closed-source dataset relies on random free apps, respectively free apps with in-app purchases, from the *Google Play* store. The store only provides applications in the `.apk` file format which requires decompilation to source code prior to the analysis. The corpus initially contained 889 apps, but we similarly removed all applications that did not specify the use of the `android.permission.INTERNET` permission in the manifest file. In the end, we performed the analysis on a corpus of 834 closed-source apps. Out of these 834 apps 40.1% (335 apps) could be successfully decompiled, and from those 49.8% (167 apps) could be successfully analyzed by *Jandrolyzer*. The remaining apps either exceeded the time constraints or crashed the tool, thus yielded no results. Unlike open-source apps, an enormous 94% of the applications require the `android.permission.INTERNET` permission. This result confirms the sentiment that the majority of the Android closed-source applications utilize the internet to display information and ads using web views or to synchronize data using (web) servers.

6.2 Manual analysis

First, we manually examined 160 random open-source apps for used libraries to gather information about their prevalence. This information was then used to prioritize the implementation of *Jandrolyzer*'s library support; the most prevalent libraries became implemented first. We encountered nine different HTTP frameworks, *i.e.*, *Android Volley*, *AndroidHttpClient*, *AsyncHttpClient*, *glide*, *HTTP Components*, *ion*, *java.net*, *OkHttp*, and *Retrofit*. *Jandrolyzer* supports network snippet extraction for all of them and URL extraction for the most prevalent ones, *i.e.*, *java.net*, *OkHttp*, and *Retrofit*. Moreover, we found three different JSON libraries, *i.e.*, *Gson*, *Moshi*, and *org.json*. Since all three libraries have been frequently used in the open-source projects we added support to *Jandrolyzer* for all of them. Besides these JSON libraries we rarely encountered two different XML-based libraries, namely the Java built-in *org.xml.sax.XMLReader* and the *XML Pull Parser*. However, only few projects relied on these, thus we neglected them in our tool.

[1] `https://f-droid.org/repo/index.xml`

6.3 Automated analysis

In this section we present the findings from our automated analysis of 413 open-source and 834 closed-source applications using the *Jandrolyzer* tool. We focus on the use of the libraries and the use of the URLs and JSON schemes for open- and closed-source apps.

6.3.1 Open-source apps

All results presented in this subsection are based on the successful analysis of 283 open-source apps by *Jandrolyzer*.

6.3.1.1 Use of libraries

Figure 6.1 shows the libraries used in open-source projects. The `URLConnection` (37.4%), `HttpURLConnection` (24.3%), `Socket` (9.1%), and `HttpURLConnection` (6.0%) classes included in *java.net* are the preferred choice of open-source developers, especially `URLConnection` and `HttpURLConnection` are omnipresent in projects. When considering third party network libraries *OkHttp* and *Retrofit* (each 5.6%) are used the most. It is interesting to see that libraries with specific support for image downloads are similarly used, *i.e.*, *glide* and *Volley*. The network library *ion* is the least used with only three occurrences in total (1.0%). *Jandrolyzer* found neither any `SSLSocket`, nor any `android.net.http.AndroidHttpClient` instances. Additionally, we observed that open-source projects use zero to four network libraries. While we did not expect the reluctant use of libraries, one explanation could be that many developers just use boilerplate code for rather small tasks.

6.3.1.2 Use of URLs and JSON schemes

Across the open-source projects *Jandrolyzer* was able to extract 1 533 base URLs and 458 JSON schemes. The vast majority of the endpoint URLs based on the extracted base URLs were requiring at least one type of parameter(s), *e.g.*, one or more path segments (1 182, 98.8%), queries (240, 20,0%), and fragments (11, 0,9%). Figure 6.2 illustrates the extracted endpoint data. The x-axis denotes the unique elements in an endpoint, *i.e.*, how many elements of a specific type (fragments, path segments, queries) the endpoints require. For instance, the first column "1" represents the number of found endpoints that demand one fragment, one path segment, or one query. The chart shows that the majority of endpoints have one or two fragments, path segments, or key-value query pairs. We can further see that 209 endpoints exist with paths consisting of four or five path segments to distinguish between resources. Nevertheless, endpoints using more than five elements are rare. Additionally, we can identify up to 18 unique query pairs used in one specific endpoint, and that URL fragments are seldom used in endpoints; we only found up to one fragment per endpoint. Based on these findings it appears that most APIs are just built for one specific

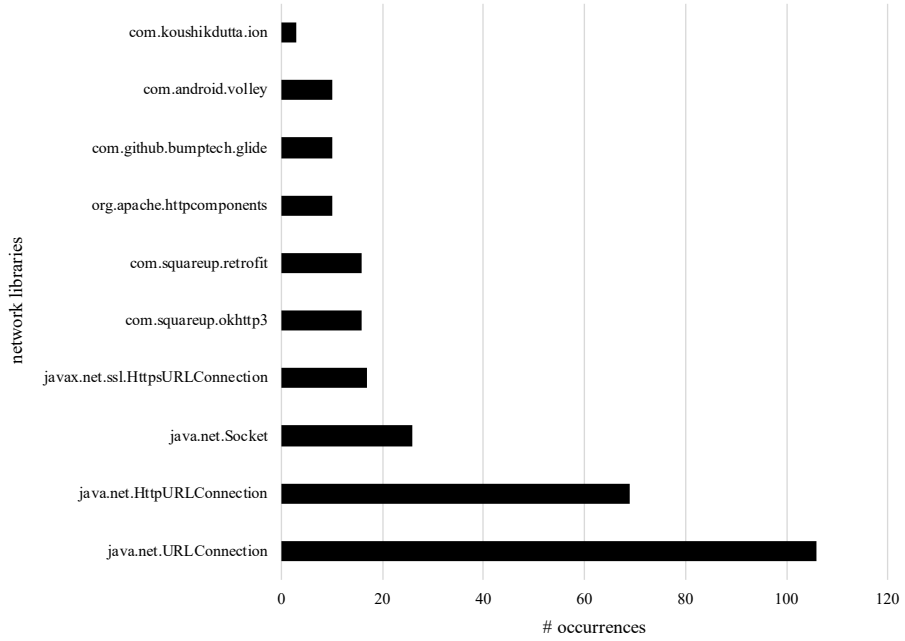


Figure 6.1: The use of different network libraries in open-source projects

purpose and thus not in need of many arguments. This is encouraging because the chances of generating a valid request with only few parameters are rather high.

We also evaluated the most common base URLs. The most used single base URL was `https://github.com` which has been observed 29 times (10.2%). Likewise *Google* services have been widely used, *e.g.*, `https://play.google.com` or `https://plus.google.com`; the tool could spot 42 instances (14.8%). Rather at the end of the ten most commonly used based URLs *Jandrolyzer* found the *OpenWeatherMap* API `http://openweathermap.org` (7, 2.4%) and the *Twitter* social network API `https://twitter.com` (6, 2.1%). We were surprised to see such a dominance of *GitHub*. However, the use of *Google* APIs is not surprising at all; many of *Google*'s own services, *e.g.*, *Google Analytics* or *Google Play* in-app purchases, rely on them. Another interesting observation is that the open-source community prefers *Twitter* over *Facebook*, which is one of the world's largest social networks.

Figure 6.3 depicts the URL schemes and the JSON value types found by our tool. The plot reveals that the `https` URL scheme (1 012 occurences, 66.0%) is much more commonly used than its insecure counterpart `http` (521 occurences, 33.9%). We can further see that `STRING` is the most used JSON value type with 1 197 occurrences, followed by `NUMBER` with only 234 occurences. It seems that the JSON value types `BOOLEAN` and `NULL` are barely used with only 130 and 14 occurrences, respectively. We conjecture that this is due to the nature of the requests: Most requests are not performing any calculations on numbers, but work on user data which is mostly consisting of strings, *e.g.*, addresses, names, locations, *etc.*

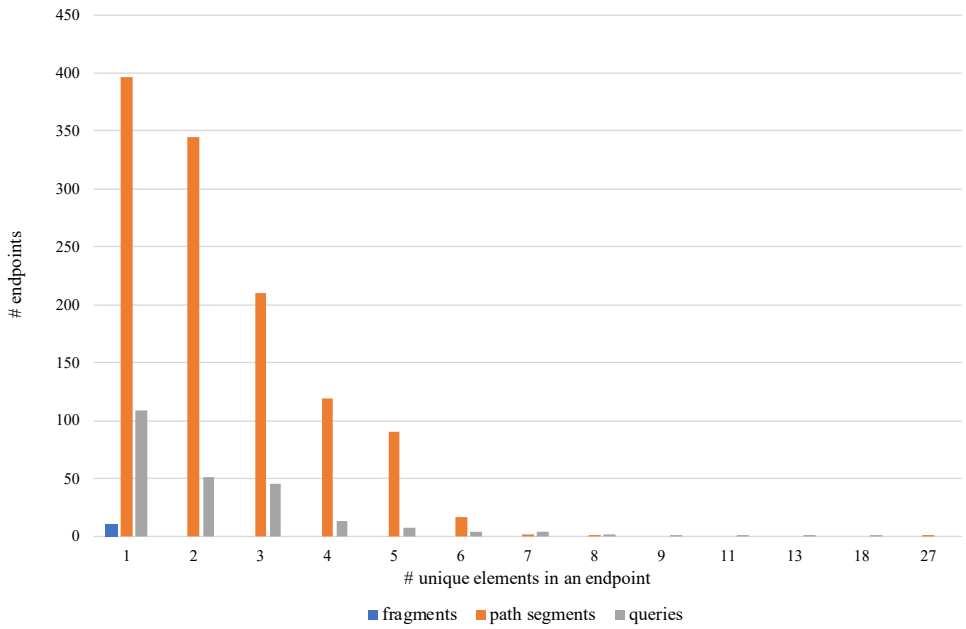


Figure 6.2: The components used in web requests of open-source apps

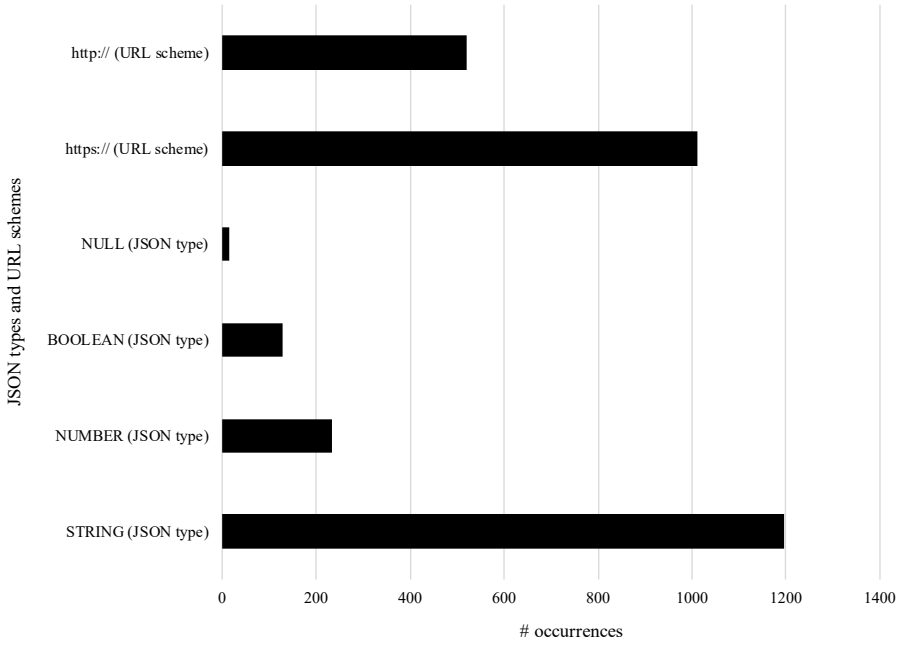


Figure 6.3: The used JSON value types and schemes in open-source applications

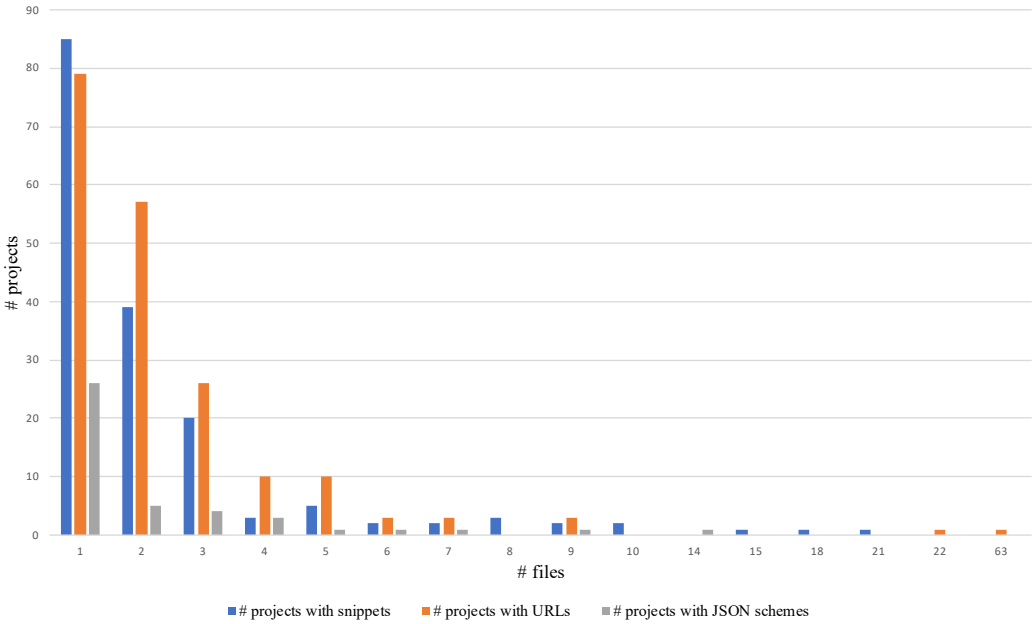


Figure 6.4: The file-spread of open-source apps' web communication facilities

Besides the URL schemes and JSON value types, we also collected the most frequent query keys and JSON key-value pairs. We found the three most used keys to be `id`, `category`, and `setting`, and we found less common keys to be `sel_start`, `sel_end`, `expression`, `precise`, `type`, `q`, and `name`. Depending on the server-side implementation the values corresponding to these keys could be easily guessed client-side to access arbitrary resources on the web server, because many of them are intentionally limited in scope (*e.g.*, `id`, `end`). For example, if the web server does not authorize each client appropriately it could mistakenly access a protected resource with a manipulated numeric `id`. In addition, the query and JSON values could be used for code injection attacks (*e.g.*, using `expression`) with the same result. Obfuscation of such interface declarations could mitigate the trivial guessing of parameters and should be integrated into the regular workflow of web developers.

Figure 6.4 illustrates the number of different files contributing to code snippets, URLs, and JSON schemes. The x-axis denotes the number of unique files in an app on which the web communication relies, *i.e.*, how many files in an app have been used for the extraction of code snippets, URLs, and JSON schemes. For instance, the first column "1" represents the number of found apps that rely on one code snippet, one URL, or one JSON scheme for web communication. We can see that in most projects the network related functionality is spread among one or two files. We observed in an app at most 21 files containing network code snippets, 63 containing URLs, and 14 containing JSON schemes. We conclude that in most projects only one or two classes are responsible for network communication. On the other hand, the further end of our findings indicates bad software design practices due to duplicated network-related code used in different components of an app.

6.3.2 Closed-source apps

All results presented in this subsection are based on the successful analysis of 167 closed-source apps by *Jandrolyzer*.

6.3.2.1 Use of libraries

Figure 6.5 shows the libraries used in closed-source apps. We can see that the classes included in *java.net* such as `java.net.URLConnection` (42.3%), `java.net.HttpURLConnection` (35.0%), `java.net.Socket` (6.8%), and `java.net.ssl.HttpsURLConnection` (3.8%) are the preferred choice. If we treat both *OkHttp* library releases as one item it would represent the most used third party library even surpassing the well known *glide* and *Retrofit* libraries. `org.apache.httpcomponents` and the asynchronous HTTP library `com.loopj.android` are the two least used network libraries with a contribution of only 1.7% and 0.4%, respectively. During our analysis we observed that closed-source apps use zero to seven different network libraries. It is interesting to see that two different releases of the same library have been used, *i.e.*, *OkHttp* releases <= 2 and 3. This indicates that many developers do not populate code repositories with library updates which could greatly mitigate potential security issues introduced by the use of outdated libraries. The heavy use of *java.net* code is surprising, however, the result presumably has been biased due to the greedy decompilation process: While decompiling the apps, the bundled libraries are decompiled in the same manner as the apps. For this reason, our analysis considers not only the in-app code, but also the library code which eventually calls the system's Java APIs. This is not the case for the open-source analysis which is built on the Gradle build system to manage dependencies. With this system, the actual library source code is not yet included at the time of our static analysis.

6.3.2.2 Use of URLs and JSON schemes

Across the closed-source apps *Jandrolyzer* was able to extract 621 base URLs and 705 JSON schemes. The vast majority of the endpoint URLs based on the extracted base URLs were requiring at least one type of parameter(s), *e.g.*, one or more path segments (1 110, 99.4%), queries (187, 16.7%), and fragments (4, 0.3%). Figure 6.6 illustrates the extracted endpoint data. The x-axis denotes the unique elements in an endpoint, *i.e.*, how many elements of a specific type (fragments, path segments, queries) the endpoints require. For instance, the first column "1" represents the number of found endpoints that demand one fragment, one path segment, or one query. The chart shows that the majority of endpoints have one or two fragments, path segments, or key-value query pairs. On a second look, one can see that two path segments are the most used configuration in request URLs, which is inverse to the trend for fragments and queries. We can further see that 105 endpoints exist with paths consisting of four to eight path segments to distinguish between resources. Nevertheless, endpoints using more than four elements are rare. Additionally, we can identify up to 10 unique query pairs used in one specific endpoint, and that

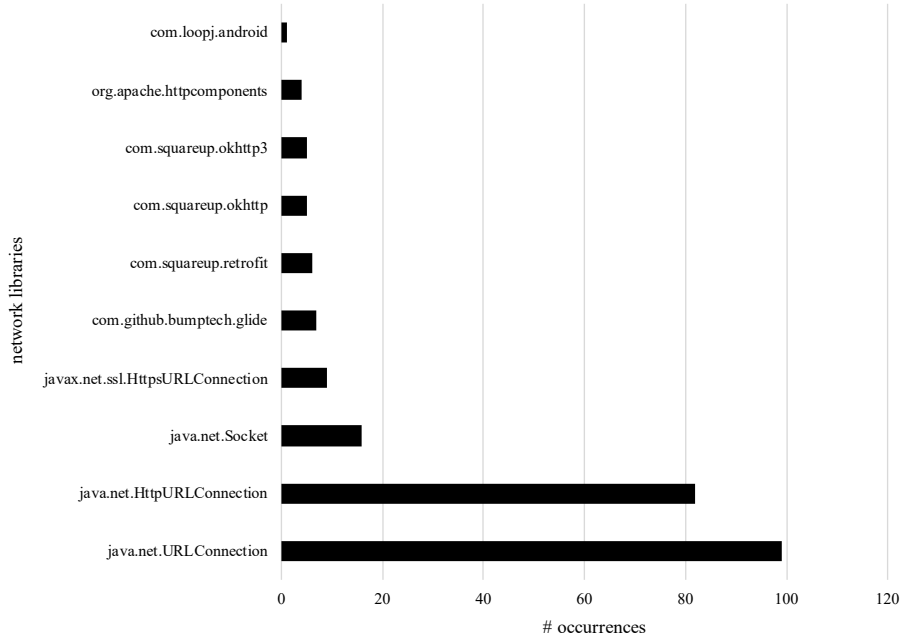


Figure 6.5: The use of different network libraries in closed-source apps

URL fragments are seldom used in endpoints; we only found up to one fragment per endpoint. Based on these findings it appears that most APIs are just built for one specific purpose and thus not in need of many arguments. This is encouraging because the chances of generating a valid request with only few parameters are rather high.

We also evaluated the most common base URLs. The most used single base URL was `http://schemas.android.com` hosted on *Google*'s servers which has been observed 75 times (33.0%). In fact, all of the ten most used base URLs were pointing towards *Google* services, *e.g.*, *Google Analytics* and *Google Play*. Additionally, many of these *Google* services relate to the distribution of ads such as `http://media.admob.com` (19, 8.3%), `https://pagead2.googlesyndication.com` (16, 7.0%), and `http://e.admob.com` (12, 5.2%).[2] We were surprised to see such a dominance of *Google*, however, we expect that the "Freemium" price model, *i.e.*, installation of apps is free but the user must later watch ads or pay a fee, is a major enabler of this setting.

Figure 6.7 depicts the URL schemes and the JSON value types found by our tool. The plot reveals that the `http` URL scheme (378 occurences, 60.8%) is much more prevalent than its secure counterpart `https` (243 occurences, 39.1%). We can further see that `STRING` is the most used JSON value type with 1 654 occurrences, followed by `BOOLEAN` with only 297 occurences. `NUMBER` and `NULL` only represent a minority with 163 and 72 occurrences, respectively. The prevalence of `http` URL schemes is devastating

[2] *Google AdMob* is a popular ad platform which provides developers SDKs for the eased integration of *Google* ads into their own apps to increase their revenue.

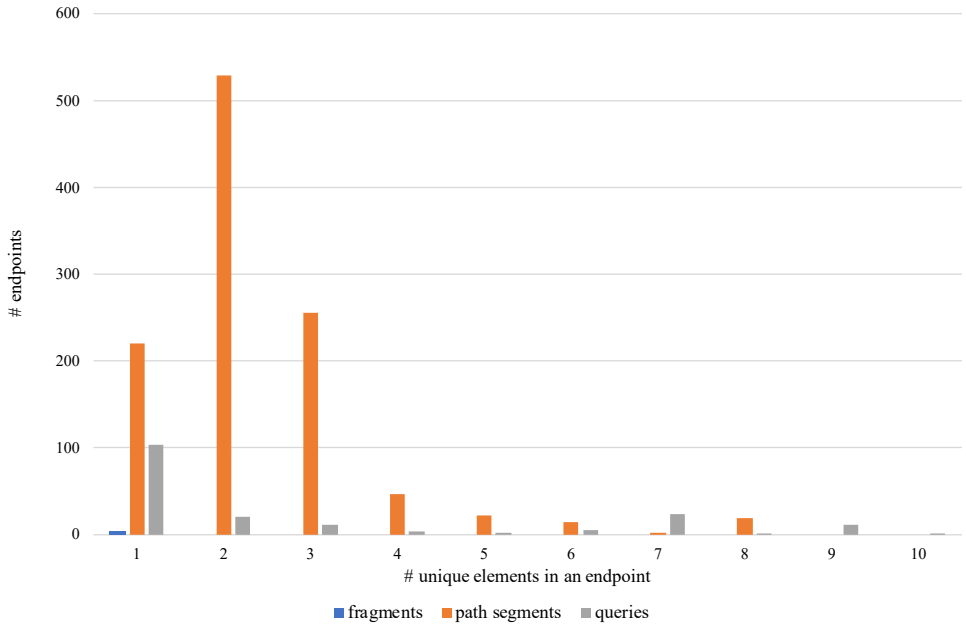


Figure 6.6: The components used in web requests of closed-source apps

and should be avoided at all cost. Insecure connections present a severe risk for data leakage, manipulation, or Denial-of-Service (DoS) attacks. For instance, it is trivial to replace range values in plain text web requests from server-side trusted and authorized users to not only retrieve more potentially sensitive information from a user. This measure also causes much more server load which could lead in extreme cases to major service outages.

Besides the URL schemes and JSON value types, we also collected the most frequent query keys and JSON key-value pairs. We found the three most used keys to be `id`, `event`, and `type`, and we found less common keys to be `value`, `valueType`, `enumType`, `what`, `session_id`, and `js`. The threats outlined in subsubsection 6.3.1.2 also apply here. In particular, the `session_id` could be susceptible to session hijacking attempts if not implemented properly on both sides, *i.e.*, the client application and the server. That is, an eavesdropper could intercept the session id and send further requests to the server who would still believe that those are originating from the legitimate sender. Moreover, the `js` key very likely further increases the attack surface. If the client adds malicious *JavaScript* code to the corresponding key the server could be receiving and executing that code and thus be susceptible to various injection attacks. In more technical terms, if the *JavaScript* code sent by the client is server-side passed to the `eval()` function this would enable arbitrary code execution on the server itself.

Figure 6.8 illustrates the number of different files contributing to code snippets, URLs, and JSON schemes. The x-axis denotes the number of unique source files in an app on which the web communication relies, *i.e.*, how many files in an app have been used for the extraction of code snippets, URLs, and JSON schemes.

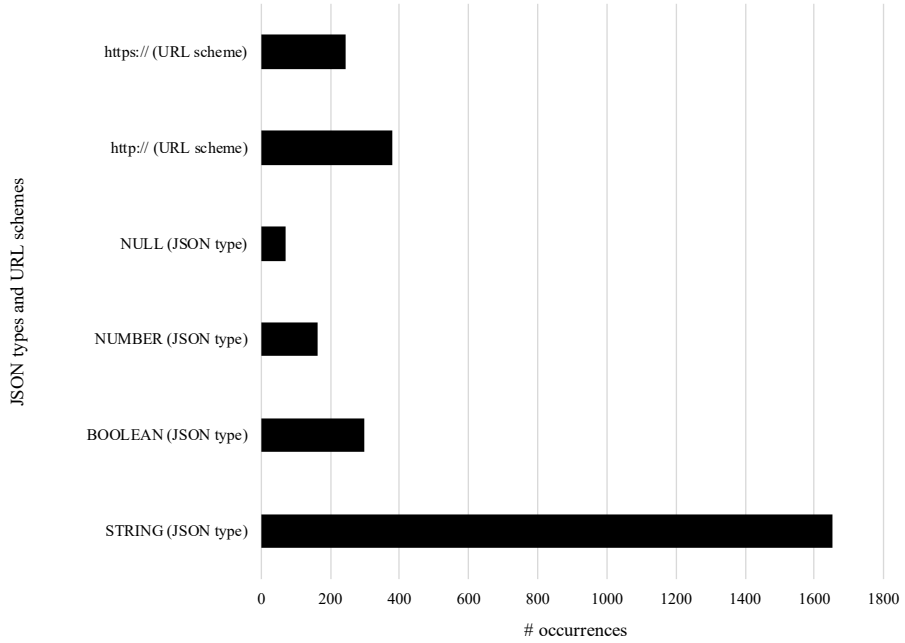


Figure 6.7: The used JSON value types and schemes in closed-source applications

For instance, the first column "1" represents the number of found apps that rely on one code snippet, one URL, or one JSON scheme for web communication. We can see that in most apps the network related functionality is spread among one to two files. We observed in an app at most 29 files containing URLs, 20 containing JSON schemes and 15 containing network code snippets. We conclude that in most apps only one or two classes are responsible for network communication. On the other hand, the further end of our findings indicates bad software design practices due to duplicated network-related code used in different components of an app. Generally, in the closed-source dataset it seems that the network code snippets are spread across more files. One possible reason could be that the closed-source apps suffer from worse software design. Another reason could be that the analysis of closed-source apps frequently included source code from decompiled bundled third party libraries which might have contained network-related source code.

6.3.3 Differences

Libraries. We found more *java.net* libraries (*i.e.*, `java.net.URLConnection`, `java.net.HttpURLConnection`, `java.net.Socket` and `java.net.ssl.HttpsURLConnection`) used in closed-source apps compared to open-source projects. However, this might have been caused by the more thorough analysis of closed-source apps where the source code of bundled libraries has been included in the analysis due to the non-trivial separation of library code. Nevertheless, the *java.net* network libraries

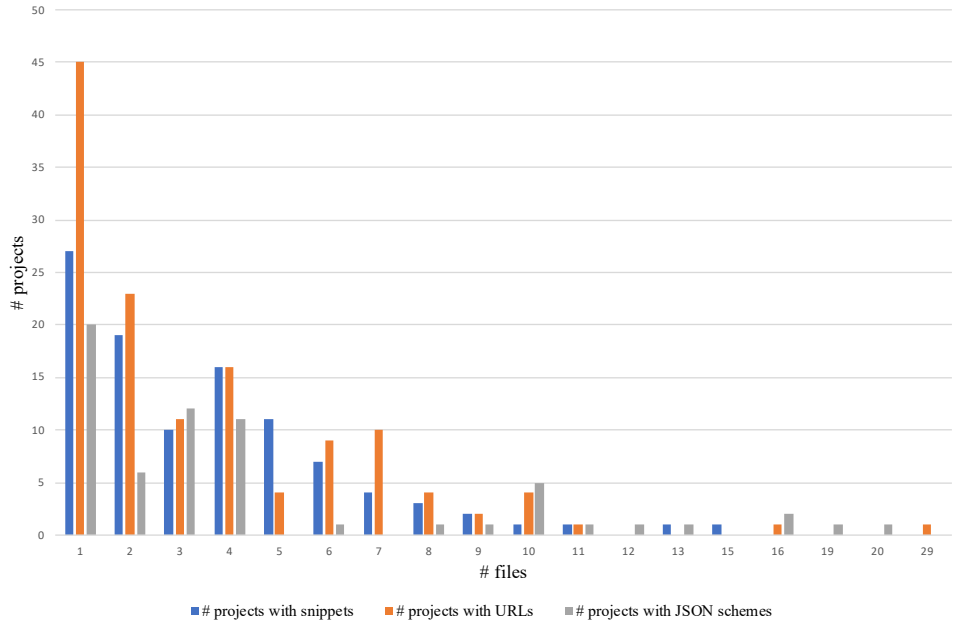


Figure 6.8: The file-spread of closed-source apps' web communication facilities

found in open-source projects still appear to be the preferred choice even though the source code of the libraries did not bias the results in these projects. Interestingly, closed-source apps use different releases of *OkHttp* while open-source apps only use more recent releases. Furthermore, the libraries *ion* and *Volley* have only been used in open-source apps, while *HttpComponents* and *LoopJ* have only been used in closed-source apps. Surprisingly, we did not find any instances of the well-known `AndroidHttpClient` and `SSLSocket` classes. Finally, we can see that the use of *glide*, which supports exhaustive image downloading and caching features, is much more prevalent in closed-source apps.

Requests. Open-source apps relied on simpler request paths including only one or two path segments, while closed-source apps mostly included two or three path segments. Unexpectedly, the opposite is true for key-value pairs: Open-source apps frequently use one to three pairs, while closed-source apps majorly use one pair. Fragments have only been used very sparsely in both datasets.

Base URLs. While the open-source apps contained no ad services in the ten most used base URLs, the closed-source apps heavily used ad services to increase their revenue. Besides those advertisement items, we only found one major difference: Open-source apps tend to use *GitHub* and *OpenWeatherMap* services, but closed-source apps primarily use different *Google* services.

JSON value types and URL schemes. We could observe no major differences in the used JSON value types; both show similar uses with closed-apps slightly prefering the `boolean` over the `number` type. However, we found one major difference in the used URL schemes. Open-source apps principally rely on secure `https` connections (66.0%), which is in stark contrast to the closed-source apps that largely use

the insecure `http` protocol (60.8%).

6.4 Validation

Jandrolyzer is able to send requests based on the reconstructed URLs and JSON schemes in the message body, if applicable, by using the HTTP methods `GET`, `PUT`, `POST`, `PATCH`, `DELETE`, or a subset thereof. The data received from the server, *e.g.*, the response message, response headers, and the status code is saved to a file for later analysis. However, because there is no guarantee that the app developers respected the REST architectural style guidelines, we instead decided to validate the results manually to prevent any accidental damage on remote systems. For example, it is very likely that certain apps allow the deletion of resources on the server by HTTP `GET` requests instead of HTTP `DELETE` requests which we cannot easily foresee by any automation.

6.4.1 Manual endpoint validation

For our manual endpoint validation we randomly selected 25 open-source and 25 closed-source apps for which we reviewed the analysis results generated by *Jandrolyzer*. We reviewed the collected snippets, URL endpoints, JSON schemes, and variables. After we ensured that no harm could be done, we manually sent requests to the collected endpoints using the *Postman* tool.[3] We tried different HTTP methods and message bodies using the collected JSON schemes.

The manual endpoint validation yielded very pleasing results: More than 90% of the endpoints successfully replied to our requests. The few endpoints that were not responding suffered either from incomplete URLs (*i.e.*, `https://example.com/api/<STRING>/users`), or from IP addresses which were presumably offline at the time of our investigation.

We collected several interesting findings while validating the endpoints.

One finding is that the use of the status code is not consistent across different endpoint servers which could lead to severe confusion among developers using the API. For example, most web servers correctly send the status code `200` to indicate a successful request, but some erroneously send this status code when an error has occurred and then use the message body to specify the error message in JSON. This is against any good practice of implementing a RESTful service and an indicator for misunderstood architectural concepts. Furthermore, developers misunderstanding API architecture might also cause reliability and safety issues due to their lack of knowledge.

A second finding is that most API endpoints reveal information about the server configuration in the response headers, *i.e.*, the frequently sent response header `Server` often reveals the server name, the operating system of the server, and its version number. In our experiments we found the most prevalent

[3] `https://www.getpostman.com/tools`

endpoint servers to be run on *Apache* and *Nginx*. Moreover, some servers also leaked information about the API's backend by returning the response header `X-Powered-By`. Server responses containing this property disclose potential security vulnerabilities present in the backend, *e.g.*, by returning `X-Powered-By: PHP/7.3.3` the interrogator knows that the server-side API implementation is based on PHP release 7.3.3.

Other findings include that many apps show web content inside a web view, *i.e.*, a web browser component within the application, and that many apps use the `Server` header as well as other non-standardized HTTP headers. Finally, we can confirm that ad services are much more prevalent in closed-source than in open-source apps.

6.4.2 Discovered security issues

Server-side disclosure of running applications. Information about the web server, its version number, the used operating system and programming languages are detrimental to the security of the API endpoint. During our validation we observed one server returning the `X-Powered-By: PHP/5.5.23` response header even though the current PHP release at that time was `7.3.3`. By considering the Common Vulnerabilities and Exposures (CVE) database we found that this application framework suffers from 57 known security vulnerabilities.[4] The vulnerabilities range from simple DoS attacks, access control taming, and cross-site scripting to arbitrary code execution on the server. Most of the web server daemons and API implementation frameworks we validated were outdated with known security vulnerabilities.

Server-side disclosure of API code. We found multiple API endpoints leaking internal error messages. In most cases the actual error message including the corresponding stack trace has been transmitted as plain text in the server's message response body. Consequently, the error messages included information about method names, line numbers, and file paths revealing the internal file system structure of the server. By applying trial and error techniques to search for security issues an intruder could gain further knowledge about the endpoint's inner workings to illegitimately access the system.

Server-side use of insecure HTTP. Loading web content in web views without securing the communication with HTTPS makes the application vulnerable to Man-in-the-Middle (MITM) attacks which could be used to steal information, or to allow client-side malicious code execution.

Server-side shell access. One application used the request message body to issue shell commands on the server which directly leads to arbitrary code execution on the server. User input must *never* be trusted and should be server-side validated to prevent potential code injection attacks.

Server-side lack of authentication and authorization. We found several intentionally private APIs that were completely lacking any authorization mechanisms. These APIs were leaking information, *e.g.*, location data, room occupancy, sensor readings, and even allowed the creation of new users on a server.

[4] `https://www.cvedetails.com/vulnerability-list/vendor_id-74/product_id-128/version_id-183021/PHP-PHP-5.5.23.html`

Client-side disclosure of credentials. We further found multiple hard coded API keys and other sensitive information, *e.g.*, email addresses. More specifically, we were able to extract API keys to access *Google Maps*, *Mapquest*, *OpenWeatherMap*, the *San Francisco transit* API, and *Telegram* through a bot that allows sending messages in its name. These services could allow impersonation, information leaks, or financial infringements for the application developers due to API overuse or lockdowns.

7

Threats to Validity

One major threat to validity is the completeness of the study, *i.e.*, it is not guaranteed that we found all major libraries used for web communication in Android applications. Furthermore, we only focused on the Android platform. This may not be representative of the mobile application market in its entirety since the use of web communication in other mobile operating systems, *e.g.*, iOS, has not been adequately considered.

Another major threat is the bias introduced with the selection of apps. We conducted the automated analysis using our *Jandrolyzer* tool for all open-source apps and for random closed-source apps in our dataset, however, not every app category has been equally represented in the analysis. The open-source apps have been collected from *F-Droid* and the free closed-source apps from the *Google Play* store. Some applications from *F-Droid* are also distributed through the *Google Play* store which could result in applications appearing in both datasets. Other third party stores have been excluded from our study, because some of them alter the apps' source code which would further bias our results [24].

We developed *Jandrolyzer* to the best of our knowledge, but nevertheless, it suffers from the inherent limitations that come with static source code analysis, *e.g.*, the large demand for resources. Hence, we built the tool with a focus on performance rather than on accuracy, because we intended to retrieve results from as many apps as possible even though we risked broken or incomplete results. That is, we increased the speed of the analysis process by oversimplifying some programming language constructs, *e.g.*, conditions, and by introducing constraints, *e.g.*, a maximum resolving depth for variable values, and a maximum

execution time for the decompilation process and one for the analysis process.

In the manual validation of the analysis results we only considered the output of our tool, however, we neither investigated apps manually to confirm the results of *Jandrolyzer*, nor did we consider apps that did not succeed the analysis. In addition, the authors were performing all the analyses and evaluations themselves which is a threat to construct validity through potential bias in experimenter expectancy. Finally, to avoid any damage on production servers we did not test the altering of resources on remote systems.

8

Future Work

In this thesis we present the prevalent Android web communication frameworks and the corresponding analysis tool *Jandrolyzer*, which can extract and validate API endpoint data from open- and closed-source apps. However, there exist many opportunities to improve the tool or to more thoroughly reason about the results:

Tool: Dataset. We tested our analysis tool on a corpus of 413 open-source and 834 closed-source applications. However, the current small-scale analysis could be transformed into a large-scale analysis to increase the expressiveness of the results by virtue of the increased app diversity.

Tool: Analysis time. The decompilation and analysis processes consume much resources. Therefore, we could increase the maximum decompilation and analysis time for each app. With this measure we would gain more successful analyses especially for rather complex apps.

Tool: Library support. We implemented tool support for the most prevalent web communication libraries. Nevertheless, support for more network and data conversion libraries would increase the quality of our results. In addition, we could improve the analysis quality for the existing libraries to better model and reconstruct their use of web communication, *e.g.*, by accurately considering conditional statements.

Tool: Server configuration assessments. *Jandrolyzer* supports the automated validation of API endpoints based on the collected and reconstructed data. Unfortunately, the software metrics available in response headers are not yet automatically assessed for any security-critical information. For example, the tool

could immediately determine the specific software releases used on API servers and automatically search for vulnerabilities in relevant CVE databases. Moreover, the testing of the applications' typical network ports on web API servers using default credentials might also lead to interesting insights, nevertheless, regulations must be obeyed.

Tool: Automated exploitation. The tool provides the necessary information to access API endpoints. Whenever sensitive keywords appear in queries, *e.g.*, `startFrom`, `endWith`, `js`, DoS vulnerabilities could be disclosed in an automated manner, or parameters could be used to test for potential code injection weaknesses.

Tool: Dynamic analyses. Static code analysis techniques can be combined with dynamic code analysis methods to validate the correctness and the completeness of the reconstructed data for a specific application. In particular, we could try to install and run applications through the command line interface of the *Android Debug Bridge* (ADB).[1] ADB allows one not only to install and start Android apps, it also enables one to list all available activities, *i.e.*, the different views, of an application, to grant permissions, and to launch an app's specific activities. Furthermore, a reverse proxy can be used to intercept the app's network communication. While we performed some initial tests, we did not investigate this area any further due to three major open issues. The first open issue remains the certificate pinning of apps: Certain apps include their own certificates stored inside of the app and reject any other certificates, and even otherwise trusted certificates from the trusted store of the device. This security measure prevents more thorough analyses as changes to the app's code would be required to eavesdrop the app's network communication. The second open issue is the unavailability of a subset of the app's functionality due to missing user input. For instance, many views require additional user input to be fully functional, *e.g.*, a user name and a password, a network address, a paid subscription, *etc.* The automated insertion of such values is a non-trivial problem. The third open issue remains the background communication of apps. Some apps cause spontaneous network traffic throughout the day. While we can see and analyse this background communication it is hard to trigger by any action as it is often bound to timers or specific user actions performed on the device itself.

Reasoning: Manual assessments. Our tool simplifies the manual investigation of privacy related issues such as the synchronization of personal information with web services without the user's knowledge or consent. Consequently, more in-depth followup studies on the results are advised.

Reasoning: IDE support. The insights gained in this thesis could be used to create a plugin for Android Studio to raise awareness of potential security issues related to web APIs among mobile application developers by providing actionable feedback through code linting and quick fixes.

[1] `https://developer.android.com/studio/command-line/adb`

9 Conclusion

We conducted a manual static analysis of 160 open-source projects to compile a list of commonly used network and data conversion libraries and to analyze their patterns of use. Subsequently, we developed a first static analysis tool to assess the popularity of the collected network libraries and to establish an automated process to find potential network code snippets based on regular expression matching techniques to further increase our knowledge of the developers' web communication use. Unfortunately, this tool yielded countless false positives because of the missing context and type information. Nevertheless, a further manual reverse engineering of these code snippets still provided additional insights into the implementation of web communication facilities in Android applications, *e.g.*, the use of XML and JSON, hard-coded API endpoints, API keys, access tokens, and request body content such as JSON schemes.

Based on these findings we created the analysis tool *Jandrolyzer*. *Jandrolyzer* automates the decompilation, the network code snippet detection, the reverse engineering, the extraction of API endpoints and JSON schemes, and the validation thereof. The decompilation is performed by *JADX* and the reverse engineering as well as the extraction is built on *JavaParser* which parses source code, creates ASTs, and provides symbol resolution techniques to ensure accurate findings. *Jandrolyzer* provides API endpoint extraction strategies tailored to the most prevalent web communication libraries such as *OkHttp* and *Retrofit*, and custom fit JSON scheme extraction strategies for the most common data manipulation libraries such as *Gson*, *Moshi*, and *org.json*. Lastly, *Jandrolyzer* is able to test API endpoints with various combinations of HTTP methods and request bodies. The server responses are stored on disk for later analysis. Incomplete

JSON or URL key-value query pairs are reconstructed with help of the source code by filling in missing values with values found at variables using a similar name.

With the help of this tool we conducted an analysis on 413 open-source projects from *F-Droid* and 834 closed-source applications from the *Google Play* store. Out of these 1 247 apps our tool was able to successfully process 450 apps (36.0%). We found that for both open-source and closed-source applications the prevalent web communication channels are based on *java.net* classes, followed by third party libraries such as *OkHttp* and *Retrofit*. By far the most used JSON value type in JSON schemes is `STRING` for open- and closed-source apps. We further found in closed-source apps substantially more advertisement services, network code that is spread across more files, and more complex endpoint paths consisting of more path segments. Surprisingly, the secure HTTPS protocol is used in the majority of extracted endpoints from open-source applications, but the opposite is true for closed-source apps: The majority of endpoints extracted from closed-source applications use the insecure HTTP protocol instead.

During the manual validation of the results from *Jandrolyzer* we finally encountered numerous security issues: The unnecessary disclosure of server configurations, outdated web servers and language interpreters with known security vulnerabilities, insecure web content display in web views using the HTTP protocol, leaks of internal error messages, hard coded API keys and other sensitive data in source code, shell commands inside a request body, and ultimately, we found public private APIs lacking any kind of authentication or authorization mechanisms.

This work represents a baseline for further research in this direction.

Bibliography

[1] A. Arora, S. Garg, and S. K. Peddoju. Malware detection using network traffic analysis in Android based mobile devices. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 66–71. IEEE, 2014.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[3] K. Benton, L. J. Camp, and V. Garg. Studying the effectiveness of Android application permissions requests. In *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 291–296. IEEE, 2013.

[4] R. Chang, L. Jiang, W. Chen, H. He, S. Yang, H. Jiang, W. Liu, and Y. Liu. Towards a multilayered permission-based access control for extending Android security. *Concurrency and Computation: Practice and Experience*, 30(5):e4180, 2018.

[5] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Analyzing Android encrypted network traffic to identify user actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, 2016.

[6] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for Android. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 481–494, 2014.

[7] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61. ACM, 2012.

[8] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz. Security code smells in Android ICC. *Empirical Software Engineering Special Issue*, 2018.

[9] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in Android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130, Sept 2017.

[10] O. S. Gómez, A. A. Aguileta, R. A. Aguilar, J. P. Ucán, R. H. Rosero, and K. Cortes-Verdin. An empirical study on the impact of an IDE tool support in the pair and solo programming. *IEEE Access*, 5:9175–9187, 2017.

[11] K. Kulkarni and A. Y. Javaid. Open source Android vulnerability detection tools: A survey. *arXiv preprint arXiv:1807.11840*, 2018.

[12] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011.

[13] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2015.

[14] M. Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. "O'Reilly Media, Inc.", 2011.

[15] A. Mendoza and G. Gu. Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769. IEEE, 2018.

[16] J. Shen, Z. Chen, S. Wang, Y. Zhu, and M. U. Hassan. DroidDetector: a traffic-based platform to detect Android malware using machine learning. In *Third International Workshop on Pattern Recognition*, volume 10828, page 108280N. International Society for Optics and Photonics, 2018.

[17] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2347–2356. ACM, 2014.

[18] 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 *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS '14*. Citeseer, 2014.

[19] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. 05 2019.

[20] R. Verdecchia, I. Malavolta, and P. Lago. Guidelines for architecting Android apps: A mixed-method empirical study. 2019.

[21] N. Wang, B. Zhang, B. Liu, and H. Jin. Investigating effects of control and ads awareness on Android users' privacy behaviors and perceptions. In *Proceedings of the 17th international conference on human-computer interaction with mobile devices and services*, pages 373–382. ACM, 2015.

[22] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[23] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[24] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[25] Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting developer credentials in Android apps. In *WISEC*, 2015.

[26] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen. Deepflow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data. In *2017 IEEE symposium on computers and communications (ISCC)*, pages 438–443. IEEE, 2017.