Security Threats in Android-based Opportunistic Networks

Bachelor Thesis

by

Martin Nowak

born in
Leverkusen

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

June 2017

Supervisor:
Andre Ippisch, M. Sc.

Abstract

The development in the mobile technology sector has made huge progression in the last decades. It produced many new technologies and made improvements to existing telecommunication mechanisms that enabled a new form of mobile devices, the so called smart devices. Smart devices are small handheld devices with powerful hardware and plenty of storage, capable of executing complex computation tasks and of communicating with each other. Eventually, smart devices were employed on the paradigm of Opportunistic Networking. Since then, many approaches were developed that use modern smartphones and that exploit the mobility of their users to establish the network. While much research was done for network establishment and routing, the research for security just started to attain more interest in recent years. Security is a cornerstone for gaining the trust of users. Being able to offer them a secure, reliable communication environment is a key point for success and high adaptation rate.

In this thesis security threats against Android-based Opportunistic Networks were researched and proposals for possible solutions are given. A special focus of the thesis was on the opptain networking application as a representative example. The course of action was to first explain and characterize threats found in corresponding literature as well as to find threats by analyzing the opptain source code and behavior. This was followed by a discussion of possible solutions and mitigation techniques.

It is evaluated, that the attacks on Android devices mainly target data confidentiality, integrity and authenticity while the attacks against the network functionality primarily aim to disrupt the network and in consequence decrease its performance and availability.

Promising looking solutions and mitigation to threats can be achieved by a better access control to the network application, thresholds against spam, encryption by default and a form of trust and reputation system to find trustworthy communication partners. In conclusion, further research on real devices to test and evaluate the former must be carried out, since the aspects were mostly discussed theoretically.

Acknowledgments

I would like to thank my friends and all the other people that supported me during my work on this thesis.

I wish to express my gratitude to my parents who always supported me, no matter what.

A special thank you goes to my supervisor Andre Ippisch, who suggested me the thesis topic. He always helped me when I had questions and gave valuable tips.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The development in the mobile technology sector has made huge progression in the last decades. It produced many new technologies and made improvements to existing telecommunication mechanisms that enabled a new form of mobile devices, the so called smart devices. Smart devices are small handheld devices with powerful hardware and plenty of storage, capable of executing complex computation tasks and of communicating with each other. Researchers around the world saw the potential of the devices and started to work on systems that would leverage the new technology. Eventually, smart devices were employed on the paradigm of Opportunistic Networking. Since then, many approaches were developed that use modern smartphones and that exploit the mobility of their users to establish the network. While much research was done for network establishment and routing, the research for security just started to attain more interest in recent years. Security is a cornerstone for gaining the trust of users. Being able to offer them a secure, reliable communication environment is a key point for success and high adaptation rate. This thesis will focus on security threats for Opportunistic Networks on Android smartphones, that should be considered in the implementation, and proposes mitigation mechanisms.

1.2 Related Work

The Android network application called opptain, that is analysed in this thesis, was developed by A. Ippisch in his thesis [Ipp15] to offer a multilayer framework for opportunistic networking on Android devices. He discusses the challenges present on the Android operating system which need to be addressed to create an Opportunistic Network and formulates the design decisions that are then implemented. The security related characteristics of the topic were only touched rudimentally as the focus was on providing the basic functionality for an Opportunistic Network. This thesis will extend that aspect on a deeper level.

J. Leßenich developed a routing protocol for Opportunistic Networks, called Neighbourhood Aggregating Social and Geographic Label Protocol (NASGL) [Les16]. It follows a context aware and quota-based approach and it was also implemented to be used in opptain.

1.3 Outline

In this chapter the motivation for researching security threats in Android-based Opportunistic Networks was elucidated and the opptain network application, which will be used as a representative example in this thesis, was introduced.

In Chapter 2 the basics of Opportunistic Networks, of the Android operating system and of the opptain networking application are discussed. Section 2.1 additionally mentions the possibilities and challenges of Opportunistic Networks briefly. In Section 2.2 and in Section 2.3 some implementation details about Android, including its security features, and opptain are given.

In Chapter 3 the threats and attacks are researched and discussed. Section 3.1 characterizes an attacker and lists possible attack targets of a system. Then, attacks that operate on the Android device and that target the network application functionality are explained in Section 3.2. Following, attacks that target the Opportunistic Network functionality are explained in Section 3.3.

In Chapter 4 solutions and mitigation techniques for the attacks from the previous chapter are discussed. Most notably an access control mechanism for the opptain network application is

proposed in Section 4.1.1 that could also be implemented by other applications and Section 4.3 explains the concept of social trust, which seems to be a promising solution for choosing trustworthy communication partners.

Chapter 5 concludes the thesis and briefly summarizes the research results. The main findings and challenges will be presented. Finally, problems and solutions, that might be possible topics for future work, are listed.

Chapter 2

Fundamentals

This chapter covers and explains the fundamentals required for a better understanding of this thesis. At first the term Opportunistic Networking and the concepts behind it are explained in Section 2.1. Following that, we take a brief look at the Android operating system and its baseline security features in Section 2.2 and last but not least we present the Android network application opptain and its features in Section 2.3.

2.1 Opportunistic Networks

Opportunistic Networks (OppNets) are a special kind of delay tolerant networks (DTN), which are a subclass of ad-hoc networks. The key characteristic is the communication of nodes without an existing direct route between them. Data is exchanged by opportunity and dissemination functions under the store-carry-and-forward principle. The communication between hops is purely P2P. In detail, one-hop communication is synchronous P2P with direct content exchange while multi-hop communication is asynchronous P2P with content being temporarily stored in the network nodes while being forwarded. Forwarding in OppNets is enabled by exploiting the mobility and social interaction of the users, who form the network with their smartphones or other smart devices while using it.

2.1.1 Basics

The idea is that OppNets are established by mobile and (temporal) stationary nodes, that are distributed in an environment without a centralized access or entry point to the network. Every node plays an active role as both network transmitter and receiver while also being a forwarder, relaying communications from other network participants. Because of their mobility, nodes will meet occasionally and get into radio range. When they do, their devices establish a local mobile ad-hoc network (MANET) to communicate. The intermittently-connected nodes use this opportunity to exchange and forward their data. Ideally, the next hop will be closer to the destination than the current holder of the data, to increase the chance of a successful transmission to the destination.

Since nodes are autonomous and free to move (attribute of MANETs), sporadic and short connections coupled with long disconnects and partitions of the network are frequent as well as expected, so that no real end-to-end path for directly forwarding data exists (attribute of DTN) when the destination is not in one-hop range. Therefore delivery of messages is delayed and not instantaneous most of the time. All this leads to a special network topology. To be more specific, it is a dynamic topology that can change quickly and randomly at any time. Consequently classical deterministic routing with a routing table, using connections over stable-links does not work because neither the tables can be build nor do the connections exist. But despite that, nodes can communicate with each other. Routing in OppNets is either done by just flooding packets into the network or with more attention to previous encounters and experiences, context sensitive information like location and routing data collected from other peers. It is a well working probabilistic approach, albeit it generally grants no guarantee that the packets will reach the destination if the recipient is not a one-hop peer. Notably responses might never reach the original sender, so he might never become aware that his transfer completed successfully. Both is also true for flooding.

2.1.2 Possibilities and Challenges

The decentralized infrastructure of OppNets eliminates the risk of a single point of failure. Therefore OppNets can be used as a communication alternative to the infrastructure based internet. Especially in disaster scenarios, regions without communication infrastructure or even in situations where this infrastructure is shutdown on purpose.

On one hand the infrastructure-less approach brings the challenge of a more complex routing and administration, that has to be dealt with and which is often the cause for a lower performance of the network in comparison to normal hierarchic ordered systems. But on the other hand the rapid rise of mobile device users and the sinking prices for continuously better getting hardware, make it easier to realize OppNets because the advanced services needed, that is, connection requests, routing/forwarding and security, can be implemented in a more efficient and feasible way. But limited energy resources in mobile devices that rely on batteries are still a big problem as node operations like transmission, reception and other calculations still deplete a lot of power. This calls for an efficient management mechanisms to conserve energy and increase lifetime to be developed. But this is not a trivial task, since the network is likely to be very heterogeneous, that means available computing power and resources will differ a lot from device to device.
The main challenges of Opportunistic Networks with mobile devices are: Reduced energy consumption, support of many devices, easy deployment methods, development of mechanisms to provide security and privacy for users. The latter two will be discussed in this thesis.

2.2 Android System and Security

Android is an open source operating system developed by Google. Initially Android was designed for mobile devices such as smartphones and tablets but since its first release in 2008 it was branched out by Google for *TV*, *Cars*, *Things* and *Wearables* and various other systems by other companies. It is built on top of a modified, long-term support version of the Linux kernel. The kernel was chosen for its portability and rich set of features [M. 17]. Device manufacturers can easily develop their hardware drivers for a well-known kernel and developers can stop worrying about low-level system functionality that Linux will handle for them. The functionality in question is:

1. **Process Management:** Starts and stops processes, allocates and deallocates memory and resources, handles threading
2. **Driver Model:** Ensures an environment in which drivers can be easily integrated, helps to have a reliable running system

3. **File System Management:** Manages the file system which controls the data storage service
4. **Network Stack:** Responsible for all network activities and communication
5. **User accounts:** Manages users, can assign permissions and separate users from security sensitive system components

Additional to that, Android added some modifications as already mentioned above. One set of added features are specific drivers for mobile devices, needed to enable cellular communication protocols like LTE. Furthermore, drivers for inter-process communication via binders, logger capabilities, ashmem for shared memory functions, pmem for persistent memory functions and wake locks to keep the system awake were added. A special extension is the power management module which plays a crucial role in extending the lifetime of the battery powered mobile devices.

2.2.1 Platform Architecture

Taking a look at the Android Framework [Goo17g] makes clear, the Linux kernel is just the foundation of a sophisticated software stack. Right above it lays the Hardware Abstraction Layer (HAL). It is a collection of library modules that provide access to the device hardware via standard interfaces, which higher-level frameworks can use. Next up is the Android Runtime (ART), the process virtual machine in which Android applications are executed. ART has a focus on low memory consumption. To accomplish this it executes DEX files, a bytecode format with a minimal memory usage specially designed for Android. Since Android 5.0 (API level 21) ART is the successor of the Dalvik VM and apart from optimization introduced Ahead-of-time (AOT) compilation to Android. For the purpose of performance many core Android components and services are built from native code, for example ART and HAL. Therefore Android is equipped with native libraries written in C and C++. Through Java framework APIs the functionality of some is exposed and Android NDK allows a direct access to those libraries. The previously mentioned Java API Framework assembles the entire feature-set of the Android OS available to developers. It includes APIs to simplify the creation of apps by offering a convenient access to core, modular system components and services (View System, Resource Manager, Notification Manager, Activity Manager and Content Providers). On top of the stack are the System Apps, that Android delivers to use the basic features of the device, like a dial app, web browser, and so on.

Application Components

The basic building blocks of an Android application are Activities, Services, Broadcast Receivers and Content Providers [Goo17a]. Each has a distinct lifecycle in terms of when it is created and destroyed and each is a possible entry point into an application for either the system or a user. A brief explanation of these components follows:

1. **Activity:** A single screen with a user interface. The user can start an activity to interact with the app. Typically an app has multiple activities with one acting as the entry point, also called launcher activity.
2. **Service:** A background component with no user interface to keep an app executing. Services run in the background to perform long-running operations or to perform work for remote processes. Other components can start a Service to run until the work is finished or a Service can be bound to run until it is released.
3. **Broadcast Receiver:** Broadcast Receivers are a possibility to receive messages from the system or other apps. The special point about this is, that the app does not need to be running to get the broadcast. They are short running background tasks triggered by Broadcast Intents and can be displayed in the status bar.
4. **Content Provider:** Is a persistent data storage associated with the app. It can be used to share data with other apps. The data is identified with URIs that can be defined with permissions to allow access for certain apps.

In order for the Android OS to know that the components exist in an app, the app must contain all its components in the AndroidManifest.xml file. The manifest file is read by the OS to gather information about an app. It declares used and defined user-permissions, components with optional IntentFilter as well as required permissions to use them and much more.

Communication Mechanisms

Communication between different components or even different apps can be done in several ways. The most used one are Intents. An Intent is an asynchronous messaging object someone can use to request an action from another component. When created, several fields and data may be set, then the Intent is send through an Android API call which implicitly specifies the type of the destination component. If the Intent states a specific component it is

explicit and directly delivered. Otherwise it is implicit and resolved to the components that have a matching IntentFilter. Activities, Services and Broadcast Receivers are able to receive Intents. IntentFilters can be declared by application components to receive implicit Intents, indicating which operations and data they can work with. If the OS determines a component to start, attributes of the Intent are compared with the IntentFilter. When they match the Intent will be delivered. In the case of multiple matching Activities and Services an implicit Intent is only delivered to one component, while in the case of Broadcast Receivers all get the Intent. Thereby must the user choose an Activity if there is not a default one for the Intent while for Services the system randomly chooses one of the available without notifying the user or since Android 5.0 an exception is thrown.

2.2.2 Security

Security in Android is handled on two layers. The first layer is the system security, which is enforced by the Linux kernel. The second layer is on the app level, realized by Android using a permission system.

Kernel Security Key security features of the Linux kernel that are used in Android are the user-based permission model, process isolation, extensible mechanisms for secure interprocess communication (IPC) and the ability to remove insecure parts of the kernel [Goo17h]. Android assigns each application to a unique Linux user to identify and isolate its resources. In practice this means that a user ID (UID), chosen by the kernel and unknown to the app itself, is set to all app files as the owner. In comparison, other operating systems usually let many applications run under the same user and permissions. When an app is started, it launches in its own Linux process with a dedicated new instance of the ART (or Dalvik). The virtual machine effectively isolates applications from another. The aforementioned points basically create an Application Sandbox in the kernel that ensures that code and memory manipulations between different applications and processes cannot be carried out.

App Security Android [Goo17b] uses the principle of least privilege, so apps can only work with resources or access other protected system parts when they request the permission for this action in the manifest. The permissions on this level are enforced by the Java

Framework API. Apps can also define custom permissions to be accessed. For Apps to share resources and communicate, one possibility is to run with the same UID, which requires that the apps are signed by the same signing key/developer. Other ways to share and communicate despite the isolation of apps are the previously described Intents, Content Providers, Services and Binder. All listed items are inter-process communication mechanisms that are delegated through the system and not directly, if they leave the process. Binder is a capability-based remote procedure call mechanism for in- and cross-process calls, that is also used in the implementation of Intents and Service Binders.

2.3 opptain

Opptain [Ipp15] is an Android network application that leverages the widespread distribution and power of modern smartphones to build an Opportunistic Network. It was developed with the goal to provide an easy to use platform that can be deployed on nowadays available resources. Users should be able to communicate and transfer data secure and fast without being bothered to interact with the app. An internet connection is not necessary to establish a working network and third-party apps can benefit from opptain as network layer by using the provided application interface to relay their data. As an application that provides networking functionality for other apps opptain can be classified as a middleware for the Android operating system (see Table 2.1). It operates between the transport and application layer. Key features that are offered include connection of Android devices over Wi-Fi to build an OppNet, messaging over packets, user identification, encryption and aggregation of meta information for routing in the network. As a remarkable side note, opptain was developed to run on unrooted off-the-shelf smartphones.

Opptain Protocol Stack			
Protocol Layer	Device A	Device B	Device C
Application	API app		API app
Middleware	network app	network app	network app
Transport	TCP	TCP	TCP
Network	IP	IP	IP
Data Link	802.11 MAC	802.11 MAC	802.11 MAC
Physical	802.11 PHY	802.11 PHY	802.11 PHY

Table 2.1: Opptain Protocol Stack

2.3.1 Connectivity

The connectivity in opptain is realized with the Wi-Fi tethering hotspot feature on Android. This approach is a workaround to control the connection automatically, as the existing ad-hoc modules, Wi-Fi Direct and Bluetooth, do not offer an API to programmatically accept connections. This is important for a convenient user experience. Just like the user expects the normal internet to work without any action from his side, opptain has to follow this criteria to become widely adopted and used. Devices can connect if at least one opens a tethering hotspot which others can join as clients. The resulting pseudo ad-hoc network is then used to exchange data and network information over TCP, that comes from own applications or that are being forwarded. The decision for either being a hotspot or client is determined by a prior scanning phase in which opptain searches for other devices in vicinity. Depending on the results and meta data, like past encounters, routing info and signal strength a hotspot is chosen. If no hotspot is available or suitable, the device can rescan or become a hotspot. All aforementioned tasks, from network initialization and management to inter-device and inter-application connections for file-handling are handled by a background service. The service achieves the mentioned user independent functionality.

2.3.2 Identification

Device identification and connection security are implemented in a pretty clever way. When opptain is installed for the first time a public-private-key pair will be generated for identification, authentication and encryption. The public key is used as DeviceId and encryption key at the same time, so that no additional key exchange for end-to-end encryption, which can be hard to realize in OppNets, is needed, once the DeviceId of a communication partner is known. Besides that the public key fingerprint is part of the SSID to identify a hotspot before a connection. Furthermore opptain offers to verify the entity behind a key by QR code scanning it from an application activity. On the connection event of two devices a challenge response authentication takes place to exchange keys for point-to-point encryption. The encryption algorithms in usage are: *"timing-attack resistant XSalsa20 stream cipher with a 192 bit nonce for encryption in general, the Elliptic Curve Diffie-Hellman (ECDH) on Curve25519 for key exchange and Poly1305 for authentication"* [IG17]. Opptain also defines three custom permissions. On the on hand there are read/write permissions for a Content Provider that stores contact information and on the other hand there is a communi-

cation permission that must be held by third-party applications in general, if they want to use the application interface.

2.3.3 Routing

Routing in opptain is addressed in two ways. The first method does not really use any information and just spreads messages into the network. The routing protocol is named Spray and Wait [SPR05]. The Sender sends a copy of the message with a defined Time-To-Live to n meet nodes. These nodes only forward to the destination and do not relay to other nodes. An also implemented extension of this approach is Binary Spray and Wait [SPR05]. Here the origin sends the message to two nodes with a value of n divided by 2. These data ferry nodes are able to relay them to other ones they meet, if they get a message with n greater than 1. The second method tries to make optimized forwarding decisions based on the network situation and the nodes it encounters. It is named Neighbourhood Aggregating Social and Geographical Label Protocol (NASGL) [Les16]. NASGL is a quota-based context-aware hybrid that uses principles of the History of Encounters and Transitivity of PRoPHET [LDS04], the Meetings of HiBOp [BCJP07] as well as social and geographical context data. If two nodes meet they exchange all known routing and meta information about connections, the network, social and geographical data. Based on these information a node can calculate the probability that an encountered node will successfully deliver the message and then can decide to forward it or not. In NASGL two or more nodes can enter an aggregation mode when they are static to each other. The nodes will then exchange all their decent neighbors with each neighbor and new arriving nodes. This information allows to calculate a cluster in which nodes are close together. Communication in the cluster should be relatively fast and reliable.

The data structure used by opptain to forward and route packets is called OpptainBundle. It is a bundle object that consists of various fields which are holding information for routing and the payload it carries. The elements are as follows: *BundleId, Origin, Destination, CreationTime, TTL, NetworkPath, ApplicationPackage, PayloadData, PayloadFile, PayloadInfo, PayloadMeta and AdditionalInformation.*

Chapter 3

Attacks and Threats

In this chapter motivations of an attacker and points of interest of a (secure) system are introduced in Section 3.1. Following that, attacks aimed at the Android operating system and software as well as against Opportunistic Networks, that might be used to target opptain and its functions, are researched and analyzed in Section 3.2 and Section 3.3.

3.1 Characteristics of Attacks and Security

An attack can be described as follows [SE07].

Passive attack The attacker silently listens in the background. He does not intent to interrupt any functionality or to disturb the routing between nodes, but rather eavesdrops on application messages and data as well as routing traffic to gather potentially exploitable information.

Active attack An attacker actively uses vulnerabilities, exploits and other weaknesses available on the target to launch an attack. It is not uncommon that the knowledge on this issues is often acquired beforehand by a passive attack. He will try to gain access to core components for the purpose of stealing sensitive data and to disrupt the normal functionality, leading to unexpected, maybe even undesired behavior. For instance, malicious nodes in an OppNet will disturb the routing protocol by modifying routing data and flags, fabricating false routing information, impersonating other nodes or redirecting traffic.

3.1.1 Attacker

Let's have a look at the attacker now. First of all, the source of an attack in opptain, more specifically against its established network, has to be identified and defined. It can be started from an outsider or an insider. Outsider attackers, as the name implies, are not part of the network and do not have any sort of trusted keys. They typically rely on message relay, replay, or delay to influence the network. Insider threats occur when a fully trusted node, with appropriate keying material, is compromised. Network parameters and access can be manipulated directly.
The incentives an attacker could have are divers, just like the ways to carry out an (successful) attack. A popular target is the authorization or access control of a system. If an attacker manages to circumvent it, he has unrestricted access to network, functions and resources. Consequences can be data loss, theft and/or manipulation. Another critical aspect is the availability. In particular applications like opptain that rely heavily on the cooperation of their users to function in a reliable way, may be focused by attackers or competitors that threaten to actively disrupt the network by implanting malicious misbehaving nodes.

3.1.2 Security Aspects

To get a further understanding, some characteristics of a secure and reliable networking system are given [SXB07] [KM14]. On the on hand we have to consider the conditions for maintaining the service in a reliable way.
Authorization: Access control to systems or resources to only allow selected authorized entities access to the network, its functions and resources.
Availability: Functions and information must be available when needed. To guarantee this, computing systems storing and processing information, security controls protecting it and communication channels used to access it must be operating reliable and correct.

On the other hand is the main function of messaging. Especially for messages, that are transmitted over open and unsecured channels, certain characteristics are desired to consider them as save and sound. However, without clearly specified guidelines and enforcement of these, entities with a malicious intent can, and most likely will, try to injure the following. But just satisfying the guidelines will not be sufficient. Controls by all participating devices have to be carried out or else attacks may go unnoticed.
Confidentiality: The property to not make information available and to guard it from unau-

thorized individuals, entities or processes. It is usually achieved by cryptography in network security.

Integrity: Data integrity refers to the condition that data maintains accurate and complete over its entire life-cycle. This means that data remains unaltered, making it an indicator for unauthorized modifications. Integrity services are designed to protect against deliberate or accidental alteration ensuring proper functioning of the underlying system.

Authentication: Verifies the identity of an entity in the network. Without this attackers can impersonate another entity to access services assigned to that entity or to carry out actions in its name.

These are the obstacles that an adversary will try to bypass if they are implemented. So the classification of attack targets can be summed up as follows: authorization, availability, confidentiality, integrity and authentication. The possibilities for a system intrusion are mostly based on actions like collecting meta information, perturbing communications, data aggregation, exhaustion of resources and will be discussed in the next sections. Figure 3.1 gives a rough classification of the attacks against Android-based OppNets.

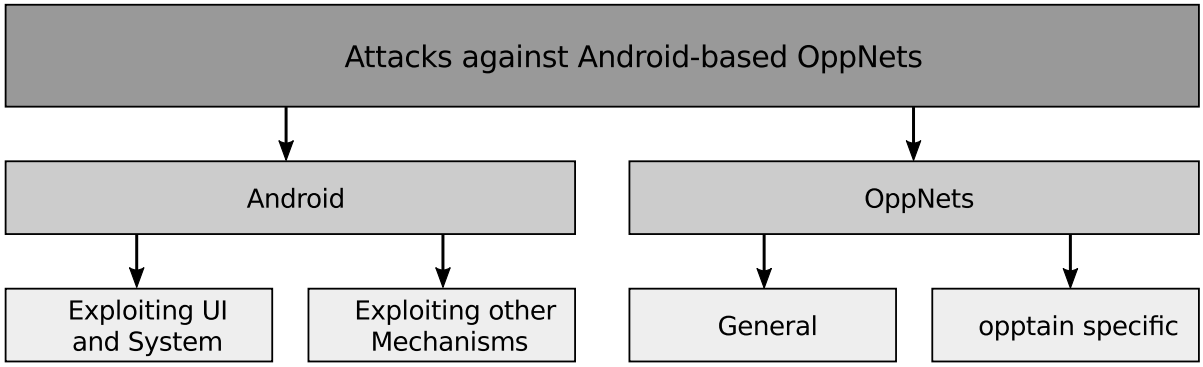


Figure 3.1: Rough classification of attack types

3.2 On the Android Device

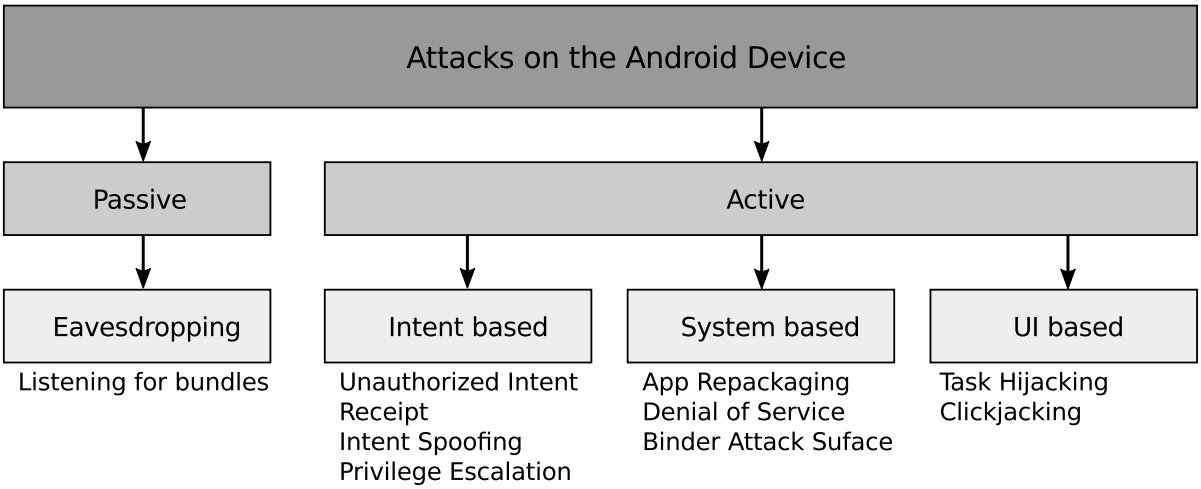
In this section we focus on attacks and vulnerabilities which target the function of the opptain network application on Android. Attacks requiring root level permissions are not considered, because adversaries with that privilege have non-restricted access to every system and non-system component. Every security feature implemented on the device would be circum-

vented instantly most of the time. The attacks, Figure 3.2, will be carried out by malicious applications. There are several possibilities that this applications will end up on the target device:

- They can be a repackaged version of a popular app.
- App with useful functionality, that carries out malicious actions in the background.
- Users can be tricked to install them with social engineering.

One could argue, that an user had the ability to stop the installation and thereby prevent the attacks. In particular if the requested permissions during installation seem suspicious. But most of the users are not aware of the security and privacy risks induced by permissions. As [FHE+12] showed, only 17% of Android users pay attention to permissions during installation. A shocking number of 42% are not aware of permissions at all.

Figure 3.2: Taxonomy of Android attacks



3.2.1 Intent Based Attack Surfaces

Intent based attacks make use of the functionality of *implicit Intents* or *Intent Receivers* and are regularly exposed by incautious developers themselves when they use implicit Intents for intra- or inter-application communication. Internal components and messages might unintentionally become accessable by third-party applications. Any untrusted app may be an attacker in this scenario, while a victim may be any app. There exist mainly two types of this

kind as described by [KCHW12].

Unauthorized Intent Receipt When app components send *implicit Intents* to other ones either in the same app without using a signature-level permission or to another app that should be the only recipient, a component of the right type with matching *IntentFilter* can intercept the Intent. The possible resulting attacks and consequences depend on the Intent type and the invoked component.
Broadcast Intents are vulnerable to passive eavesdropping, which can harm security or privacy if the Intent contains sensitive data. *Ordered Broadcast Intents*, which are delivered to Broadcast Receivers in a priority order, are vulnerable to both active denial of service attacks and malicious data injection, as well as eavesdropping. Each recipient can stop the Intent from propagating to the rest or change the data before passing it to the next recipient. *Activity and Service Intents* are vulnerable to hijacking attacks, in which an attacker intercepts a request to start an Activity or a request to start or bind to a Service and the malicious application starts its own Activity or Service in its place. This attack allows an attacker to steal data from the Intent, hijack the user interface in a way that may be nontransparent to the user, and return malicious data to the starting component.

Intent Spoofing *Intent Spoofing* can happen if *internal components are exposed* to other applications. That takes place when the component is private and declares IntentFilters for receiving implicit Intents, without explicitly setting the exported option in the manifest to false. The component may be vulnerable to attacks in which a malicious application spoofs the internal Intent. *Broadcast Receivers* are vulnerable to broadcast injection, in which the receiving component is tricked into believing a malicious broadcast Intent came from another component in its own application. The spoofed broadcast may then cause the victim component to change some state in a way that damages the user's security or privacy or even transmit malicious data contained in the broadcast elsewhere. Additionally, dynamically at runtime registered receivers are always public. *Activities and Services* are vulnerable to unauthorized launch or bind attacks, similar to cross-site request forgeries on the Web, and their exploitation can have similar consequences.

Privilege Escalation Another vulnerability that is often achieved via intents is *Privilege Escalation*. The definition of a Privilege Escalation on Android is: *"An application with*

less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)" [DDSW11]. So basically a transitive usage of privileges might lead to an unwanted privilege escalation, unnoticed by the Android permission system. The implied attacks can be categorized into two main types [RZZL14], the *confused deputy privilege escalation* and the *colluding application attack*.
In the case of a confused deputy privilege escalation the attacker application exploits vulnerabilities of a target benign application, the deputy, to perform unauthorized operations. It is a form of permission re-delegation, that can occur if:

- apps accidentally expose internal functionality
- on purpose exposed functionality is used in malicious, unexpected ways
- an attenuating authority, that should control the exposed functionality is implemented poorly

In a collusion attack at least two apps collude to gain an over privileged permission set. The malicious applications will either communicate directly over *overt channels* like IPC or internet sockets or use a more stealthy approach over indirect covert channels. *Covert channels* can be settings, Intent dates and types used by an application, space of certain file system elements or timing channels on the hardware level.

Opptain API

Opptain offers two methods for third-party applications to interact with the API. A broadcast receiver *OpptainBroadcastReceiver* and a service *OpptainRemoteService*. Both can handle the same incoming Intents that request the actions *REGISTRATION*, *BUNDLE* and *BUNDLE_REVOCATION*. The service also has a binder to attach to, which will be discussed in Section 3.2.7.

The *REGISTRATION* action registers the package in the Intent via its name for the use of opptain. *BUNDLE* passes an OutgoingBundle to opptain and if the sending app is registered, that bundle gets converted into an *OpptainBundle*, put into the database and announced to the *RoutingProtocolManager*. There are two major problems in both processes. First of all, the registration allows every app to register itself, without asking the user if he approves this. Of course, the requesting third-parties will have to hold the mandatory

de.opptain.waitress.permission.COMMUNICATION permission to communicate with both interfaces, but an user might want to control the access or did not even notice that the apps have the permission, as addressed at the start of this section. Second, the package name in the Intents is assumed to be the one from the sending third-party app. This value is set by the app itself though, since Intents offer no way to get the sender. An adversary app might lie about its package name, either to be allowed, if a hypothetical check at registration would exist, or to act malicious in the name of other benign apps to remain undetected.
BUNDLE_REVOCATION allows an app to revoke an OpptainBundle by its *BundleId*. This revoke deletes the bundle from the local database, if it is still contained in there. The problem with this call is, that it does not check if the app is registered and more severe, even if registered, allows to revoke not owned bundles. So if an attacker finds out a BundleId he can disrupt the corresponding app's communication by just revoking the bundle. At the current time no direct side channels leaking the Id, besides debug and trace logs that should be stripped from release version, were found in opptain. Future features or system changes could accidentally introduce such leaks though. Nonetheless, another easy way to get the Ids is through *Unauthorized Intent Receipt* of implicit Broadcast Intents from third-party apps. Although this is to blame on the developer and not opptain, the user could get angry for not blocking the revoke or because he thinks opptain does not work. In the worst case, the user will uninstall opptain for this reason.

All this could be interpreted as combination of a *confused deputy privilege escalation* and *Intent Spoofing*. Even though *OpptainBroadcastReceiver* and *OpptainRemoteService* are not private, the threats exposed from the three actions are definitely not by design and should not be publicly existent.

3.2.2 App Repackaging

App Repackaging is a process of disassembling an app's *Android Package Kit (APK) file*, that gives access to the source code of the program. Attackers can insert malicious code or manipulate critical components to turn a benign application into a *Trojan Horse* or *malware*. After repackaging the app it will be distributed over less or non monitored third-party app-stores. Popular apps are often targets of this kind of attack. The steps to repackage an application on Android are quite simple. After an attacker acquires the APK, he can disassemble the app with a disassembler such as the popular *apktool* [ibo17]. The generated malicious byte-

code is just embedded into the benign code and if necessary the *AndroidManifest.xml and resources* are modified. Now the maliciously changed program can be reassembled again using apktool.

This attack can be very effective against opptain in a certain scenario. The reason for that is the built in *APK-Sharer*. The idea behind this module is the distribution of opptain and any third-party app by sharing from person to person as alternative to official, centralized app stores. So in the case of non available internet connection smartphone users interested in opptain can install the shared APK. The problem in this situation is clear. The user has no way to check the validity of the app and has to trust the person to whom the APK belonged. Less likely, but still possible, are *social engineering* attempts in which particularly inexperienced users are encouraged to install the repackaged app from *unofficial or suspicious sources*. A successful installation leaves the attacker with a hijacked node. He can do anything from spying on the users communication to disturbing the network functionality with attacks also described in this thesis. An adversary can even establish a botnet if he manages to spread his repackaged opptain version onto numerous devices. The controlled nodes would fetch instructions over the OppNet or when a wireless connection is available, since opptain has also normal internet access permissions.

As a side note: if debug logs are not removed from the code of a compiled version, an attacker can re-enable them in the repackaged manifest. While testing with a release build of opptain, some logs that leaked bundle information, were still shown.

3.2.3 Denial of Service (DoS)

Denial of Service attacks fundamentally aim to disturb the availability of a service. On Android devices, DoS can be caused by Intents and make the device unresponsive to the user, target and terminate other apps or even force a soft reboot. Intents as fundamental communication mechanism can be abused by third-party applications because the Android OS does not put any limit on the amount or rate of Intents directly sent from an app.

Soft Reboot

The *Soft Reboot* [JES16] happens when the *Android framework* crashes, but the Linux kernel continues execution by restarting the framework. During the restart the reboot animation is displayed, leading to the description of a soft reboot. The crash is achieved by a service that rapidly sends Intents to the *ActivityServiceManager (AMS)* to start many new activity instances. Each new instance will run in a separate task, through the *MULTIPLE_TASK* and *NEW_TASK* activity flags, and cause the *system_server process* of the AMS to create a socket pair for user touch events. To amplify the procedure, instances are delegated to also launch new activities. When the process reaches its soft limit of 1024 file descriptors, it will eventually crash triggering the soft reboot, because either the attempt to open a new socket leads to an uncaught exception or the system_server is killed by the watchdog deamon process because it is in a deadlock. A malware that starts the DoS after every boot, by declaring an IntentFilter for the *BOOT_COMPLETED* Intent action, can only be uninstalled in save mode or with the ADB command line tool.

An attacker, that wants to make opptain unavailable to the user, can listen for the *PACKAGE_INSTALL* Intent action [Goo17f] and then launch the soft reboot. To make it more convincing, the activity that is launched in quick succession must look like the opptain *InitialWork* or launching activity. Additionally, disabling the activity transition animation will make it look like only on activity is started. Although the following crash, triggering the soft reboot, was provoked by the attacker, the user will most likely think it is caused by the newly installed network application and uninstall it. The attacker might be a competitor app, that is motivated to make opptain look unreliable and unstable, to not be replaced.

BundleSpam

A self tested DoS attack, achieved while working on this thesis, was carried out by *spamming opptain* with *OutgoingBundles*. The idea was to flute the *OpptainBundleMemory* with random bundles so that either the network app would crash and be unavailable for use or older legitimate bundles would get deleted.

To fill the memory can take some time because the *PayloadFile* itself will not be copied to the database, it is just referenced in the stored *OpptainBundle* to safe space. Instead the *Pay-*

loadData field must be used and this is limited because of Intent size restrictions [Goo17j]. The restrictions are induced by the Binder transaction buffer, that is used to buffer data before it is transferred between different apps via IPC. This buffer is limited to a fixed size of 1 megabyte concurrently and is shared by every binder transaction in progress for the process. If an Intent would not fit into the buffer, maybe because it is to large or because multiple fitting Intents arrive simultaneously, so that not all can be buffered, it is discarded. With that in mind, a PayloadData size of half a megabyte was chosen. The overhead of the Intent is neglectable as can be see in Table 3.1, so there should not be buffer collisions, unless other heavy IPC mechanisms in opptain are running, and the memory should be filled relatively fast.

The app written to execute the DoS is called TestApp. Its launcher activity contains a *spam button*, which will start a thread that sends 10 bundle Intents to opptain via the *OpptainRemoteService* every second. A *stop button* will stop the thread. The relevant bundle attributes for the attack are *PayloadData* which are just random bytes, and *Time-To-Live (TTL)* which was set to 10 minutes. The rest can be set arbitrary. After about 10 minutes the test devices started to get sluggish and shortly after opptain would crash, because of an *OutOfMemoryException*. When opptain is then started again, it would always crash with an OutOfMemoryException when the app begins to fetch data from the OpptainBundleMemory. This is the case when either the *Database or BundleOverview activities* are opened; or in *automation mode*, when peers start to exchange packets. This behavior, in regards to operation with the database, even happened 10 minutes after the attack, where every packet should have expired due to their TTL. Taking a deeper look into this issue revealed that the bundles are not deleted and neither waiting an additional 24 hours for an automatic clear would help, Table 3.2. The only possible solution is a clean *reinstall of opptain*.
Table 3.2 also shows that not all of the bundles were received (*10 Bundles a 0,5 MB * 60 Sec * 10 Minutes = 3000 MB*). Sending 50 bundle Intents with the sizes of Table 3.1 with a rate of 10 per second and a rate of 20 per second worked fine. So size and rate of sending are not too large nor the problem. A problem is though, that even with this amount of bundles the automation mode can crash because of the memory. These issues need further investigation, especially since 50 bundles is not to uncommon from a normal usage without malicious intent.

Element	PayloadData	OutgoingBundle	Intent
Byte	524288	525088 (+800)	525480 (+392)
Byte	124288	125056 (+768)	125444 (+388)

Table 3.1: Size of a bundle during BundleSpam

Opptain size in megabytes				
After	Install	BundleSpam	last bundle should expire	24 hours
Device 1	10	2300	2300	2300
Device 2	17	1530	1530	1530

Table 3.2: Opptain size at different times

Unique Identifier Duplication

An application can define several components in the *AndroidManifest.xml* file which are enforced to be *unique* [RAA+14] [JESS15] among all installed applications by the Android operating system. All following statements are true for Android versions since API level 16 or later. Older versions might show the same behavior but are not considered as they are not supported by opptain.

A Content Provider is uniquely identified by the *android:authoritites* attribute and starting with Android 5.0 (API Level 21) permissions are uniquely identified by their *android:name* attribute. All other attribute values are not considered for the uniqueness. In practice, trying to install an app that has a Content Provider or a permission which is already claimed by an installed app will cause the installation to fail. An *"App not installed"* message is displayed on the for testing available devices HTC One M7 and Huawei Y360-U61 as well as in the emulator. Looking at the system log, either an *INSTALL_FAILED_CONFLICTING_PROVIDER* or an *INSTALL_FAILED_DUPLICATE_PERMISSION* error code followed by the duplicate Content Provider respectively permission and the package name of both the failed and already installed application are logged by the *PackageManager*. Users unaware of this will not be able to track the source of the problem. The error for a duplicate *applicationId* is similar, the reason for the failure is different tough. Android will think the app is an upgrade for the installed app with the same Id and then cancels the install process after checking the signatures, that are different.

These scenarios are essentially an *installation denial*, targeted against the availability. With the intent to also distribute opptain over unofficial channels in mind an adversary might use this to stop the installation and spread of opptain.

3.2.4 Eavesdropping

Eavesdropping in the context of Android is the act of passively intercepting messages or data of another app. When it comes to eavesdropping the communication between opptain and a third-party app, an adversary has mainly two options. One is to exploit unsecured implicit bundle Intents like in Section 3.2.1. This only affects third-party apps and not opptain itself, since opptain limits the Intent to the package it is sending to. The other option is to scan the shared external directory for *PayloadFiles*. While for the first method the *de.opptain.waitress.permission.COMMUNICATION* is needed, the second method only needs the *android.Manifest.permission.READ_EXTERNAL_STORAGE*, which is an often requested permission and thus the eavesdropper may seem completely ordinary and unrelated to opptain for the user. For instance, it could be camouflaged as a music app.
The *attack vector* in the second approach is established through the way opptain handles the sending of large files and can injure the confidentiality of a message. As mentioned in Section 3.2.3 Intents are size limited and therefore files of around 1 megabyte or larger are placed in the shared external storage so that opptain can access the PayloadFile of a third-party app and vice versa because apps not signed by the same signature cannot access the others private directory. The bundles will then only contain paths to these files. For public files, that are already stored in the external storage this is no problem but for private files it is since they must be copied to the external storage and as a consequence will be exposed. Furthermore, even *NetworkBundles*, which are only forwarded, are stored externally although they will not be delivered to any installed app. An adversary has to just check the corresponding external directories for new PayloadFiles at this point (see Figure 3.3).

As a proof of work, to show that this threat can easily be exploited, an *Intercept* functionality was build into the TestApp. It can be started from the launch activity via the *Intercept button*. The opened activity will then scan the *apk directory* in opptain's external directory. The reason for that is, that opptain saves an APK copy of every registered app in there to possibly share it with the build in *APK-Sharer*. This scan is just done for performance reasons, another way would be to just parse every app manifest for used opptain permissions to obtain the third-party apps. All found apps will be shown in a *ListView* from which they can be selected to list their external storage content. A more sophisticated malware would implement some mechanism to be notified each time a new file is created and copy it if it seems to contain promising data.

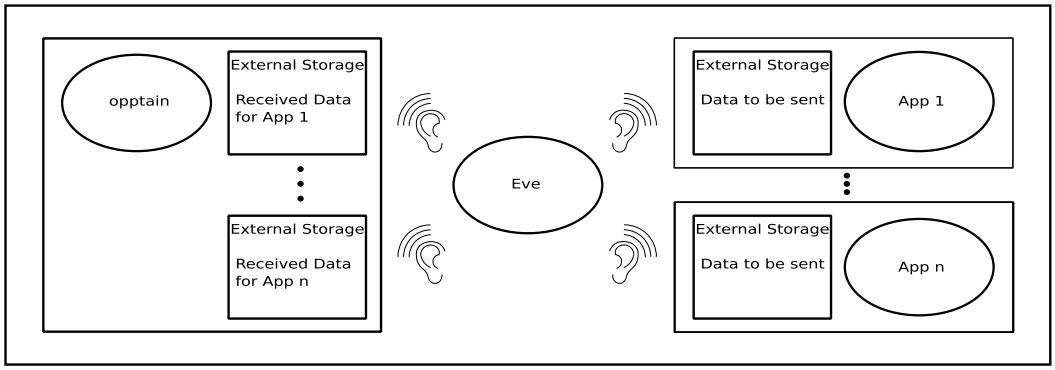


Figure 3.3: Eavesdrop from third-party app files

3.2.5 Task Hijacking

Task Hijacking [RZX+15] attacks the Android multitasking mechanism administrated by the *Activity Manager Service (AMS)*. It is enabled by the fact that Android allows Activities from different apps to co-reside in the same task. An attacker can now abuse task options to let his malware reside in the victim apps task and hijack the user session of the victim. Possible harms can be stealing of sensitive information, user privacy infringement, DoS of the device and apps to name a few. All apps are vulnerable to the implied risks of Task Hijacking, including system apps. In general the attacker will either launch a spoofed UI when a user starts an app, without the user's awareness, that is phishing or effectively lock other apps from usage and act like some kind of ransomware.

AMS is a system service that supervises all *Activity instances* and controls their life cycles by organizing them into tasks to manage them and to support multitasking. It is one of the most critical system services in Android because AMS also supervises service components, Intent routing, broadcasting, Content Provider accesses and app process management. A task [Goo17i] is a collection of activities visited by users in the same session. The Activities are stored in a back stack inside of the task. There is always one foreground task, namely the one holding the current displayed foreground activity, while the rest are background tasks. Since activities from different apps can co-reside in the same task, a malware can manipulate other Activities, if they are placed into a task where the malicious one gets the program control by the system.

The situation, in which we have a back stack with instances of both a malicious and a benign app, is called *hijacked task state*. It is a result of a *hijacking state transition (HST)*, which ex-

ploits task control features to create a HTS. A state transition either pushes a malicious activity on the target task's back stack or the victim is tricked to be pushed onto the malware back stack. The set of task control features used as exploit conditions are the activity attributes *taskAffinity*, *launchMode* and *allowTaskReparenting* and the *FLAG_ACTIVITY_NEW_TASK IntentFlag*. The attributes are defined in the <*Activity*> tag inside the manifest whereas the flag is set at runtime. A description of the their function can be found at [Goo17i]. The exploiting events after a successful task hijack are Activity callback functions and Framework API functions. In particular the *callback function onBackPressed()* can be overridden by an attacker. As a result the back button will trigger the attackers code, instead of finishing and resuming the next Activity on the stack. The Android Framework provides functionality to create *tasks with pre-populated activities*. On the one hand the utility class *TaskStackBuilder* and on the other hand the *API function call startActivites()*. This allows attackers to build back stacks that will be started as new task with previously not displayed activities.

Task hijacking can *breach UI Integrity* with spoofing attacks and phishing attacks (back hijacking) and *UI Availability* with being able to prevent uninstalls and act as ransomware. The former does not really apply to opptain because no external activities are used and the latter concerns every Android app. However, the breach of *UI Confidentiality* is possible. Since Android 5.0 apps can get information about their owned tasks and all contained activities without permission. The key point is, that the root of a back stack is considered the owner. Consequently, malware can run a task, with a malicious activity as the root, in the background with its task affinity set to the opptain package name. As long as it is started before an instance of opptain, opptain will get pushed onto the malicious task on start, because the launcher starts apps with the *NEW_TASK* flag. To remain unnoticed the malware can set the activity attribute *excludeFromRecents*. Next, if the QR-code Activity is launched the attacker can launch a spoofed version, that shows his public key. This results in a confidentiality breach for messages that are sent to the targeted user via the wrong encryption key. Another threat is that the malware always knows when opptain is started. This knowledge can be used to exploit that opptain only checks the integrity of APKs for the APK-Sharer while launching in the *InitialWorkActivity*. When the activity finishes, the APKs can be rendered useless or replaced with repackaged versions.

3.2.6 Clickjacking

In a *Clickjacking* attack [WBDJ16] the target app is covered by a malicious app that puts an opaque or very low transparent UI on top of the screen that is *"transparent"* to user input and that wants to trick a user to click on a specific position. The user input will be redirected to the victim UI, causing actions that the attacker would not be able to trigger himself. For example, a click on a button or link. All this happens while the user is expecting to interact with the on top displayed window.

For this intrusion to take place unnoticed, the target window must be launched by the adversary after the malicious UI, without coming to the front. To achieve this the attack window can be a floating one which has a higher priority to stay on top of normal application windows. Also to not make the user suspicious after he performs an input action the malicious window must react promptly to the action. The problem hereby, touch events are not noticed by the attacking component as it is transparent to them. Consequently, workarounds have to be figured out. The most plausible ones the attacker will use are callbacks from the victim activity, broadcast receivers for broadcast intents from app or system that inform about user action or system state change and observing accessible, known to be changed databases.

Following the functionality of the attacker component will be discussed. *Floating views* are normally used to display UIs of multiple apps together, for facilitating cooperation between apps and simplifying user operations. They are higher in priority than normal app views, but require the uncommonly used *SYSTEM_ALERT_WINDOW* permission. With one exception being the *Toast type* requiring none. When created by a service, floating windows can still float on screen even when their host app is brought to the background and untouchable floating windows, running in a service, can bypass the Android system enforcement that blocks the penetration of user input between windows created by different activities.
In comparison to an Activity, where associated views are loaded automatically, views from a Service need to be added explicitly with the *addview API* of the *WindowManager*. The important LayoutParams available are: width/height, position, general window type and behavioral flags. The size and position parameters are set to occupy the full screen, so that the user does not notice the attack by operations running underneath the window. The window type must be chosen with a higher priority than normal windows, as explained above, and must be utilizable by the malicious app. Therefor the layer value, determined by the *Z-order* of the window management, has to be greater. The two main window flags to consider:

FLAG_NOT_TOUCHABLE since the floating window passes all touch events to the target, and the optional *FLAG_NOT_FOCUSABLE*; not advised in case the target has a focus-able element, for instance a text field.

Considering the *Target Window*, it can be from any third-party app, system app or other system UI. Security-sensitive functionality that normally requires specific permissions to be used programmatically (e.g. system settings, camera) can be used manually by tricking the user to click on a button in a target app that offers the function. Since the invocation of a target activity containing the window usually does not need permissions, Clickjacking can indirectly launch attacks respectively use features without declaring otherwise necessary (sensitive) permissions, which hence are bypassed. The target is launched either programmatically with Intents/Schemes or manual, which is more complex and less stealthy. Against opptain, Clickjacking could be used to clear the Wish-/Blacklist or databases. This would require multiple clicks to which the floating window must respond, and is therefore hard to realize. In combination with Task Hijacking it might be easier but this must be tested. The main reason for this attack to be mentioned is the proposed access control system in Section 4.1.1.

3.2.7 Binder Attack Surface

Applications in Android execute inside of *sandboxes* and therefore depend heavily on *Inter-Process Communication (IPC)* to interact with the system and other apps. The Binder driver, as cornerstone of IPC, can be used as an attack gateway [FS16]. More specifically the Binder interface is exploited and not the Binder directly, as its mechanisms itself appear to be quite secure.

A *Binder* provides a message-based communication channel between two processes that functions similar to a classical client-server architecture. The client sends a transaction and then retrieves the response from the remote server via the Binder framework. Before data can be send over the Binder driver, it must be marshalled into a serializable data container object, called *Parcel*. The receiving side then de-serializes and un-marshalls the Parcel to get the data. This allows *Binder-based Remote Procedure Calls (RPC*) in which the parameters and return values are send to remote server and client respectively. The aforementioned is in fact often implemented by Android system services.

Android provides the *Android Interface Description Language (AIDL)* for developers to define an explicit RPC interface that client and server will have to follow. From this interface, Stub and Proxy classes can be generated automatically. The classes implement low-level Binder library details and ensure that the declared RPC parameters are serialized, send, received and de-serialized correctly.

The Binder attack surface fundamentally allows an attacker to directly *inject faulty transactions* into (system) services by manipulating the Binder interface, and hence bypass all client-side sanity checks. Normally, this should not be an issue when the server side checks the transactions before continuing to process them, but this is often overlooked. Especially the help of AIDL to generate an interface with the required IPC stack makes developers incautious to the security risks, as they are not confronted to deal with the details of RPC and Binder. Resulting consequences can range from Denial-of-Service (DoS) attacks to privileged code execution, depending on the payload and the target of the malicious transaction. Attack vectors enabling the attacking of the Binder interface are:

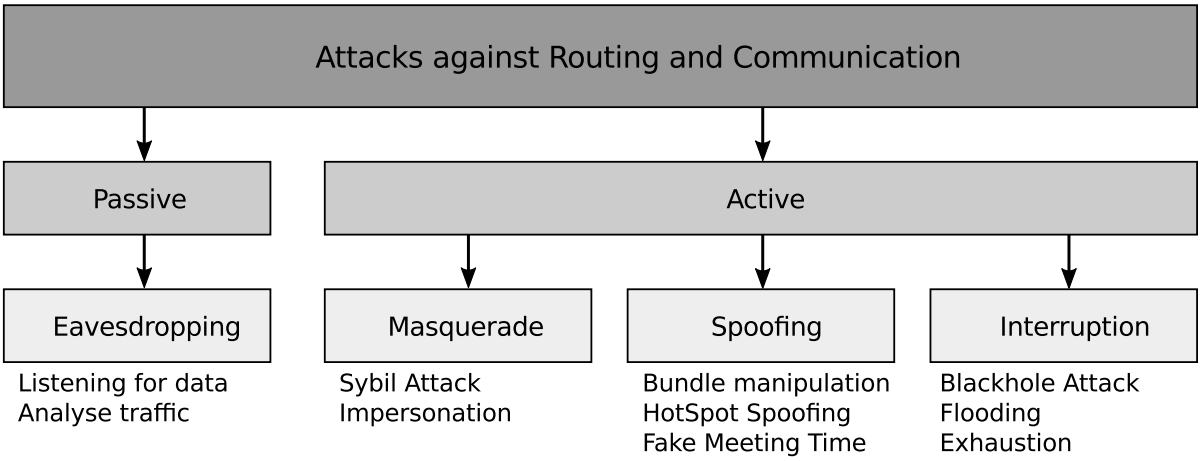
- Manipulation of not exposed RPC methods and parameters of public APIs that are frequently left unchecked, causing their (de-)serialization process to be unprotected with no sanity-checks.
- Bypass every sanity-check in public APIs with a forged malicious transaction, that is directly injected into the Binder. The generated Stub and Proxy classes are ignored.
- Exploit the serialization process of certain data types to create poisoned Parcels that provoke a bad recursion in the de-serialization process on the server.

The second point is also exploitable in opptain. Third-party apps can bind to the *OpptainRemoteService* to invoke some methods that are defined via an AIDL. On a successful bind the Binder to communicate with the service is handed to the binding app. Malware can then pass transactions with invalid parameters to the service, via directly injecting it to the binder driver, to provoke a *RemoteException* that will crash the service because the exceptions are not handled by opptain. Additionally, the *passBundle* method, that takes an *OutgoingBundle* and converts it to an *OpptainBundle*, does not check the *timeToLive* and *applicationPackage* attributes to be correct and in bounds.

3.3 Against Routing and Communication

In this section we discuss attacks and threats which target the routing and communication capabilities between peers in Opportunistic Networks and that are applicable on opptain. In contrast to an adversary from the previous section, the attacker will have full access on the system from which the attacks are launched, with exception of when a repackaged opptain on a victim device is used. The attacks, Figure 3.4, will be carried out by malicious nodes that run a self developed network app, which implements the opptain protocols, or that use a repackaged version of opptain. The attacks and threats are often similar and differ only in some points and details. The differentiation can be the motivation, impact, channel or used data of an attack.

Figure 3.4: Taxonomy of OppNet attacks



3.3.1 Blackhole Attack

Blackholes are *malicious nodes* in Opportunistic Networks that attract packets from other nodes by sending them forged and manipulated metrics (e.g. fake routing info) [KM14]. This non-genuine metrics are designed in a way that will cause other network members to evaluate the malicious node as an excellent forwarder. As a result, many members will favor the evil node over benign ones in the choice of choosing a next hop to whom they send their data packets to. Once the Blackhole *receives the packets*, they *can be dropped* or used for more advanced attacks.

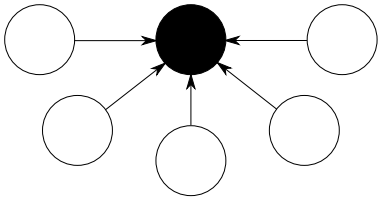


Figure 3.5: Schema of a Blackhole attack

This kind of attack is a *source of great disruption* for the already by definition challenged Opportunistic Network and degrades the integrity of routing information. To setup the attack, the malicious node first has to *analyze the network* and its routing protocol. The more knowledge collected, the better will be the execution and success of the attack. Valuable information are: how does the handshake work, how large are packets, is data encrypted or important routing variables to name a few. Possible ways to obtain this knowledge are by reverse engineering (in the case of opptain also possible with app repackaging) the program that runs the network or by a passive attack like eavesdropping and analyzing the network traffic. When deployed, the malicious node has to operate normally for a short time to collect current information about the network participants and potential targets in vicinity. This phase allows to learn about the needs of the local environment and to derive the information that must be forged for a high success rate. Subsequently, if a connection with another peer occurs the malicious one *lies about its routing information* to intercept packets for a requested target node with the intention to drop them.

Greyhole As mentioned above, this Blackhole attack can be more advanced, to be more specific it can be extended to a *Greyhole* attack [SE07]. In this specialized type of attack the node can choose between forwarding and discarding packets, based on several attributes like time, location and destination. The decision for a behavior depends on the intended result. To isolate targeted nodes, the packets will be dropped. To not attract attention in an area with high node activity or noticed intrusion detection efforts, normal forwarding and routing will be performed. Another attack besides dropping would be to *purposely forward the packets to bad forwarders*. This means, instead of sending network bundles to nodes with a higher chance of meeting the destination, the data will be send to nodes that are further away. The delivery will be even more delayed and the packet indirectly consumes additional network resources because it must be relayed over more nodes than normally. Because of the *Time-To-Live* in opptain a bundle could also expire before it can be delivered and thereby stress the

network by wasting resources that could have been used for legitimate data dissemination.

3.3.2 Sybil Attack

A *Sybil node* is a fake identity in the possession of an adversary [KM14] [SXB07]. In a *Sybil attack*, the attacker generates multiple Sybil identities that also might impersonate other nodes, like in the *HotSpot Spoofing* attack. Most of the time the attacker uses one device and switches between the identities but he could also use several devices dedicated for certain identities, to have multiple Sybils active at the same time. Sybil users are easy to generate in Opportunistic Networks because a *one-to-one binding* between an identity and a real entity cannot be ensured. Therefore, each new user enjoys the benefit of the doubt at first, for the network to function in environments with sparse node activity and to enable network growth.

Through the Sybils, a single agent can obtain a higher influence in the system, evade bad reputation or masquerade his actions. For example, the Sybil attack can be used to increase the reputation of the attacker, if the Sybil nodes route the packets among each other, or just write each other to the network path, to generate meetings and *artificially high success rates*. This will eventually lead to an attraction of more network traffic. But Sybil users might also not write themselves into the delivery path, so that the receiver will thing he is close to the sender of the data. This would cause a spreading of wrong routing information into the network, towards other benign nodes, with originally good intentions in mind. Additionally, systems and protocols that focus on *redundancy mechanisms* to resist security threats from faulty or malicious devices are weakened. For example, routing protocols like (binary) spray and wait normally only use redundancy as means to ensure packets are delivered from A to B in an efficient way and to not rely on a single network participant, who might forward in some strange fashion, like unnecessary loops. But if multiple colluding Sybil nodes take over the packets, it is as if the packets were only forwarded to one misbehaving node. Equally bad impact can be achieved against reputation systems as they rely on direct and disseminated user ratings to evaluate a peer. Without a solution to check the authenticity and integrity of the received ratings, Sybils can simply lie to manipulate the reputation.

It should be clear by now, that Sybil nodes can degrade the performance of networks by a good amount. Detection systems to spot and avoid them are essential for OppNets.

3.3.3 Denial of Service

Denial of Service on the networking layer perturbs the normal communication between nodes. It can be aimed against the network in general or specifically target peers.

Flooding

Flooding [SXB07] is an attack that saturates the network with an abundance of *fabricated packages*. The bundles can address a real node which in turn will also be flooded or an invalid to keep the bundles being forwarded in the network for a long time. The performance of the network will decrease and valid packets might get deleted on poorly equipped devices, in an attempt to free resources for the flooded packets. Its hard to identify and trace the attack because the messages look like normal ones and are also received as valid. If this attack is performed in quick succession *(rushing)* with bundles containing false routing information in combination with multiple nodes (Sybil scenario), the effect of estimating the best forwarding hop on contact gets impaired because the stored routing data will be unreliable.

In the current implementation the opptain network app has no feature to counterfeit this approach. There is no trust or reputation system to block communication or contact with suspicious peers nor are there any thresholds to limit the receipt of bundles after a schema.

Exhaustion

Exhaustion stresses a node or resource so much that it cannot be used by other participants or in the worst case eventually crashes [KM14]. An adversary can force repeated collisions and/or continuous re-transmission to occupy the channel. This can be done by actively sending corrupted data that will cause re-transmission, do not sending ACKs which consequently also causes re-transmission or *SYN flooding* of a node in Server/Hotspot mode to provoke an exhaustion of the sockets in the target node, achieved by creating many half-opened TCP connections. Especially weak nodes will crash fast under the latter.

SYN flooding is possible in the opptain networking application. The peer that is hosting the HotSpot and therefore opens a *ServerSocket* to let other clients connect, does not check

or limit the amount of connections per peer, As can be seen in Listing 3.1, taken from the *ServerThreadTcp* class. A possible reason for the application of this exhaustion is to disconnect other peers and afterwards trick them to connect directly to the attacker server.

Listing 3.1: Unchecked client connection

```
while (mIgnoreWhileWarning) {
    final Socket clientSocket = mServerSocket.accept();
    InetAddress inetAddress = clientSocket.getInetAddress();

    final ClientInfoTcp clientInfo = new
       ClientInfoTcp(mDispatcherThread, clientSocket, this);

    KeyPersistence keyPersistence = KeyPersistence.getInstance();
    Handshaker handshaker = new Handshaker(clientInfo,
       keyPersistence.getSigningKeyPair(), false);
    handshaker.handshake();
    ...
}
```

3.3.4 Bundle Manipulation

One may think the data sent between devices using opptain is safe considering that the handshake protocol first authenticates *paired peers* to authenticate each other and then exchanges the data in an encrypted format. While this assumption of security is true in the case of former described *point-to-point connections*, it fails for the data residing in the met peer after the transfer. This is because, in its current state, the implementation of OpptainBundles can not guarantee confidentiality, integrity and authentication for the routing information and data it holds. This is due to the fact that the bundle is currently *not signed nor is it end-to-end encrypted by default*. Following the bundle attributes are described in more detail and it is evaluated what the results and consequences from tampering the values are.

The *BundleId* is a *32 Byte Id* used to uniquely identify a bundle, so if an attacker changed this Id by only one byte it would lead to opptain believing it is a new Bundle. This can have multiple unwanted consequences ranging from harmless to dangerous. The harmless case

would be a user who sends packages multiple times with new Ids to increase the chance of a successful delivery. Although harmless in the scope of security aspects this behavior puts additional stress on the network, that can decrease the overall performance, and is unfair in comparison to honest acting users. Dangerous is it when a malicious node starts to replay bundles by just changing the Id.

Origin is a *32 Byte SodiumDeviceId*, it is used to identify the device which created the Bundle and at the same time it also acts as the public key. If it is changed the sender is spoofed. This can be used by a *Man-in-the-Middle* to receive the answer to a request by the original sender. This was also tested with a simple test app called ABE. The app allows to *chose being Alice, Bob or Eve*. During the test three devices were used with each running a different user. Additionally, the Eve device had a modified version of opptain running that changes the *Origin* of all outgoing *NetworkBundles* to its own *DevideId*. When either Alice or Bob sends a first hello message it will be send to Bob respectively Alice who will respond with some sensitive data. In the test Alice and Bob were not in range of each other, so that the bundle was forwarded to the intermediate Eve first. Eve then modifies it as described and later receives the answer intended for Bob/Alice.

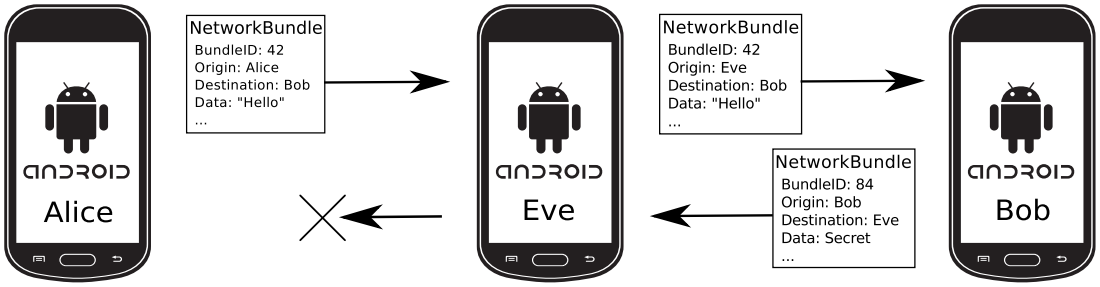


Figure 3.6: Man-in-the-Middle attack through Bundle Manipulation

Destination is the *SodiumDeviceId*, it is used to identify the destination for which a bundle is created. It can be modified to make a bundle never reach its intended node.

CreationTime and Time-to-Live are used to calculate the expiration time of a bundle. An attacker can change this values to either make a bundle expire early or to increase its life-time.

ApplicationPackage the app which created the bundle. Malware can just lie about its package to communicate without registration. Moreover, when a bundle is forwarded completely it is delivered to the app, if it is installed. A threat is that this happens even when the destination

is not reached, yet.

PayloadData, PayloadFile, PayloadInfo and PayloadMeta hold data and various info about the Payload. They can be altered to either cover changes that injure the integrity or to make the data useless on purpose. If files are not encrypted they can be simply read but even if they are they can still be manipulated and rendered useless.

NetworkPath and AdditionalInformation are information fields used by opptain, that are intended to be changed during forwarding. The network path can be tampered arbitrary. In the case of *(binary) spray and wait* the *quota* can be adjusted for various reasons. Decrease it to reduce the spread of copies and lower the delivery chance. Or set it to a high amount to flood the network. In general the possibility to add data to this attributes can intentionally or not inhibit the delivery of the bundle to a third-party app. This is due to the limit of the previously discussed *Binder Transaction Buffer* in Section 3.2.3. It can lead to transactions being marked as successful even though the app never received the bundle.

3.3.5 HotSpot Spoofing and Impersonation

HotSpot Spoofing is an attack in which a node impersonates the identity of another node. Particularly the *HotSpot SSIDs* from previous connections are used to masquerade as a genuine node. To be even more stealthy, the *MAC* can also be set to the copied user's device one; requires root on Android, but an adversary will have this permission level most certainly. The SSIDs that opptain uses are of the following format: they begin with a *"HS"*, then comes the *24 byte fingerprint* of the *DeviceId* and the *last 6 byte are used as flags*. Since the attack vector is the HotSpot functionality the attacker is the peer that opens the HotSpot.

When a new user comes in proximity of the spoofed HotSpot, he will try to connect to it as client. Either he connects because he does not know the HotSpot or because the *identity* behind the fingerprint is on the internal *wishlist*. Once both devices are paired, the client peer will initiate a *handshake* to exchange encryption keys to start a point-to-point encrypted session. At this point, the spoofing will be revealed, since the checking of the challenge, send with the handshake, fails. The client will disconnect and move on to the next access point. The reasons for this attack are mainly to collect new *DeviceId's* and to trick users into unnecessary connections to make them miss genuine HotSpots. The latter could even lock the user in place, if the attacker is the closest node and manages to quickly spoof another

identity that the user does not know before the user scans for new HotSpots. This works because a node will always chose the *server with the best signal strength* as long as there are no servers for which the user has packets or for which a previous connection did not finish. Thus an attacker will most likely not use smartphones but rather *Wi-Fi devices* with a strong signal that *emulate opptain*. Especially users new to an environment with no contacts to deliver data to are susceptible. A blacklisting of *fingerprint/deviceId and MAC* will not help as they continuously are changed by the attacker. The user can only escape this connection if he walks out of range or a server with higher priority comes in range.

In addition to these both attacks, a more severe one is available. An attacker must not use the deviceId corresponding to the fingerprint. There is no check in the handshake that tests if the fingerprint of the DeviceId is equal to the one listed in the SSID. The handshake will be successful and the adversary can communicate with the victim. *Blacklisted devices* can use this to connect because again only the fingerprint in the SSID is used to control if a device is in the list.

3.3.6 Fake Meeting Time

In opptain, the client defines the *ConnectionID* for a session. He can chose the same Id for every connection because the server does not check if a Id was used previously. Consequently, the database will be filled with multiple entries containing the same *ConnectionID* and all of them are updated when a connection ends, which is also not checked. Peers with a *short connection* time can thus fool the victim to think they are a frequently encountered node. For instance, a node that connects for a couple of seconds and then returns after 10 hours to again connect only for a couple of seconds, will have a meeting time of 10 hours. *NASGL* uses the meeting time to calculate decent *neighbors for aggregation*. The attacker can in this way artificially increase his time to be evaluated as decent and be integrated into the aggregation. This will cause wrong routing information to be spread that convey a false local network view.

Miscellaneous

The following points are feasible to be a threat for confidentiality, integrity and authenticity.

- After the *Time-To-Life* of a forwarded bundle, that was not delivered to any installed app, expires, the bundle is deleted correctly from the database but the external PayloadFile remains.
- When the user is tricked to reinstall opptain, the old *key pair* is lost. How can other network participants be informed about the invalid deviceId? Same problem when a keypair was stolen by malware. No mechanism to revoke a public key exists.
- A received *PayloadResponse* is not checked for its size. An adversary can send data that is larger than originally advertised. The size should be compared with PayloadInfo.
- Uninstalled apps stay registered in opptain.
- Keep opptain active with TTL clearing alarms to cause a high battery usage.
- The handshake can be interrupted by jamming faulty bundles with wrong challenge answers; its hard to achieve this attack but possible.
- *Hash* of a completely received file is not checked.

Chapter Conclusion

In this chapter selected threats and attacks against Android-based Opportunistic Networks were researched in literature or derived from the opptain implementation. The attacks and consequences were mainly discussed theoretically in this thesis. Testing of them is a task for future work. Moreover some of the Android attacks are enabled through the user installing malicious apps. In this scenario, they take a share of the blame for possible consequences. Table 3.3 summarizes the main targets of each attack and states if it was implemented during this thesis.

Attack summary			
Attack	main targets	implemented	app
Intent based	1, 3, 4	no	
App Repackaging	3, 4	no	
DoS Android	2	yes	TestApp
Eavesdropping	3	yes	TestApp
Task Hijacking	2, 3	no	
Clickjacking	1, 4	no	
Binder Attack Surface	2, 4	no	
Blackhole Attack	2, 4	no	
Sybil Attack	3, 4, 5	no	
DoS OppNet	2, 4	no	
Bundle Manipulation	3, 4, 5	yes	ABE
HotSpoot Spoofing	3, 5	no	
Fake Meeting Time	4	no	

Authorization(1) Availability(2) Confidentiality(3) Integrity(4) Authentication(5)

Table 3.3: Attack summary

Chapter 4

Solutions and Mitigation

In this chapter solutions for the previously described attacks and vulnerabilities are discussed. Not all solutions will solve a problem completely but at least mitigate it and raise the effort for a successful attack. Some literature proposals for solving Android attacks require changes of system components. This is out of scope and not considered, since most users are not willing and capable to flash their device with a modified Android version. Only solutions that can be directly implemented in an application are reasonable.

4.1 Mechanisms against Intent based Attack Surfaces

In general, developers should be aware of the distinction of *inter- and intra-application communication*. Once the difference is understood, the separation of inter- and intra-application communication components into separate ones dedicated to just one of the both tasks should always be done, if possible.

Unauthorized Intent Receipt Solutions to stop Unauthorized Intent Receipt are quite easy to realize [KCHW12]. If a recipient is known, developers should make the Intent explicit by passing the *full component* name into the Intent. But even if the full name is not known the Intent can be narrowed down to the desired application by passing the package name to the corresponding field. Since in Android versions below 5.0 Intents that match multiple Services are delivered to one randomly, without telling the user which, a check should be

performed before sending. The check should query all possible resolving Services and check if the correct is present. Only if present, the Intent should be send, with the right component set. Alternatively a developer could make his app only available on Android 5.0 and higher.

Intent Spoofing First of all developers can review their code and verify that they do not expose any internal component with an *IntentFilter*. As a best practice the *exported option* should be always used and set to the components expected exposure level to prevent capability leakage [KCHW12]. Components intended for inter-application communication with the system or other apps by the same developer can be protected by a *signature-level permission* in the manifest file. In this case an Intent is rejected, if it is coming from a non system app or with mismatching signing (in most cases not from same developer). For app internal broadcasts the *LocalBroadcastManager* should be used (available since Android 3.0) instead of a dynamically created Broadcast Receiver.

Privilege Escalation There are some best practices that can reduce the chance for a confused deputy privilege escalation although they do not prevent it completely [RZZL14]. *Publicly available components* should always be protected by a permission that is mandatory to access it. Furthermore if a component invokes API calls it should always check that the requesting app has the required permission for the desired action.

4.1.1 Access Control

As discussed in the previous Chapter 3.2.1, the currently employed registration model for apps, that want to use opptain, is not safe. There is an urgency for a proper access control mechanism that handles the registration and all further interaction between opptain and any third-party application.

It is necessary to reliably identify the source of a request to be able to make profound decisions in the further course of actions. Hence Intents do not contain information about their sender nor can opptain trust an application without uncertainty to provide its actual *package name* in the Intent, another approach has to be found. Within the context of this problem, a *simple authorization mechanism* is developed, that acts as a proof of work for secure registra-

tion and considers the threats like *Task Hijacking* 3.2.5, *Clickjacking* 3.2.6 and *Intent based attack surfaces* 3.2.1.

Prevent Task Hijacking and Clickjacking The solution must be resilient to *Task Hijacking* and *Clickjacking*. With regards to *Clickjacking*, the Android framework offers a *touch-filtering mechanism* to protect windows from receiving any user input event when obscured by another visible window, by simply discarding the input. Specifically, the touch filtering can be activated by calling the *setFilterTouchesWhenObscured* function on a view, or by setting *android:filterTouchesWhenObscured* in a layout to true. The downside of this approach is that even benign apps will cause input blocks. That could be confusing to the user in case of a harmless floating view (e.g. a pop-out video player), if the app does not respond to interaction anymore [WBDJ16]. A possible Task Hijacking attempt can be discovered by *checking the manifest* of all installed apps. If an app declares a task affinity for our app the user should be warned, so that he can uninstall the app if he thinks this app should not have the task affinity. Furthermore, the critical component should use the *singleInstance* launch mode, that will start the activity in a new task, into which no other activities will be launched by the system [Goo17i]. The component is always the only member in the task. This *thwarts hijacking state transitions* that would put the activity on a malicious task.

Implementation The previously used TestApp and ABE are used to implement and demonstrate the authorization. TestApp implements a service that handles the authorization and registration of third-party apps. It offers an AIDL interface with a registration method for apps, that must be implemented to communicate via IPC. The reason for this approach is the possibility to get the *UID* of a calling app with *Binder.getCallingUid*(). This enables to find out the package name via the *PackageManager*. If an app tries to use the service for the first time, the service opens an authorize activity. The activity shows the package name of the application that wants to register and asks the user to allow or deny the registration. This is done via *Accept* and *Deny* buttons. Leaving the activity without any action cancels the registration. If the user authorized the registration of the app, it can start to use the functionality, otherwise calls will be ignored and not executed.

The further communication between service app and third-party app can be carried out with either a *Binder* or *Intents*. This was not yet implemented due to time limits, but an explanation follows. The *Binder* will be returned to the application after a successful registration.

This means the Binder with the registration method and the one with the functionality are separated from each other. The intention is to make the service more resilient against *Binder Attacks* 3.2.7 by not exposing it publicly. The disadvantage of the binder approach is that multithreading must be handled manually. For the *Intent* approach, a *token* will be returned by the registration method after a successful registration. From this point onward, every send Intent of an app must be authenticated with the token. The token cannot be send with the Intent itself though, since a careless app might use implicit intents that can leak the token, as in Section 3.2.1, which a malicious app in turn might exploit to communicate without registration. Instead the token will be used as a *salt for a hash function*: before the Intent is send an attribute will be hashed and placed into an extra field. When the service app receives the Intent, the attribute will be hashed with the token that belongs to the app claiming to be the sender. If the hashes do not match the Intent is discarded. The attribute for the hash can be the *BundleId* or a *nonce*, that is send along with the token. Since the token is only used internally, it should not be leaked on accident anymore; might be leaked on purpose though. In comparison with the binder approach multithreading must not be minded by the service app.

Improvements There exist a couple of improvements that can be added to the service app. The authorization activity can be extended with a *code input* field that must be filled before an action can be chosen, so that users do not just accept it blindly like permissions. System *broadcasts for uninstall* events should be scanned to know if a registered app was uninstalled and thus its access must be revoked. The app could offer options to let the access expire after a certain time. Especially the token approach should be implemented in this way, so that potentially unnoticed token leaks will be mitigated. Another option would be to let the user define a blacklist of instantly blocked apps. Finally, it should be no problem to extend the opptain networking application with this feature. Malicious apps can then be stopped to access functions they are not supposed to. For example, *getBundle* and *revokeBundle* will only execute when the requested bundle belongs to the app.

4.1.2 Intent and Bundle Threshold

A Solution for *Intent DoS* is to limit the amount of intents that are received and processed afterwards. An implementation via threshold values seems reasonable but has to be carefully

tested and evaluated, because it is difficult to construct a satisfactory defense schema against the Intent DoS in practice. The attacked component or resource is often shared and thus countermeasures with weak restrictions will not be enough to stop the attacker from reducing the availability. On the contrary, a too restrictive countermeasure will limit the component access for genuine usage.

A *bundle threshold on the network side* is also necessary since every packet that is addressed for us or that is a broadcast will be accepted. In an environment with high node activity, the device storage can fill quickly through this circumstances. In addition to the threshold, it would be desirable to have a sophisticated *rating algorithm*, that can efficiently evaluate if a bundle or Intent is bad (e.g. spam). Bad data would not be processed further and thus not increase the threshold count. This could help to reduce the amount of genuine bundles and intents that are being rejected because the threshold was reached.

4.2 App Verification

Android requires from each app that it is signed by the developers certificate [Goo17c]. Unsigned applications will already fail at installation time. This enables the verification of an installed application by comparing the certificate with the authenticated certificate from the developer.

Self checking App Repackaging can be detected in this way. A solution against naive attackers [Sco17] is therefore to implement a *signature check* into the app. On each application start this check will be executed to verify the current signature is associated to the release *developer certificate*. The certificate signature must be stored in a static hardcoded string. Signatures returned by the PackageManager can then be compared to the string at runtime. This solution is weak considering experienced attackers who will simply remove the check in the repackaging process.

Third-party checking A more promising proposal to check the signature is through the *opptain API*. Specifically third-party apps can check after installation or after an opptain update if the available opptain version is genuine. The API will provide the method along with an *embedded release certificate*. Granted that the signatures do not match the third-party app should inform the user about the tempered version. Of course this solution only works if

the third-party app is installed from a trustworthy source, implementing the official API. A suggestion for the method is seen in Listing 4.1.

Listing 4.1: Validating opptain certificate

```
boolean validateOpptainCertificate(Context context) {
    PackageManager pm = context.getPackageManager();
    PackageInfo packageInfo = pm.getPackageInfo(opptainPackageName,
      PackageManager.GET_SIGNATURES);
    byte[] cert = packageInfo.signatures[0].toByteArray();
    InputStream is = new ByteArrayInputStream(cert);

    CertificateFactory cf = null;
    X509Certificate c = null;
    try {
       cf = CertificateFactory.getInstance("X509");
       c = (X509Certificate) cf.generateCertificate(is)
    } catch(CertificateException e) {
       return false;
    }
    return c.equals(opptainCertificate);
}
```

4.3 Trust and Reputation

To tackle the problems introduced by *malicious nodes*, *trust and reputation* metrics can be applied to evaluate if data should be forwarded to a particular node or not. First of all, the distinction between the two terms must be clarified, as both are often used interchangeably by mistake. *"Trust is a particular level of the subjective probability with which an agent assesses that another agent or group will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action"* [Gam88]. *"Reputation of an agent is a perception regarding its behavior norms, which is held by other agents, based on experiences and observation of its past actions"* [LI04]. *"In contrast to trust, which tries to predict a future action, reputation is a passive property depending on past actions"* [TL09]. Both metrics can be

used as a substitution for *certificate authority (CA)* systems, which cannot be implemented in infrastructure-less decentralized Opportunistic Networks, to offer a solution for assuring a certain level of safety in the interaction between possibly unknown users.

4.3.1 Trust

Trust is a metric effective for thwarting Sybil and impersonating nodes. That is because Sybil nodes often have fewer trust relations and lower activity times than normal users, making them detectable or ignorable due to low trust values. A trust form appropriate to use in opptain is *social trust* [TLA10]. In comparison to other trust approaches, it is not based on network interactions but rather on human social interactions. It can be distinguished into two different approaches, explicit social trust (*te*) and implicit social trust (*ti*), by leveraging the social network structure and its dynamics.

Explicit social trust *Explicit social trust* is derived from consciously made friend ties by building a robust graph of paired users. Direct friendships are established through users, that meet each other face to face due to their mobility to authenticate their identities in a *secure pairing*. During the pairing and in all future encounters the users will exchange all their friends collected so far. This friendship data is then used as the basis for the calculation of a *tree-like friendship graph*. The graph is rooted at the users node at *level 0* and branches off to a *maximum of n levels*, with *n* being the *maximum hop count*. Edges always represent a hop to the next level and are sequentially, this means edges that would lead to a node on the same level or skip levels are ignored. The construction is implemented as a modified breadth first search algorithm. The depth and inter-connectivity of a node in the graph are used as parameters for the trust function, which is designed to decrease the trust with increasing depth and to increase it for well connected nodes in the graph. Direct friends have a *trust value of 1*, that the algorithm progressively propagates through the graph. The trust te_j that a *node j* on the next level gets, is depending on the amount of children the parent had, the number of parent nodes and the hop *distance of j from the root*. To ensure a decrease of trust in sparse graphs, a degradation factor, that simulates a minimum amount of children, must be used. Additionally all indirect nodes are capped to a maximum trust value of 1, as they should not be trusted more than direct friends.

Since the users *paired consciously*, impersonation is thwarted directly and other misbehaving or Sybil users can be identified quickly once they start malicious activities and attract negative awareness. So, explicit social trust verifies that the user behind an identity is genuine with honest intentions. Furthermore, as a CA cannot be assumed in an OppNet, the nodes take the task of *signing their friend's certificates* themselves, allowing for the transitivity of trust. But explicit social trust only trusts direct friends or very well connected peers fully. This is a trade-off between preventing malicious peers that manage to fool a trustworthy friend to gain full trust and not limiting trust to only consciously paired friends. The *disadvantage* of explicit social trust is the secure pairing that requires direct interactions and cannot be automated, leading to a *loosely connected graph, if only a few friendships* are established, that does not really convey trust to a lot of met peers.

Since *opptain already offers a secure pairing via the QR-Code* scanning mechanism only the implementation of the trust calculation must be added to employ it into the routing protocol forwarding decisions. The *complexity* for calculating the trust *te* is $O(b^d)$, as it can done while constructing the friendship graph, with *b being the branching factor and d being the depth*. The *communication overhead* depends on the amount of friends which is correlated with the branching factor, so $O(b)$. Considering the trust propagation the two most notable characteristics [TLA10] found out are: for many real and synthetic traces nodes with a trust value higher than 0.1 where *mostly just 3 hops* away and the propagated trust is only depended on the node degree, not the network size or structure, making it highly *salable*. Sybils from a single device always had lower trust than normal users due to their low social interconnection, leading to them not being selected.

Implicit social trust *Implicit social trust* leverages mobility properties to assign trust to unpaired wireless contacts that are encountered regularly in everyday life, so called *familiars*. This can be neighbors, coworkers or other students for example, which are often in close proximity but to whom a *close relation does not exist*. Furthermore, nodes are also classified by their *similarity*. Familiarity can be obtained from the accumulated contact time or contact frequency with peers that are in vicinity of each other, while similarity between two peers can be obtained by comparing the amount of common familiars they share. The set of familiarity values is exchanged with each encountered node and every time a node gets new data or when connection times get updated it *normalizes the values* and then builds a *graph with a two hop trusted environment*, based on the familiarity and the similarity of surrounding peers. The weighted graph represents a local approximation of the network usable for routing to trusted

peers. The implicit social trust ti_j in an encountered *node j* is calculated by simply adding both discussed values once they are acquired. The choice for a two hop environment stems from the uncertainty that further away hops are still in the near surrounding.

Implicit social trust conveys *trust based on the persistence of an identity* to not be a fast switching one. It requires no conscious user interaction, therefore it can be automated and in contrast with explicit social trust it does capture the mobility dynamics which can be used to establish trust. On the contrary, the *honest intent of an node can not be verified* with this trust, but as with the explicit one, misbehaving peers can be identified easily through the persistence that identifies them.

As opptain also tracks contact time and NASGL exchanges neighbors for routing decisions and aggregation, implicit trust should be easy to implement. The *complexity* for the trust *ti* has to be calculated for a *two hop environment graph*, leading to $O(b^2)$, with *b as branching factor*, correlating to the number of familiars. This can become really complex if there is no appropriate aging algorithm to keep *b* at a feasible size. As for *communication overhead*, it is the same as with explicit social trust $O(b)$ and *relies just on the number of familiars*. The maximal trust propagation is bound to a maximum value of 2, since familiarity and similarity are normalized to 1. The number of trusted nodes is comparable to explicit social trust. A notable finding from comparing the distribution of the trust values with the groups generated by community detection algorithms [TLA10]: the *structure of local communities is indirectly exposed* by the implicit social trust graph. For Sybils to gain a lot of influence via familiarity, they would have to increase their own familiarity while decreasing the familiarity by all other nodes which is hard to achieve let alone unnoticed. To gain high similarity multiple Sybils would need to collude, but even then transitivity of trust is only limited and they would need to be active at the same time.

Applications Using the social trust values in practice, a user will only communicate with nodes that have a *trust that exceeds a desired threshold* value. It could also be implemented with stages of actions that require a different threshold to let nodes with lower trust participate. One such application would be the forwarding of non sensitive broadcast messages. The social trust values can be *used independently* or can be combined for a *unified trust value* [TLA10]. When combined they can be weighted, $t_{total} = we*te + wi*ti$, to adjust the influence of a trust to the total combined one, depending on the situation and environment. For instance, in hostile environments a higher weight for the explicit social trust is more ben-

eficial. The biggest problem associated with social trust are environments in which the user has no social interactions [TLA10]. In this case there has to be either an *adaptation of the aging* to faster environments to build a new trust graph, or an alternative system, like deriving trust from reputation systems, that might not be as safe but at least provide some basic means for secure communication.

4.3.2 Reputation

Reputation is a metric to rate the behavior and quality of service of nodes in Opportunistic Networks. Adversaries that try to disrupt the network function will receive bad reputation for their actions and thus can be detected through their poor rating. Evil nodes like a *Blackhole* can then be ignored by the other benign network participants. A threat to reputation systems are forged ratings, that can manipulate the ratings of a peer to get a high rating, abandon bad reputation or evade responsibility for actions. For instance a misbehaving node can create ratings that indicate a genuine node is acting malicious to cover himself. The ratings for the reputation are retrieved from *subjective experiences* with a node and received as *second-hand experience* from encountered trusted peers that also observed the node in question, and are used to calculate the cooperativeness.

The realization of reputation systems in OppNets is hard, due to Sybil and liars that trick the system easily, if it is not implemented in a safe manner. In the research of this thesis no satisfying solution was found that would function in the whole network and simultaneously be resilient to attacks. But [LD13] proposed an approach that seems reasonable in combination with the *social trust* scenario. Although declared as trust-based framework, with the definition from the beginning of this section in mind, it is more of a reputation system. It uses so called *Positive Forwarding Messages (PFM)* as evidence of the forwarding behavior of a node and thereby can effectively detect two kinds of attacks: deliberately dropping data and arbitrarily forwarding data. In some sense, the systems assist each other. PFM can detect misbehaving nodes, that can then be identified through social trust while social trust mitigates Sybils and impersonation and therby effectively reduces forged metrics, increasing the accuracy of PFM. Additionally to that, *Encounter Tickets (ET)* [LWS09] can be introduced. ETs verify that two nodes really encountered each other. It is still exploitable, but an attacker would first need to meet the node, about which he lies, so ETs increase the attack effort.

4.4 OpptainBundle enhancements

The two main problems with *OpptainBundles* are that they are not signed nor are they encrypted by default. They are only encrypted, if the third-party app does it. Both issues can be fixed relatively easy and the tools to do so are already available in opptain. First of all, it must be clarified that not the whole *OpptainBundle* can be encrypted because almost all parts of it are needed for routing and forwarding purposes. Only the *PayloadData* and *PayloadFile* have to be ciphered by opptain. Similarly broadcast bundles cannot be encrypted. In order to safe computation power, *symmetric cryptography* is used for the data. The used *secret key* will be encrypted with the *destinations public key* and is then appended to the data. Furthermore, a third-party app should inform opptain if it has encrypted both elements itself, to not do it twice. This could be done by an extra attribute in *PayloadInfo*, e.g. *isEncryptedByApp*. The next step is to sign the bundle. Again, not the whole bunde can be signed because *NetworkPath* and *AdditionalInformation* are constantly changed as the bundle disseminates through the network. Besides that, the signing is done after PKCS#1 [JMKR16]. In short terms: first the bundle will be *digested* with a hash function, next the resulting digest will be signed with the *sources private key* and then the signed digest as well as the sources public key are appended to the bundle. This enhancement will stop bundle manipulations and guarantee *confidentiality*, *authenticity* that the origin is the real sender and *integrity* for all attributes, except *NetworkPath* and *AdditionalInformation*.
A positive side effect of this approach is, that *certain malicious* bundles can be dropped by benign nodes. Either they are not signed or the integrity check fails. The latter is possible since the origin of a bundle is also the public key that can be used to verify the appended digest.

Some minor enhancements that could be implemented: a threshold option in *opptain* for the amount and maximum size of bundles to be accepted, finding a solution to protect the information in *NetworkPath* and *AdditionalInformation* from forgery or manipulation. Considering, that AdditionalInformation can be bloated to over 1 megabyte so that an Intent delivery of the bundle would fail, the attribute might be changed to a path, like PayloadFile.

4.4.1 Private Data Storage

Since data, that would exceed the maximum size of an Intent, must be saved on the external file system to be shared with opptain, sensitive data could get exposed if it is not encrypted beforehand. To solve this problem a *FileProvider* [Goo17e] seems reasonable. The FileProvider allows apps to securely share selected files from their private storage; also works for external files. An implementation of this solution could be integrated into the opptain API to make it easy to use for developers.

4.5 Binder Attack Mitigation

A solution to decrease the chance of an open Binder attack surface, is to use lint with metadata tags on RPC interface elements to see potential parameter violations during testing. Additionally it is inevitable to *always perform sanity checks* on both sites of the IPC but especially on the server side of the Binder it is indispensable. Because even when the client does non checking the server can still fend off malicious inputs. [FS16] proposed an intrusion diagnostic that blocks transactions on receive when they are similar to previously encountered, dangerous transactions. Therefore sender and information of the incoming transaction, that caused a fail, are logged along with a signature of the transaction. The sender is retrieved by calling *Binder.getCallingUid()* and querying the *PackageManager* with the UID to get the package name. The information of the transaction are recorded during de-serialization if an unexpected exception occurs. The info contains parcel meta-data, the transaction signature and the parameter that failed to parse. New incoming transactions matching this signature can then be blocked if the user so wishes.

4.6 HotSpot Spoofing Mitigation

To thwart *HotSpoot Spoofing*, the client should check directly if the fingerprint from the SSID and the fingerprint calculated from the received public key match. The connection must be terminated if the spoofing is recognized. In the current implementation of the handshake, the adversary will have collected the public key of the victim. To stop this, two extra steps

in the protocol are required. Specifically, instead of starting with *initiateChallengeBob*, a new handshake start *requestVerificationKeyBob* from Alice to Bob, which Bob answers with *sendVerificationKeyAlice* should be added.

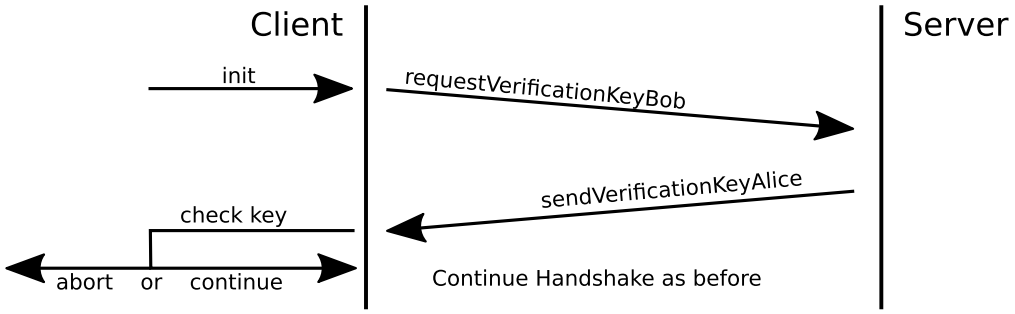


Figure 4.1: Handshake extension proposal

The attack in which a spoofing HotSpot uses a strong Wi-Fi device to lock peers and block their communication can be mitigated. Defining a threshold for the maximum amount of failed handshakes because of authentication failures is the first step. If the threshold is reached, a peer should stop trying to connect to HotSpots for some time. Another solution is to detect and avoid non Android devices that emulate opptain. Ways to achieve this are *DHCP fingerprinting* [Raj17] and *network scanner applications* (e.g. nmap [Gor17]). Both methods are approximations and not absolutely correct and a skilled attacker will be able to masquerade himself as Android device.

4.7 White- and Blacklisting

The opptain networking application offers *white- and blacklisting functionality*. As demonstrated in Section 3.3.5 blacklisting is not very effective in OppNets with nodes being able to generate their own DevideId on the fly. If a malicious node notices that it is blocked, it just generates a new Id. Trust and reputation systems are a better solution to block nodes in this situation. Nodes with a low rating will be simply ignored.
Whitelisting is more promising. Notably in scenarios with many nodes in vicinity, from which several are whitelisted, a node can chose to be picky and only communicate with white-listed devices without experiencing a high decrease in performance.

4.8 Fake Meeting Time Mitigation

Fake Meeting Time is easily fended. The Server just has to check if a *ConnectionId* was already used before and if a database update changes multiple connection entries. If a previous *ConnectionId* is received, it must be replaced with a new unused one. Furthermore, different following actions are possible, depending on whether the client is also the same as previously or a new client accidentally chose a used key. If it is the old one, the server might want to drop the connection and give the client a bad rating. In the case of the new client, the server may tell the client to chose a new *ConnectionId* before continuing.

Miscellaneous

Android apps supporting a minimum *API of 21 will be available for 71,5% of all Android devices* on the market [Goo17d]. Considering the former and the security threats in older Android versions that got fixed in *Android Lollipop*, a suggestion to raise the minimum API version of opptain to API level 21 seems reasonable. Beginning with Android 5.0 (API level 21), the system throws an exception if a call to *bindService()* is made with an implicit Intent. This prevents adversaries to hijack service sessions for opptain that were requested via implicit intents by incautious developers. Moreover, multiple apps cannot define the same permission anymore unless they are signed by the same signature. This prevents attackers that exploit the mechanism that the first requester decides the characteristic of the permission for all instances. Beforehand, the system would ignore the values from others and did not even notice about it. Last but not least, the functionality to get information about other app tasks was removed.
For the case that a user has to reset his device or that he was tricked to uninstall opptain, there should be an option to backup the public-private-key pair in a safe manner. Of course, it should be possible to load the backup to recover the old identity.

Chapter Conclusion

In this chapter solutions and mitigation mechanisms for attacks from the previous Chapter 3 were researched in literature or found after an analysis of the problem. The proposed solutions were mainly discussed theoretically in this thesis. Like for the attacks, testing of them is a task for future work.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The objective of this thesis was to research security threats against Android-based Opportunistic Networks and to find possible solutions. A special focus while working on the thesis was on the opptain networking application as a representative example. The course of action to achieve the task was to first explain and characterize threats found in corresponding literature as well as to find threats by analyzing the opptain source code and behavior. This was followed by a discussion of possible solutions and mitigation techniques.

The attacks and threats in the given scenario can be partitioned into attacks on the Android device and attacks against routing and communication functionality of Opportunistic Networks. The former mostly violate confidentiality, integrity and authenticity of messages send via the network while the latter mostly try to disrupt the functionality of the network by spreading forged meta information and by not forwarding data in the expected way. Short examples for both categories are adversaries listening to insecure communication channels on Android and malicious network nodes that deliberately drop packets they receive, called Blackholes.

The regarded network application opptain was found to be insecure in several points due to either flaws in the current implementation or in general through its behavior. The most severe: it relies on third-party apps to tell the truth about their identity; packets are not signed, leaving them vulnerable to manipulation; all encountered nodes are treated equally, making

it easy for an adversary to launch an attack. To solve the first, a short proof-of-work for an access control system was developed, that authenticates the real app name. For the second, a possible implementation approach for the signing process was described. And for the last, a combination of promising trust and reputation systems, that were found in literature, was proposed.

5.2 Future Work

While working on the thesis most threats and proposed mitigation mechanisms were only discussed theoretically because discussing them practically would have gone beyond the scope of this thesis. In general, the implementation and testing of them are options for possible future work.

First of all the access control system and packet signing can be implemented into the opptain network application. Moreover, all the other threats found during this thesis should be tested practically to find out their exact impact. Based on that, extensions to solve respectively mitigate the threats accordingly should be developed.

Furthermore a design to implement a module for the trust and reputation system into opptain should be discussed. Once this is done, the implementation and testing of parameters can be conducted.

Another point to test are the Intent and Bundle Threshold. Specifically, values that guarantee a high success rate to block spam and still offer efficient operation of the program need to be investigated. Conditions for adapting the values are another point of interest. Also a comparison of having a single threshold versus having multiple for different categories could be evaluated to find the better approach of both.

Bibliography

[BCJP07] C. Boldrini, M. Conti, J. Jacopini, and A. Passarella. Hibop: a history based routing protocol for opportunistic networks. In *2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–12, June 2007.

[DDSW11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[FHE+12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. Technical Report UCB/EECS-2012-26, EECS Department, University of California, Berkeley, Feb 2012.

[FS16] Huan Feng and Kang G. Shin. Understanding and defending the binder attack surface in android. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 398–409, New York, NY, USA, 2016. ACM.

[Gam88] Diego Gambetta. Can we trust trust? In Diego Gambetta, editor, *Trust: Making and Breaking Cooperative Relations*, pages 213–237. Blackwell, 1988.

[Goo17a] Google Inc. Application Fundamentals | Android Developers. `https://developer.android.com/guide/components/fundamentals.html`, June 2017. Last checked: 30.06.2017.

[Goo17b] Google Inc. Application security | Android Open Source Project. `https://`

source.android.com/security/overview/app-security, June 2017. Last checked: 30.06.2017.

[Goo17c] Google Inc. Application Signing | Android Open Source Project. https://source.android.com/security/apksigning/, June 2017. Last checked: 30.06.2017.

[Goo17d] Google Inc. Dashboards | Android Developers. https://developer.android.com/about/dashboards/index.html, June 2017. Last checked: 30.06.2017.

[Goo17e] Google Inc. FileProvider | Android Developers. https://developer.android.com/reference/android/support/v4/content/FileProvider.html, June 2017. Last checked: 30.06.2017.

[Goo17f] Google Inc. Intent | Android Developers. https://developer.android.com/reference/android/content/Intent.html, June 2017. Last checked: 30.06.2017.

[Goo17g] Google Inc. Platform Architecture | Android Developers. https://developer.android.com/guide/platform/index.html, June 2017. Last checked: 30.06.2017.

[Goo17h] Google Inc. System and kernel security | Android Open Source Project. https://source.android.com/security/overview/kernel-security, June 2017. Last checked: 30.06.2017.

[Goo17i] Google Inc. Tasks and Back Stack | Android Developers. https://developer.android.com/guide/components/activities/tasks-and-back-stack.html, June 2017. Last checked: 30.06.2017.

[Goo17j] Google Inc. TransactionTooLargeException | Android Developers. https://developer.android.com/reference/android/os/TransactionTooLargeException.html, June 2017. Last checked: 30.06.2017.

[Gor17] Gordon Lyon. Chapter 8. Remote OS Detection. `https://nmap.org/book/osdetect.html`, June 2017. Last checked: 30.06.2017.

[ibo17] ibotpeaches. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. `https://ibotpeaches.github.io/Apktool/`, June 2017. Last checked: 30.06.2017.

[IG17] Andre Ippisch and Kalman Graffi. Infrastructure Mode Based Opportunistic Networks on Android Devices. In *Proceedings of the IEEE International Conference on Advanced Information Networking and Applications*, pages 1–8, March 2017.

[Ipp15] Andre Ippisch. A fully distributed Multilayer Framework for Opportunistic Networks as an Android Application. Master's thesis, Department of Computer Science, Heinrich Heine University Düsseldorf, March 2015.

[JES16] Ryan Johnson, Mohamed Elsabagh, and Angelos Stavrou. *Why Software DoS Is Hard to Fix: Denying Access in Embedded Android Platforms*, pages 193–211. Springer International Publishing, Cham, 2016.

[JESS15] R. Johnson, M. Elsabagh, A. Stavrou, and V. Sritapan. Targeted dos on android: how to disable android in 10 seconds or less. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 136–143, Oct 2015.

[JMKR16] Jakob Jonsson, Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. Pkcs# 1: Rsa cryptography specifications version 2.2. 2016.

[KCHW12] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 69–80, New York, NY, USA, 2012. ACM.

[KM14] Shafiullah Khan and Jaime Lloret Mauri. *Security for Multihop Wireless Networks*. Crc Press, 2014.

[LD13] Na Li and Sajal K. Das. A trust-based framework for data forwarding in opportunistic networks. *Ad Hoc Networks*, 11(4):1497 – 1509, 2013. 1. System and

Theoretical Issues in Designing and Implementing Scalable and Sustainable Wireless Sensor Networks 2. Wireless Communications and Networking in Challenged Environments.

[LDS04] Anders Lindgren, Avri Doria, and Olov Schelen. Probabilistic routing in intermittently connected networks. *Service assurance with partial and intermittent resources*, pages 239–254, 2004.

[Les16] Jannik Lessenich. Routing protocols for Android based opportunistic networks. Bachelor's thesis, June 2016.

[LI04] Jinshan Liu and Valérie Issarny. Enhanced Reputation Mechanism for Mobile Ad Hoc Networks. In *Second International Conference on Trust Management : iTrust 2004*, pages 48–62, Oxford, United Kingdom, 2004.

[LWS09] F. Li, J. Wu, and A. Srinivasan. Thwarting blackhole attacks in disruption-tolerant networks using encounter tickets. In *IEEE INFOCOM 2009*, pages 2428–2436, April 2009.

[M. 17] M. Zinoune. Why is Android built on Linux Kernel? | Unixmen. `http://www.unixmen.com/why-is-android-built-on-linux-kernel/`, June 2017. Last checked: 30.06.2017.

[RAA+14] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. A systematic security evaluation of android's multi-user framework. *CoRR*, abs/1410.7752, 2014.

[Raj17] Rajesh K. DHCP Fingerprinting: Identify Device type, vendor and OS. `http://www.excitingip.com/4513/dhcp-fingerprinting-identify-device-type-vendor-and-os`, June 2017. Last checked: 30.06.2017.

[RZX+15] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, Washington, D.C., 2015. USENIX Association.

[RZZL14] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in android applications. *Int. J. Secur. Netw.*, 9(1):40–55, February 2014.

[Sco17] Scott Alexander-Brown. Android Security: Adding Tampering Detection to Your App. `https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app`, June 2017. Last checked: 30.06.2017.

[SE07] Dagmara Spiewak and Thomas Engel. *Applying Trust in Mobile and Wireless Networks*, pages 39–66. Springer US, Boston, MA, 2007.

[SPR05] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: An efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-tolerant Networking*, WDTN '05, pages 252–259, New York, NY, USA, 2005. ACM.

[SXB07] Xu Kevin Su, Yang Xiao, and Rajendra V. Boppana. *Secure Routing in AD HOC and Sensor Networks*, pages 381–402. Springer US, Boston, MA, 2007.

[TL09] Sacha Trifunovic and Franck Legendre. Trust in opportunistic networks. *Computer Engineering and Networks Laboratory*, pages 1–12, 2009.

[TLA10] S. Trifunovic, F. Legendre, and C. Anastasiades. Social trust in opportunistic networks. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–6, March 2010.

[WBDJ16] L. Wu, B. Brandt, X. Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63, Oct 2016.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 30. Juni 2017 Martin Nowak

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Testsoftware TestApp und ABE sowie die modifizierte opptain-Version
- die Websites der verwendeten Internetquellen