TU WIEN Informatics

Cruel Intentions

Enhancing Androids Intent Firewall

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Thomas Michael Klepp, Bsc
Matrikelnummer 0626890

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr. techn. Georg Merzdovnik

Wien, 15. Oktober 2020

Thomas Michael Klepp	Edgar Weippl

TU WIEN Informatics

Cruel Intentions

Enhancing Androids Intent Firewall

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Thomas Michael Klepp, Bsc
Registration Number 0626890

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr. techn. Georg Merzdovnik

Vienna, 15th October, 2020

Thomas Michael Klepp	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Thomas Michael Klepp, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Oktober 2020

Thomas Michael Klepp

ABSTRACT ENGLISH

The inter process communication *(IPC)* functionality of the Android platform has gained the attention of security researchers since the operating system's initial release. Because the *IPC* mechanism employs a publish-subscribe pattern by which applications choose the type of intent messages that they are prepared to receive, the system is vulnerable to malicious applications. Unsecured messages sent system wide may leak sensitive data when intercepted by unintended recipients, and unguarded exposed application components may be maliciously targeted to inject data to trigger unwanted behavior. To mitigate these dangers, the operating system received a mandatory access control system named *Intent Firewall (IFW)* in version 4.4, which allows monitoring and blocking *IPC* traffic based on user-defined rules. Because this system has limited efficiency due to its coarse filter granularity, difficult usability and lack of automatic threat detection, an upgraded version was needed. After reviewing static and dynamic research approaches as well as policy-based security tools to identify and regulate malicious intent traffic, the *Enhanced intent firewall (EFW)* was created to remedy shortcomings in the design of the original *IFW* implementation. Using this system, a set of both malicious and benign applications was analyzed to characterize dangerous intent traffic and subsequently counteract it through user-created policies. Furthermore, a detection module is presented to show the system's capability to analyze and monitor *IPC* communication in real time to detect and automatically block malicious behavior. Finally, the effects of the tool on the operating system's runtime performance are measured to demonstrate this approach's feasibility.

ABSTRACT DEUTSCH

Das Interprozess-Kommunikationssystem welches vom Android Betriebssystem zur Kommunikation zwischen verschiedenen Prozessen genutzt wird, hat seit der ersten Veröffentlichung der Plattform das Interesse der Sicherheitsforschung erregt. Durch den Einsatz des publish-subscribe Prinzips können Applikationen selbst definieren welche Nachrichten sie empfangen können. Allerdings ist dieses Konzept anfällig für Angriffe in welchen bösartige Applikationen ungesicherte Nachrichten abhören und Daten in ungeschützte Komponenten injizieren können um ungewünschtes Verhalten auszulösen. Um derartiges Verhalten zu unterbinden wurde dem Betriebssystem in Version 4.4 die *Intent Firewall (IFW)* hinzugefügt welche es Benutzern erlaubt die Interprozess-Kommunikation über Filterregeln zu regulieren. Dieses System zeigt allerdings mehrere Schwachstellen welche die Einsatzfähigkeit stark einschränkt. Sowohl die Bedienung der Firewall selbst als auch die Filtermöglichkeiten sind begrenzt, weiters fehlt dem System die Fähigkeit autonom Angriffe zu erkennen und zu blockieren. Nach der Analyse von statischen und dynamischen Forschungsansätzen sowie von regelbasierten Filtersystemen, präsentiert diese Arbeit mit der *Enhanced intent firewall (EFW)*, eine erweiterte Version der Firewall welche die Schwachstellen der ursprünglichen auszugleichen sucht. Nach dem sowohl die erweiterten Filtermöglichkeiten und die automatische Angriffserkennung des erweiterten System diskutiert wurden, werden die Auswirkungen auf die Performance des Betriebssystems beleuchtet um die praktische Einsatzfähigkeit des Ansatzes zu demonstrieren.

Contents

CHAPTER 1

Introduction

Due to its ubiquitous usage, the Android operating system is targeted by increasingly sophisticated malware, with one attack vector being the system's inter process communication *(IPC)* functionality. Although it was designed to constitute the sole communication channel between isolated processes in order to prevent their interference with each other's resources, the *IPC* system can be abused to attack unsecured applications. The intent messages used for IPC, both to launch application components and to transport data, employs publish-subscribe by which each component defines the type of intents it is prepared to receive. As discussed in Section 3.10, this mechanic allows malicious applications to perform attacks, such as intercepting unsecured messages to eavesdrop on sensitive data or injecting intents into unsecured application components to trigger unwanted behavior. The operating system's mandatory access control functionality called *Intent Firewall (IFW)* addresses this problem by allowing manually configuring filters to prevent matching intent messages from being sent or received by the defined application. Analyzing this system, however, revealed shortcomings restricting its efficiency. Although configuring the firewall rules does not facilitate all necessary filter combinations, their deployment and access to the monitoring data created when firewall rules are triggered proved inconvenient for regular use. Furthermore, the system offers no functionality of actively scanning and reacting to threats by creating rules on its own. To remedy these shortcomings, this work presents an upgraded version of the intent firewall, which was designed after evaluating the *IPC* traffic of benign applications from the *Google Playstore* and samples from a set of known malware. Based on the findings, the *Enhanced intent firewall (EFW)* allows creating finer-grained rules for filtering while offering a convenient user interface, furthermore, its extendable architecture allows adding detection modules to scan for irregularities in the *IPC* traffic which are not covered by active firewall rules. Such a module is used to address an attack performed by several previously analyzed malware samples. To demonstrate this approach's feasibility, this study measures the impact on the operating system's performance via firewall rule evaluation as well as

the scanning and subsequent rule adaptation of the detection module. After reviewing different research approaches to intent-related attacks in *Chapter 2*, *Chapter 3* discusses the structure of the operating system's *IPC* functionality and received attacks. *Chapter 4* covers the implementation details of the *EFW* application and the data collection process of the sample's intent traffic. *Chapter 5* presents the data analysis and shows how irregular *IPC* traffic can be addressed by firewall rules and an autonomous detection extension. In addition, the implementation's efficiency is demonstrated by calculating the effect of the approach on the system's performance. Finally, *Chapter 6* outlines possible future enhancements of the system.

CHAPTER 2

State of the art

Intent-based vulnerabilities and their exploitation have been researched from several perspectives. The foremost employed technique regards static and dynamic program analysis of both single and groups of Android applications. Following the evaluation of [1], who assessed intent-related security research over a period of six years, static analysis has received the most attention and was being used in 73% of the reviewed works, followed by a dynamic approach (17%) and hybrid methods employing static and dynamic analysis techniques (10%). The research covers vulnerability detection, which aims to identify benign applications which are susceptible to attacks, and malicious behavior analysis to detect malignant applications. The former represents the objective in 12% of the research, the latter regards 82% the remaining 6% cover both research questions. Lastly, mandatory access control tools are discussed, which monitor and regulate Android *IPC* traffic in real time.

2.1 Static analysis

Static program analysis evaluates an application based on artifacts such as source or byte code as well as meta information such as the program's configuration files. The difference from dynamic analysis is that the respective application is not executed but rather solely evaluated using the forementioned resources. In the case of intent-related security research, static analysis regards two widespread approaches: Firstly, taint tracking requires an application's source or byte code. This approach follows the invocation of program statements between points of interest in the programs code, where the origin of a flow is called the source and the destination labeled sink. Notable sources and sinks include sensitive system API methods and resources as well as entry points of Android components, because the goal in security scoped taint tracking is to determine if a certain sensitive point within a component's scope is reachable from outside. The research discussed below employs graph-based data structures to formulate these data

flows, where functions are represented by nodes while the edges connecting them represent the invocation of this function. Although static taint analysis provides a comprehensive picture of an application's behavior, the approach poses several challenges such as regarding precision. Because Android concerns an event and component-driven system, data flows must be tracked, through every branch of the application's code and through system calls, the component life-cycle as well as through code from employed third-party libraries. The calculation of all possible data flows and values thus demands unfeasible time and processing power, while reducing the tracking accuracy also reduces analytical precision. Analysis is also hindered by obfuscation via the encryption of class, function and variable names as well as data values such as URLs and commands to meaningless strings. The code dynamically loaded during execution and native payload such as *C++* code are not covered by static taint analysis. To circumnavigate these issues, the second category of approaches using static analysis employs more heuristic actions which depend on the meta data of applications such as the manifest file to assess the application's behavior. Techniques such as machine learning or the adaptation of the *IPC* system are used to identify intent-related security issues. Similar to the taint tracking approach, dynamic loaded code can not be analyzed using static means.

2.1.1 Vulnerability analysis

Vulnerability analysis aims to identify applications which are vulnerable to intent-based attacks. A vulnerability can be caused by the insecure implementation of a single component which has been unintentionally exported or which uses unchecked input data. This vulnerability can stem from the interaction of multiple components.

Using static taint analysis to identify intent-based vulnerabilities in real-world applications, *ComDroid* [2] disassembles apk package files to analyze the resulting bytecode files. The analysis tracks the state of intents, intent filters and application components to identify whether a component can be reached by a certain intent causing insecure *ICC* communication patterns. *ComDroid* then issues warnings when, for example, implicit intents are sent with weak or no permissions or components are automatically exported through intent filters without being secured with permissions.
Evaluating the tool using 100 applications identified 1,414 exposed attack surfaces, which would allow spoofing, privilege escalation and data leakage attacks. Although the findings may include false negatives since the analysis does not distinguish between different paths in conditional statements, intents are treated as implicit when made implicit in one branch and explicit in another. Actual attacks on found vulnerabilities are not verified by this approach, and changes to Android's *ICC* system are suggested to close some attack surfaces.
Chex [3] aims to detect highjackable data flows in applications by utilizing app splitting, whereby an application's code is divided into several parts, each accessible via a single entry point. Predefined policies are used to analyze the data flows from all splits to identify exploitable weaknesses in the respective component. Evaluating 5,486 sample

applications uncovered 254 potential vulnerabilities, and due to its deeper analysis approach, *Chex* is less susceptible to falsely labeling exported components as exploitable. Although this method has limited efficiency due to missing knowledge regarding the order in which the application's components are invoked and the extent to which vulnerable data flows are sanitized by complex logic.
Using a similar approach as *ComDroid* [2], *Epicc* [4] interprets inter-component communication as an inter-procedural distributive environment dataflow analysis problem, which can be solved efficiently [5]. After creating a call graph to model the communication between components, *Epicc* solves the IDE problem using their analysis tool based on *Soot* [6], a framework intended to optimize java bytecode. Evaluating 1,200 applications showed a 32% reduction of false positives compared to *ComDroid* due to its capability of differentiating between multiple branches of code statements. In addition, *Epicc* can check the feasibility of attacks for found vulnerabilities in applications while *ComDroid* merely seeks potential vulnerabilities.
By comprehensively inferring the intent values and the correlations between them, *IC3* [7] targets a more precise analysis of *ICC* dataflow than previous approaches such as *Apposcopy* [8] or *Epicc* [4]. String-based values are calculated more precisely as *IC3* shows an 84% malicious data flow detection accuracy for inferring intent values, compared to 68% with *Epiccs*. The approach is inspired by *FlowDroid* [9] and builds a similar call graph, which is then fed to a *Soot*-based [6] data flow solver to track *IPC* data flows. Compared to *FlowDroid*, *IC3* finds 78% fewer possible targets for *ICC* data flows.
Addressing a rarely discussed attack vector, *PIAnalyzer* [10] aims to detect component vulnerabilities produced by the unsafe usage of pending intents. This approach examines pending intents and respective used base intents by modeling all component's usages of such intents into a call graph, which is then manually assessed to decide whether the vulnerable code is executed. Evaluating the approach on 1,000 samples showed 1,358 unsafe usages of pending intents such as the use of implicit intent to create the pending one. In 70 cases, an unprivileged malicious application could perform critical operations when obtaining an unsafe pending intent. Limitations of this approach include behavior which depends on dynamic input, and if the tool can not decide whether an intent would be explicit an runtime, the intent is treated as implicit, which leads to false positives.

To identify which inter-application communication could be replaced by intra-application communication, *IntraComDroid* [11] proposes changes to the heuristics which the Android system applies to determine the destination components of intents. These changes allow automatically detecting and patching vulnerabilities such as unintentional exposed components and intents sent unnecessarily global. *ComDroid* [2] was used to analyze 969 applications, to measure which security vulnerabilities were fixed by the approach. The evaluation shows that the new heuristics would fix 45% of unintentional exposed components and 18.5% of exposed intents, although the approach is limited to vulnerabilities which are detectable by *ComDroid* as well as to the kind of applications tested.
[12] changes the behavior of the Android *ICC* system, as suggested by [2]. The functionality which parses the manifest of an application during installation has been adapted

so that components automatically exported by an intent filter remain inaccessible from outside the application. In addition, implicit intents would be primarily delivered within the application before considering system-wide receivers to avoid information leakage. Evaluating these changes on 497 applications showed that 69% contained automatically exported components. Examining 20 of them manually suggested that 71% could be set to private and that 96% of implicit intents would be prevented from leaving the application's context. Although the evaluation also showed that these security restrictions would disrupt some intended app functionality, this could be partially remedied by including permission and namespace awareness into the security enhancements.
To reduce the false positives of previous works, *ApSet* [13] facilitates information from the Android class documentation to automatically generate test cases from a given apk, which describes the application's components and behavior regarding the use of implicit or explicit single intents. Creating test cases for 70 sample applications and matching them with predefined intent-related vulnerability patterns showed that 62 contained at least one vulnerable component. This approach improves the work of *ComDroid* [2] by evaluating the described vulnerabilities using blackbox tests; however this approach considers neither the component type broadcast receiver nor attack patterns involving several different intent actions.
To enable developers to adopt best practice recommendations regarding intent-related security, [14] developed a plugin for the Android Studio IDE, which analyzes the code for common intent-based vulnerabilities. The vulnerability detector was created to check the source code during implementation to detect the OWASP Top 10 mobile security risks [15]. The tool warns the developer when a component is automatically exported by defining an intent filter or when globally sent intents are used for internal communication. For each found vulnerability, the plugin offers an alternative implementation which follows the best practice guidelines.
To improve the detection precision of previous works, *VanDroid* [16] uses a model-driven approach, whereby an apk is analyzed to formulate security aspects of the Android application into a formal model, which is then matched with patterns of predefined security vulnerabilities such as data leakage and injection attacks. The approach was analyzed using 130 applications and found 501 vulnerabilities. VanDroid showed a detection precision of 100the with the tools *IccTA* [17], *FlowDroid* [9] and *Amandroid* [18], thus outperforming all compared approaches.

2.1.2 Malicious behavior analysis

The goal of malicious behavior analysis is to identify attacking applications which aim to exploit vulnerabilities in the operating system or other applications. To detect such behavior, taint analysis has been employed to analyze vulnerabilities and alternative approaches. In addition to finding a single malicious application, collusion attacks are discussed, where multiple malignant applications cooperate to perform an attack.

To detect malicious data flows, *FlowDroid* [9] parses the bytecode of applications to find suitable sources and sinks for potential data flows, which are then processed by *SuSi* [19],

a tool which automatically detects data sources and sinks in the Android system. After creating a call graph describing the invocation flow of each components life cycle methods, *Soot* [6] is used to track the flow from sources to sinks. *FlowDroid* showed a 86% detection rate for malicious data flows when testing 500 samples taken from Google Playstore and 1,000 malicious ones. This approach improves on the similar work of *Chex* [3] through a higher detection precision due to considering callback and component life cycle methods and through greater sensitivity to the context and objects involved in the data flow. In addition to inherent limitations of static analysis, the approach does not consider multi-threading but instead treats all operations as occurring sequentially.
Similar to *FlowDroid's* approach [9], *DroidSafe* [20] also employs *SuSi* [19] to create a list of sources and sinks for their analysis; however, *DroidSafe* additionally uses manual identification to ensure tracking all sources and sinks necessary for the analysis, since some are not discovered automatically. When performing static flow analysis between these endpoints, *DroidSafe* managed a detection rate of 93% of malicious data flows when evaluated on 24 applications, including 13 false positive findings. This approach thus surpasses *FlowDroid* regarding detection precision. Although this approach's accuracy depends on a complete list of sources and sinks, endpoints missed or not considered as sensitive by *DroidSafe*, can-not be detected, and furthermore no implicit data flows are mined.
AmanDroid [18] generates a flow graph to model interactions between components and API methods of the Android system, representing an abstraction of the applications behavior and allowing a more comprehensive analysis of data flows over *ICC* between components compared to previous works. Testing 753 benign and 100 malicious applications revealed several data leakage and injection attacks between the sample applications. *AmanDroid* improves on *FlowDroid* by tracking *ICC* calls over multiple components and on *Epicc* [4] by leveraging the result values of inter-component analysis for *ICC* calls. The limitations of *AmanDroid* include cases where security exceptions are thrown in a sampled application leading to false negatives and methods using reflection and concurrent running threads are not tracked.
Building on *FlowDroid* [9] and *Epicc* [4], the approach of *DidFail* [21] combines both. After analyzing the manifest file of an application, FlowDroid [9] is used to track data flows inside components while *Epicc* [4] identifies and tracks properties of sent intents. After individually analyzing each sample application, the collected data are used to find data flows from sources and sinks between different components. This approach has proven feasibility for both benchmark tests and malicious applications created for evaluation. The approach also inherits the limitations of *FlowDroid* and *Epicc*, and *DidFail* only examines intents related to activity components.
Similar to *DidFail* [21], *IccTA* [17] can detect and track malicious data flows over multiple components. After generating possible data sources and sinks with *SuSi* [19] and building a flow graph between them, multiple sample applications are merged, with *Epicc* generating links between components and *FlowDroid* facilitating intra-component taint analysis. When evaluating the approach using 1,260 malicious applications, *IccTA* showed a 96.6% precision rate for detecting malicious data flows, which improves on both

employed approaches' accuracy while expanding the detection scope of data flows to multiple components; however, the limitations of both approaches are inherited as well. Using the approach of signature-based malware detectors such as anti-virus software, *Apposcopy* [8] defines the semantic characteristics of Android malware and aims to identify these signatures in sample applications. Similar to *Epicc*, [4] a call graph models the data flow between components used in the taint analysis, which relies on hand-written models of 1,100 Android system classes. Evaluating 1,027 malicious applications showed a 90% accuracy in detecting malware, although the precision varies based on the type of malware. The signature-based approach includes an inherited vulnerability to obfuscation techniques such as dynamic code loading and reflection; furthermore, the time-consuming detection technique does not allow the instant discovery of malware.

To detect potential data leakage and injection vulnerabilities, *PCLeaks* [22] employs *FlowDroid* [9] to create a flow graph to model the relation between component's sources and sinks. The collection of sources and sinks are generated by *SuSi* [19] since this approach builds on *FlowDroid* [9]; however *PCLeaks* treats all entries as sources and exits to accommodate cases in where a source or sink method call is overwritten by a custom implementation. To verify potential leaks, an automatically generated Android application attempts to exploit the vulnerable component. Evaluating 2,000 applications revealed 290 findings, 75% of which were confirmed as true positives, although not all *ICC*-related system methods are considered in the analysis.

To reduce the challenge of inter-application communication analysis, *ApkCombiner* [23] utilizes the fact that the Android OS uses the same functionality to facilitate inter-application and intra-application communication. After merging one malicious and one benign application into a single apk, *ApkCombiner* can employ *IccTA* [17] to detect inter-app vulnerabilities as encapsulated within a single application. Evaluating 3,000 samples showed that *ApkCombiner* can detect inter-application communication; however this approach is limited by its scalability since combining more than two applications increases the chance that combining resources and dependencies of the applications leads to a non-executable package.

Attempting to detect permission leakage between applications, [24] aims to improve on *Covert* [25] and *ApSet* [13] by comprehensively handling asynchronous calls and Android life cycle calls. After building a graph for *IPC* calls and determining the target components of the intents, this approach matches all found execution paths with predefined vulnerability patterns. *Soot* [6] is employed to handle calls over Android's life-cycle methods, while *FlowDroid* [9] is used to tracks call paths from sources to sinks. Evaluating 550 applications showed a true positive rate of 68%, although the system failed to analyze 15.6% of the applications due to the large computational overhead of the taint analysis. While previous works do not analyze intents which are obtaining and reused by a component other than the sending one, *ICCA* [26] aims to detect such cases. After analyzing the bytecode of an application and using Soot [6], *IC3* [7] is used to infer values and methods of the *ICC* calls, while considering the reused intents. Modeling this data on a call graph allows tracking the components' intent communication. Through this method, *ICCA* improved on *IC3* by detecting 73 revised intents in the tested application

set and by tracking 431 implicit intents, which *IC3* is not designed to handle.
To detect data leakage between applications, *SDLI* [27] computes a summary for each analyzed sample, including a list of tainted information sent through intents and of which intents an application can receive. These data are extracted from both manifest and bytecode files. Comparing the summaries of samples using a static analyzer detected applications leaking data outside of their contexts. *SDLI* tracks explicit and implicit intents and considers all types of Android components, and evaluating 47 applications from the Google Playstore showed that all but one included at least one case of information leakage. Using a similar approach as *FlowDroid* [9], [28] uses data from the application's bytecode and manifest file to create a call graph of the used *ICC*-related methods using *IC3* [7]. Based on this graph sources and sinks in different components are connected and taint analysis is applied to track the flow of the *ICC* calls to decide whether attacks can be performed on this path by one of the communicating components. Evaluating 60 applications, showed that this method can detect more types of vulnerabilities than previous works such as *Chex* [3].

Aiming to identify malicious applications by analyzing manifest files, [29] compiled a list of keywords from the manifests of 30 benign and 30 malicious applications. This analysis showed that manifest properties such as actions, intentfilter categories as well as used permissions and process names indicate application's maliciousness. Comparing these keywords to manifests of 130 known malicious and 235 benign applications allowed calculating a malignancy score for each sample, which showed a true positive rate of 91.4% for benign applications and 87.5% for malicious ones. The evaluation also showed that certain types of malware such as adware can not be detected by this approach since their manifest files were too similar to those of benign applications.
Aiming to discover combinations of characteristics associated which malware, *Drebin* [30] extracts features from application manifest files and used API calls from the disassembled source code and embeds them into a vector space. To efficiently detect these patterns, machine learning algorithms are used on the vector model to decide whether a sample should be considered to be malware. Using 10 virus scanner services to divide test samples into 123,453 benign and 5,560 malicious applications, *Drebin* showed a precision of 94% when detecting malware, with a false positive rate of 1%. While aiming to be resilient to obfuscation during analysis, the precision of *Drebin* is susceptible to pollution attacks, whereby malware showing benign features can poison the training dataset.
ICCDetector [31] similarly only considers how an application uses inter-component communication. Using *Eppic* [4] to extract ICC communication patterns from the apk files of 5,264 malicious and 12,026 benign application samples, *ICCDetector* uses this data to train a decision-making engine to detect malware with a precision of 97.4% and false positive rate of 0.67%. Thus, *ICCDetector* shows higher accuracy than *Drebin* [30] yet experiences the same limitations regarding the training data.
Combining the analysis of intents with that of used permissions, *AndroDialysis* [32] plans to achieve higher detection precision compared to systems considering only one of these aspects. Using extracted data from manifest files and source code to create a statistical

model, which is fed to a decision-making algorithm, allows labeling a sample application as benign or malicious. Evaluating 1,846 clean applications and 5560 infected ones taken from the *Drebin* dataset [30] showed 91% precision when only using intents as indicator of maliciousness, 93% when only using permissions and 95,5% when combining both aspects.
IntGet [33] aims to distinguish between malicious and benign applications by comparing the property values of the used intents. To obtain these values *IntGet* analyzes applications' manifest and bytecode, to create a list showing implicit and explicit intents as well as which and how many *IPC* method calls are used by the application. Performing these steps on 20 benign samples and 20 malicious ones from the *Drebin* dataset [30] allows defining characteristics which are typical of malware, such as certain intent actions and frequency concerning the use of *IPC*. This approach shows that infected applications exhibit different pattern in the usage of actions in intents; however the work discusses these results only for the component-type activity.

Aiming to improve previous approaches such as *Epicc* [4] which focused on analyzing single applications, *Covert* [25] intends to detect malicious behavior originating from the collusion of multiple applications. In an initial step, a formal model of each sample application and the configuration of the respective Android framework version used in the analysis are created using the *Soot* tool [6]. Afterwards, formal analysis is performed on the combined models of the applications and framework to find vulnerability patterns. This approach detected malicious behavior with a precision of 60% after testing on 200 samples, and it is vulnerable to false positives since only static analysis is employed.
Fuse [34] aims to detect app collusion in a specified set of applications by initially performing single-app analysis on each application in the set, followed by a second analysis which considers the relationship between the applications. For each app, *Fuse* generates an extended application manifest which contains the sources, sinks and data flows between them. Based on this data, a call graph is generated to model the data flow between the set's applications, and then taint analysis is performed to find data flows to reachable sinks. Although performing a more detailed single app analysis compared to *Chex* [3] and *Epicc* [4], *Fuse* showed a lower precision compared to *FlowDroid* [9].
Employing two different approaches to detect app collusion, with both relying on the analysis of used permissions and communication methods, [35] extracted data from the application's byte code and from the manifest file. The first approach uses this information to model the possible communication methods of application pairs and then examines the model using a set of rules to identify colluding applications. The second approach uses machine learning techniques to detect colluding pairs of apps. For testing purposes, 4 sets containing 11 colluding and 3 benign apps were created. While the rule-based method detected all malicious app sets with 8 false positives, the second one found only 2 app sets with 5 false positives. These approaches, however were not evaluated using actual malware, the success of the rule-based approach depends on the focus of the employed rules and both methods include scaling-issues, since the detection must be performed for all possible combinations of applications. *MrDroid* [36] uses an empirical approach to rank applications suspected of collusion. In an initial step, a call

graph is built to show pairs of apps sharing multiple *IPC* connections by parsing the application's byte code and manifest files, after which *IC3* [7] is employed to analyze the data to discern the sources and sinks of the *IPC* calls. The connections between apps' pairs are ranked using a scoring system, which analyzes whether the communication is uni or bi directional for example, to identify high-risk applications. This approach was tested on 500 samples and resulted in rating 6 application pairs as high risk and 169 as medium risk for colluding with each other. During the evaluation, *MrDroid* identified all 8 colluding app pairs in the sample set. This approach includes limitations in that intent attributes are considered for the inspection but no data flow analysis is performed, which might allow malicious applications to remain undetected, and only pairs of apps can be detected for collusion. To detect leaks of sensitive data across multiple apps, [37] creates a model containing information concerning the use of *ICC*-related classes such as intents and intent filters as well as the usage of methods which can access sensitive resources like the device id. By analyzing this data using a model checker, *ICC* paths which leak data from one application context to another are uncovered. This approach was evaluated on a set of 8 sample applications implemented for this purpose. While the initial tests checked tuples of applications for collusion showed promise, the aim is the ability to detect three or more applications performing collusion attacks. Furthermore, this approach claims to be able to identify leaks not detectable by its predecessors *IccTA* [17], *FlowDroid* [9] and *DroidSafe* [20]. Aiming to detect colluding applications, *DialDroid* [38] extracts permissions and intent filters from applications' manifest files and uses *IC3* [7] to determine which intents can pass the found intent filter. Similar to that used by *FlowDroid* [9], a call graph is generated to identify entry and exit points of intents in each applications' component. A database is compiled using this data to allow efficiently calculating potentially sensitive inter-application *IPC* paths. Evaluating 100,206 benign and 9,944 malware applications showed a higher accuracy compared to *Covert* [25] and *ApkCombiner* [23] in addition *DialDroid* improves on *IC3* by identifying 28% more intents and processing 33% more applications for which *IC3* failed.

2.2 Dynamic analysis

Unlike static analysis approaches, in dynamic analysis a program is executed to observe the behavior of the application during runtime. Although this analysis allows detecting malicious behavior which can-not be detected by a static approach, such as the origin of malicious functionality in native or dynamically loaded code, it also entails some challenges. Because malware aims to remain undetected, a goal of dynamic analysis is to create a controlled execution environment while providing all necessary input to trigger malicious or vulnerable behavior in applications. Some approaches aim to create suitable input values via fuzzing, whereby random data is employed; however the range and succession of possible inputs may reach an infeasibly large amount regarding creation and repeatability. In addition, a dynamic approach must not draw the monitored application's attention towards the analysis since malware might cease malicious behavior when recognizing an analysis attempt. Aside from providing input, a monitoring environment must also

provide or emulate resources which might be required by a malicious application, and thus some approaches employ machine learning techniques to classify vulnerable and malicious applications.

2.2.1 Vulnerability analysis

DRFuzzer [39] aims to identify vulnerabilities in real-world applications by targeting exposed components of applications with crafted intents to trigger exceptions. While the intents' action, category and data values are used unchanged, the key value pairs in the intents' extra payload are randomly fuzzed. The target components behavior towards such intents is compared to intents with extras of the expected type and value range as well as to intents with no extra payload values. Although the fuzzing showed a high success rate for causing application crashes, the underlying vulnerabilities were difficult to reproduce. To detect vulnerabilities with-out heavy system instrumentation such as that of *CopperDroid* [40], *IntentDroid's* [41] detection is executed on unchanged Android systems via debug breakpoints. This approach monitors a selected set of platform APIs, which it deems security related, to reveal execution paths when invoked. These paths are tested by predefined attack patterns, whereby intent fields are fuzzed according to preset conditions to reduce the effort from attempting unsuitable values. The approach detected 150 of 163 vulnerabilities in 80 applications, although the system's detection capabilities are limited to the restricted set of monitored API endpoints.

2.2.2 Malicious behavior analysis

By modifying the *IPC* binder library, *TaintDroid* [42] can dynamically track data flows by tagging information used in variables, files and method calls. These taint tags are tracked from their origins in trusted applications until they reach an untrusted application context, and if the tracked data leak through a sink such as the network, the system classifies this application as potentially malicious. Testing 30 applications showed 105 incidents of data leakage were flagged by *TaintDroid*, of which 68 were deemed as problematic. Evaluating the system showed a 14% performance overhead due to the adapted binder functionality. This approach has limited accuracy since it is unable to track implicit intents and it produces false positives when tracking configuration data such as the device's IMSI number.
CopperDroid [40] aims to aid tracking *IPC* calls through Android's life-cycle and binder functionality by employing virtualization of the Android image as well as the underlying *Dalvik* VM and *Linux* kernel in order to collect and track all system calls involved in the communication. During analysis, events based on the application's manifest are injected to trigger the execution of different parts of the application's code. Using the obtained data, the Android *IPC* traffic is afterwards reconstructed to classify system calls into six malicious behavior types. The system was tested on 2,900 malware samples and in 60% of the samples triggered malicious behavior which had not been previously identified.
Employing a machine learning approach to detecting malicious applications, [43] instrumented 3780 benign and known malware samples with API monitoring code and executed

them in a virtual device. Using the obtained data, features regarding the frequency of used system calls and consumed API interfaces by the samples were extracted and used to train a classifier, which was used to test 3,740 samples consisting of malicious and benign applications. Evaluating the approach showed a true positive detection rate of 96.82%, although the approach fails to provide the stimulation necessary to trigger malicious behavior of samples as well as being detectable during monitoring. [44] relies on machine learning techniques to identify malicious applications, and the created system executes sample applications in an virtual environment which integrates parts of *TaintDroid* [42] to facilitate data tracking. The approach injects events into the sample applications to trigger malicious behavior, which has been predefined by a set of intents which are deemed as malignant. For each of the defined intents, the system retains weather by the sample launched such an intent or not. These data are then used by machine leaning algorithms to classify the sample as malignant or benign. The system was trained with 15,000 benign and 15,000 malicious sample applications and was evaluated on 1,315 samples from each malware and benign datasets, which showed a high performance and a high accuracy in correctly identifying the applications as malicious. Not all types of intents were considered in the approach, and intents relating to service launches were omitted.
Aiming to increase the detection precision of approaches similar to [44], *Droidcat* [43] extracted 122 behavioral features from 135 malicious and 136 benign applications and determined hat 70 are characteristic of malicious behavior. The approach was evaluated using 70% each of 17,365 benign and 16,978 malicious sample applications to train the decision-making algorithm and afterwards classify the remaining 30%. The approach showed a 97.4% accuracy in correctly labeling the samples, although the system's efficiency depends on a balanced training set since machine learning techniques are employed. While *Droidcat* showed more stable and precise malware detection in compared to [44], the authors of [43] claimed that [44] is more precise when used for malware categorization.

2.3 Hybrid analysis

As discussed above, both static and dynamic analyses show benefits and drawbacks regarding the ability to detect vulnerable and malicious applications and regarding computational expenditure. Combining both types of analysis allows compensating for the shortcomings of the separate approaches; for instance, static code evaluation may enable limiting the range of promising input values for dynamic analysis. This hybrid approach, can increase the overall complexity of the process since two analysis steps must be performed.

Following a similar approach as *DRFuzzer* [39], *Intent Fuzzer* [45] uses extracted data from application manifest files to calculate the expected makeup of intents accepted by the application's components by using a modified version of *FlowDroid* [9]. Based on the outcome of this step, valid intents with randomized data payload, are generated to target exposed components. The applications are monitored regarding code coverage and error

handling, while receiving the intents to identify exploitable components. Fuzzing the intent data payload showed a 12,5% increase of executed code in the targeted components, compared to the usage of intents carrying no data while triggering unwanted behavior, thereby demonstrating this approach's feasibility.

[46] created an analysis framework to assess the extent of intent-based attacks on unexposed components which are not directly accessible by outside attackers. After examining manifest files and source code to identify exposed components and statements in them which utilize data received from intents, the data flow paths to these sinks are processed using a similar approach as *Epicc* [4] by interpreting the dataflow as *IDE* problem. After discovering a vulnerable data path, an intent's makeup is determined to induce malicious behavior into the unexposed component. When testing the approach on 64 samples from the *Google Playstore*, 29 exhibited vulnerable data paths, 26 of which were feasible using a concrete exploit.

During an initial static analysis phase, *RainDroid* [47] creates a vulnerability model based on extracted data from manifest files and the application's *IPC* communication patterns obtained from the app's bytecode. This model is then used during runtime by a analyzer based on *Covert* [25] to monitor the application's interactions for intent-based security problems. During evaluation, the approach detected an insecure intent's launch by dynamically loaded code, which deviated from the previously created communication model of the application.

To identify vulnerable data flow paths from unprivileged sources that could execute privileged operations in their destination component, [48] employs *FlowDroid* [9] to facilitate static taint analysis. During a second phase the identified *IPC* paths are examined using *TaintDroid* [42] to check whether a particular path is taken during runtime. From 329 analyzed applications, the static analysis phase flagged 53 apps as vulnerable, while 9 of the findings were confirmed as true positives during dynamic analysis.

[49] uses a similar approach as [45], but bases the static analysis on the sample application's smali instead of bytecode to mitigate the impact of obfuscation techniques on the analysis. After modeling a graph of *IPC* communication which describes paths from application components to sensitive system apis, the resulting data are used to create suitable intents to execute these paths to trigger the sensitive sinks. Evaluating the framework on 6,187 known malicious applications detected 79 previously unidentified cases of information leakage. Executing the intents requires the framework to enrich the sample apks with additional code, which may influence the result.

[50] extracts the names of exported unsecured components from an application's manifest file and targets them with forged intents to trigger unwanted behavior, and thus a keystore for intent communication is proposed, where the sender and receiver application must be linked by the user via a cryptographic key before being allowed to exchange intents signed using this key. Evaluating the approach showed promise while revealing issues regarding performance and application stability when receiving components fail to match the intents' key.

To supplement dynamic analysis systems such as *TaintDroid* [42], *IntelliDroid* [51] focuses

on generating targeted input values to trigger malicious behavior while *ICC-Inspect* [52] aims to model application functionality by visualizing the employed intent traffic. A call graph is created by analyzing the bytecode of an application and employing *IC3* [7], and during the applications execution a dynamic call graph is constructed which depends on the user's interaction. Combining the two analysis steps models an interactive visual representation of the *IPC* traffic, showing both the intent messages started and received as well as potential destination components of intents based on registered intent filters. While the runtime performance showed no overhead, the initial analysis ran 15 minutes on average per application when using static analysis and 3 minutes on average when omitting this step.
AndroShield [53] aims to create a more comprehensive vulnerability model compared to previous approaches by initially performing static analysis using *FlowDroid* [9], before a second dynamic analysis phase, when random fuzzed input values are used to trigger unwanted behavior in the sample applications. The collected data are used to model a vulnerability profile for each application, which ranks the samples' risk to defined vulnerabilities and their impact on the user. While finding security issues in developer's code, the *Xposed*-based [54] approach demonstrated its capability during the evaluation on 70 applications and its effective usage by end users.
NIVAnalyzer [51] focuses on detecting an attack vector named *next intent vulnerability*, whereby an intent targets a public component while carrying another intent aiming to exploit a private component of the receiving application. The vulnerability emerges when a component, such as an activity after confirming a successful login, allows redirection to another internal activity in an unsafe manner. To detect such cases, *NIVAnalyzer* initially uses static flow analysis to find occurrences of unsafe redirection in an application's smali code, which would allow this type of attack. Afterwards, suitable intents are created to confirm the access to internal components. The execution on 20,000 apps from the *Google Playstore* confirmed that 190 were susceptible to such attacks.
To build on the work of *NIVAnalyzer* [51], which did not consider all methods to transport intents which can be used for *next intent vulnerability* attacks, *NIVD* [55] compiles a list of all possible execution paths which can potentially lead to such attacks and subsequently tests their feasibility. Testing 100 applications showed 9 occurrences of such vulnerabilities, and this approach was 20% faster in detection than *NIVAnalyzer* [51], while claiming a precision of 100%.

2.4 Policy-based security tools

Policy-based security tools aim to extend the permission system of Android to monitor and manage the operations which an application is allowed to perform. In the scope of intent-based security, several approaches have created a solution which offers versatile and fine-grained policy definition while achieving a low performance overhead. The discussed tools follow two paths to achieve this task: Some approaches provide policies which are predefined by human experts or automatically derived via the previous analysis of applications; others enable tool's user to define policies or implement policy decision

algorithms which are then enforced by the system. To extend the Android permission system, some approaches require an adapted version of the operating system, while other tools adapt third-party applications in order to make them compatible with the respective policy tool.

Saint [56] extends Android's permission system to allow application developers to ship security policies with their applications to configure conditions, whereby the application may grant self-defined permissions to other packages. Furthermore, runtime policies allow specifying how components from other packages may interact with the application over *IPC*. The *Saint* framework adapts the operating system to parse these policies while installing the package; however, no policies can be defined by the user nor can application-defined policies be adapted.
To detect and prevent privilege escalation and collusion attacks, *XmanDroid* [57] changes Android's monitoring system to check *IPC* calls for potential maliciousness during runtime. The system is configured by extracting requested permissions from applications during installation as well as by a list of system policies defining conditions under which a component may interact with another, depending on their respective permissions. Aiming for independence from user configuration, *XmanDroid* grant the user the decision to allow a particular intent, provided no expert knowledge is required. Although able to prevent privilege escalation attacks over *IPC* in several test applications, this approach entails several shortcomings: When analyzing over-privileged applications, a high number of false positives disturb the monitoring mechanism, and attacks on the kernel level as well as single solely working malicious applications are not detected.
To protect middleware resources such as *ICC* and kernel-level resources such as *IPC*, *FlaskDroid* [58] introduces security servers on different system levels, which manage security policies for accessing the resources on their respective level and which synchronize with counterparts on other levels when a policy changes. The employed policies were generated via trials monitoring and analyzing user interaction. Evaluating the approach showed a slightly better runtime performance compared to *XmanDroid* [57], although *FlaskDroid* similarly suffers from many false positives when detecting collusion and privilege escalation attacks.
To identify vulnerabilities in a set of applications, *Separ* [59] first extracts data from the manifest and bytecode of applications, which are then used with predefined intent vulnerability specifications to find possible exploits during static analysis. For these exploits, security policies are generated which are enforced during runtime by an *Xposed*-based security extension [54]. During the comparison of the static vulnerability analyzer with *DidFail* [21] and *AmanDroid* [18], *Separ* detected multiple types of vulnerabilities which the fore-mentioned approaches could not. During evaluation on 4,000 applications, *Separ* identified all 385 vulnerabilities while *DidFail* and *AmanDroid* flagged 55% and 86% respectively. This approach does not consider dynamically loaded code due to the lack of dynamic analysis, and the policy enforcement showed an overhead of 11.8% for each *IPC* call.
Sealant [51] aims to detect and restrict malicious intent traffic between multiple applications by combining static analysis to find possible vulnerable *IPC* communication with

an extension of the Android framework to manage intent traffic. The initial analysis of application apks employs *IC3* [7] and *Covert* [25] to create a list of *IPC* paths which are used to configure an extension to the Android framework to monitor the intent traffic and enforces policies. Compared to similar approaches such as *Separ* [59] and *XmanDroid* [57], *Sealant's* analysis raises fewer false positives, although *Separ* can detect more types of vulnerabilities. *Sealant*, like *Separ*, creates automatic polices based on prior analysis, and the overhead of policy enforcement during runtime was measured at a mean of 25 ms per execution.

To allow policy-based control of application interaction without requiring changes to the Android system, *AppGuard* [60] rewrites untrusted applications before installation to enable the user to control the consumption of resources during runtime. The changes to the applications and creation of policies are performed through *AppGuard* which must be installed on the device as a regular third party apk. The instrumented applications show a low performance overhead and high stability during testing, although both factors depend on the employed policies. The rewriting process changes the signing key of the applications however, which can cause issues.

To decouple decision making and policy enforcement, *ASF* [61] alters the Android system to offer a programmable interface for security extensions which are loaded into the Android framework during the boot process. These modules solely manage their policies for resources on both the kernel and middleware levels, similar to *FlaskDroid's* [58] architecture, while the *ASF* framework focuses on executing the decisions of the respective modules. To demonstrate the feasibility of their approach, the authors ported the functionality of *AppGuard* [60] as a security module for the framework, with no loss in functionality. The efficiency of the activated policy modules primarily contribute to the overhead during runtime, the framework itself produces 11.8% overhead compared to systems running a stock Android version.

ASM [62] follows a similar approach as *ASF* [61] by allowing security applications to register for authorization hooks to interact with security-related system functions. A major difference from *ASF* is that the function hooks only allow tightening existing restrictions, such as the allocation of permissions. Although this approach offers less expression potential than *ASF* modules, which are assumed to be trusted by the *ASF* framework, a faulty or malicious third party security module can-not compromise the security of the whole system. During evaluation, the framework itself imposed a performance overhead of 3.3% on the system; with no security module loaded, while including a single running module contributed to an overhead of 9.3%.

Aiming to offer complex system wide policing while avoiding changes to the Android system, *DroidForce* [63] employs *FlowDroid* [9] to analyze applications and afterwards use *Soot* [6] to add the code necessary to enforce policies during runtime. Before the application can perform a privileged operation, the *DroidForce* application installed on the system is queried whether this operation is violating an active policy. While such polices may be adapted by the user during runtime, each update to an instrumented application requires a new analysis cycle. The approach offers a finer-grained policy definition compared to the similar approaches of *AppGuard* [60] and *XmanDroid* [57]

and adds no additional performance overhead during runtime. Because the range of operations monitored by the system is based on *FlowDroid's* [9] initial analysis, this approach inherits its limitations.
Similar to *ASM* [62], *IEM* [64] changes Android's monitoring system to inspect and alter the *IPC* traffic. This functionality is exposed via an application programming interface, which can be consumed by custom security modules by defining security policies and decision-making algorithms, which are then enforced by the *IEM* framework. This approach demonstrates its viability by implementing and evaluating several custom firewall modules in order to block and redirect specific *IPC* traffic. The runtime overhead of the framework itself is shown to be low, with a mean of 5 milliseconds per intent, although the main contribution to the runtime overhead originates from the active security modules.
Sealant [65] aims to handle a wider range of intent-based attacks and for greater precision in their detection compared to previous approaches by initially finding vulnerable *ICC* paths through static analysis on an application's bytecode using *Covert* [25] and *IC3* [7], where potential findings are confirmed via manual verification. The validated vulnerable paths are stored and matched against incoming intents during runtime by an interceptor component based on the user's choice for the respective path. This functionality requires changes to system components such as the ActivityManagerService. When evaluating 1,100 applications, this approach raised fewer false alarms than *Separ* [59] due to its finer-grained evaluation of *ICC* paths. This approach can also detect more types of attacks compared to *XmanDroid* [57], due to its independence from user-created policies.

CHAPTER 3

Background

3.1 Android operating system

The Android OS employs an inter-process communication *(IPC)* system to facilitate interaction between components running in different processes. This system follows a publish–subscribe pattern, whereby receiver components register filters to subscribe to intent messages. Because components are allowed to decide which messages they send and to which messages they subscribe, this mechanism can be misused to eavesdrop on communication and to inject malicious messages into unprepared components.

3.1.1 Components of Android applications

All applications running on the Android operating system are built from four elements or types of components: activity, service, broadcast receiver or content provider. The content provider component is the only one not involved in the *IPC* messaging system. Although an application can employ several components of each type, it is necessary to declare only one of either activity, service or broadcast receiver component to act as a main entry point of the application to allow its interaction with users and the operating system. Using a certain component requires its declaration in the application's manifest, which is a vital configuration file of each application named *AndroidManifest.xml*. The manifest must also declare the package name of the application as well as permissions and features used in the application.

3.1.2 Sand boxed processes

Android applications by default run in their own process, so that no application can access the resources of another. Each application package is treated as a distinctive user with a unique user identifier *(UID)*, which is assigned at the time of the installation and does not change, but can be reassigned after the package is removed. Two applications

sharing the same package name can request to run in the same process by both setting the attribute *sharedUserId* in their respective manifests. To communicate with each other, applications use inter-process communication messages, called intents, to share data and functionality.

3.1.3 Permissions

Permissions restrict applications from accessing sensitive user data and system features. Until API level *23*, an application had to list the permissions intended for use during installation and proceeding with the installation meant granting the requested permissions to the application. The permission system has since been altered so that no permission usage must be granted during installation, bu instead applications must ask each time before performing a privileged operation requiring a dangerous permission, which grants the user the ability to revoke permissions at any time. There are four protection levels dividing permissions: *normal* permissions, which are granted by the system automatically; *signature* permissions, which are signed from third party applications by a certificate; *dangerous* permissions, which involve access to sensitive user data and therefore must be granted by the user; and *special* permissions, which are viewed as particularly sensitive. Applications can customize permissions to require other applications to request these permissions before sharing their functionally. To use a permission declared by another application, the application declaring the permission must be installed first. If both applications declare the same permission, the system does not allow the installation of the second application unless both are signed with the same certificate.

3.2 Intent

An intent is a message used by the Android OS inter-process communication system to perform varied tasks, which can be divided into three use cases: transfer data, start activity or service components. The exact operation, a specific intent triggers is derived via a resolution process from the data encapsulated by the respective intent. An intent's data fields are categorized into primary and secondary ones, depending on their their level of influence on the outcome of the resolution process. As primary data fields are considered action and data, while secondary values include the category, component and mimetype of the data as well as optional data payload and flags. Actions are string values which represent the kind of task the intent performs, for example, *ACTION_VIEW* displays the specified data. An intent can only carry one action at a time, and thus the previous action value is overridden when setting an action on an intent object. The system predefines a set of actions, and some may be used by applications to request specific operations by the system, while others may only be used by the system itself. In addition, each application may define its own actions, and the action of the intent is applied to its data, specified as an URI. An intent allows specifying a particular mimetype, which is usually derived from the data URI itself during the resolution process. Examples of mimetypes include *img/jpg* to describe an image resource in the format *jpg* or *content*

to describe an arbitrary content object. Similarly to actions, mimetypes can be defined by applications, and an intent can specify a list of categories which provide additional information about the action. Categories can be added by applications to extend the list predefined by the system. During resolution, these data values are evaluated to find a suitable target component for the intent. The component field alternatively allows defining the name of a particular component class to use instead. Setting a specific component in an intent makes the values in the data fields optional since there is no need to resolve the target component for the intent. Intents which specify a component are called explicit, while intents which requiring resolution are called implicit. Intents can also carry data payloads as key value pairs, which can be simple predefined data types such as boolean as well as complex custom defined classes. For the later, the class must implement the *Parcelable* or *Serializable* interface to add them as intent payloads. Another method of encapsulating additional information in an intent regards the numerical flag, which has the value *0* if no flag value has been set or a combination of the *28* predefined flag values. A component can retrieve each of these values after receiving the intent.

3.3 Intent filter

Intent filters are data structures specifying the same data fields similarly as intents. Components can declare intent filters to indicate which intents they can handle. Each time an intent is started, the system matches the values declared in the intent against all registered intent filter to decide which components will receive a particular intent. The system does not decide which intent filters are registered by a component, which is free to register for intents it is incapable of handling. Intent filters can be declared statically in the manifest or dynamically during runtime, although the latter option is only available for broadcast receivers. An intent can specify a single action while an intent filter can declare multiple, but one must match the intent's action to pass the filter. Similar to actions, an intent filter can declare multiple categories; however , all specified categories in a filter must be matched by the intent to pass. To filter an intent's data the respective parts of a data URI can be specified separately in the intent filter. Data schemes can be individually declared, but can also be combined with a scheme-specific part to more finely filter. Scheme-specific parts can be added as a regular expression which requires a matching pattern and type. The matching type can be *literal*, which must completely match the given value, while *prefix* only checks the beginning and finally the *glob* value allows matching with a pattern. A scheme-specific part is only considered during the matching process when at least one scheme has been specified in the filter, and the same functionality is used to store and match the path of the specified data value. The data mimetypes must be specified in the format *a/b*, which allows declaring partial types such as *img/**. Furthermore, a list of host and port values can be specified for matching. In contrast to RFC host names, schemes and data types, the comparison used in the intent filter is case sensitive. Every intent filter retains a priority value which is *0* by default but can be changed to a value between *-1,000* and *1,000*, with higher numbers filters

receiving broadcasts before those with lower numbers. The priority of activity filters can only be set when statically declaring filters in the manifest file.

3.4 Activity

Activities are designed for direction interaction with human users since they represent the only components which can contain UI elements such as buttons and text fields. When an activity is started by the system, it passes through several stages of a predefined life-cycle, from the initial creation to the termination of the component. Each time a component enters one of the stages, it executes the defined code for the respective stage.

3.4.1 Declaring an activity

Activities must be declared in the manifest file by adding the *activity* tag, which must at least contain the name of the activity and may also define additional attributes. Figure 3.1 shows the declaration of different activity components as described below. Declaring an intent filter inside of an activity tag, such as *ActivityOne* does, automatically marks the component as exported and allows the activity to be started by an implicit intent. Specifying the action *android.intent.action.MAIN* and the category *android.intent.category.LAUNCHER* in an intent filter declares this activity as the main activity of the application, and this activity will be shown when the application is started (*ActivityTwo*). When declaring no intent filter and manually setting the *exported* attribute to *true*, the activity can be launched by an explicit intent started by another application than that declaring the activity (*ActivityThree*). If the activity is not exported in either manner, it can only be started by a component of its own application (*ActivityFour*). By listing a permission in an activity, only an application with this permission may start it (*ActivityFive*).

```
<activity android:name="example.exampleapplication.ActivityOne">
      <intent-filter>
            <action android:name="example.exampleapplication.EXAMPLE_ACTION"/>
            <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
</activity>

<activity android:name="example.exampleapplication.ActivityTwo">
      <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
</activity>

<activity
      android:name="example.exampleapplication.ActivityThree"
      android:exported="true">
</activity>

<activity android:name="example.exampleapplication.ActivityFour">
</activity>

<activity
      android:name="example.exampleapplication.ActivityFive"
      android:exported="true"
      android:permission="android.permission.READ_CONTACTS">
</activity>
```

Figure 3.1: Declaring activity components

3.4.2 Starting an activity

To start an activity, the calling component has to invoke the system-provided function *startActivity*, or one of the related methods, on a suitable intent. *ActivityOne* declared an intent filter and hence is exported by default, and an implicit intent specifying the action *EXAMPLE_ACTION* would match the intent filter (as shown in Figure 3.2), and it can additionally be started explicitly in the same manner as described below for *ActivityThree*. *ActivityTwo*, as the application's default activity, would be displayed when the package's main activity is defined in an intent, such as when the application's launch icon is clicked, as shown in Figure 3.3. *ActivityThree* is flagged as exported, and therefore any application can launch the activity with an explicit intent (see Figure 3.4). A similar intent can be used to launch *ActivityFour* although the intent must be launched by its own package since this activity is not exported. *ActivityFive* was exported but was however secured with the permission *VIEW_CONTACTS* and thus can only be started by an application holding this permission. To prevent exception's caused by starting an intent with no suitable activity to launch, the method *queryIntentActivities* can be invoked, which returns a list of activities which can resolve the intent (see Figure 3.5). If multiple activities are suitable to handle an implicit intent, the system lists them in a dialog for the user to choose, unless a default was previously chosen. To receive a

result from an activity once it is closed, the activity can be started by calling the method *startActivityForResult* instead of *startActivity* as shown in Figure 3.6. This method accepts an integer value which is used as an identifier to retrieve the result after the activity terminates (see Figure 3.7).

```
Intent intent = new Intent();
intent.setAction("example.exampleapplication.EXAMPLE_ACTION");
startActivity(intent);
```

Figure 3.2: Launching an activity with an implicit intent

```
Intent intent = getPackageManager().
                    getLaunchIntentPackage("example.exampleapplication");
startActivity(intent);
```

Figure 3.3: Launching the default activity

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                        "example.exampleapplication.ActivityTwo"));
startActivity(intent);
```

Figure 3.4: Launching an activity with an explicit intent

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                       "example.exampleapplication.ActivityThree"));
if(getPackageManager().queryIntentActivites(intent,0).size() > 0)
                         startActivity(intent);
```

Figure 3.5: Resolving targets for an activity intent

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                      "example.exampleapplication.ActivityThree"));
startActivityForResult(intent, 42);
```

Figure 3.6: Awaiting a result from an activity

```
protected void onActivityResult(int requestCode, int resultCode, Intent data){

super.onActivityResult(requestCode, resultCode, data);
            if(requestCode == 42) data.getStringExtra("result");
}
```

Figure 3.7: Retrieve a result from an activity

3.5 Service

Services are intended to perform background tasks and thus cannot contain UI elements to allow direct user interaction. Services run on the application's main thread by default, and derived classes such as *IntentService* provide a worker thread to sequentially handle requests and perform given tasks. Similar to activities, services have a life-cycle, in which the component is created and destroyed after finishing its task.

3.5.1 Declaring a service

Services can run in the foreground, which means they are granted high priority by the system and thus likely the last to be terminated during a memory shortage. To keep the user informed of such currently running services, a foreground service shows a notification in the taskbar which can not be closed by the user. Background services run unnoticed by the user since they are not required to show a notification, and they also are lower priority compared to foreground services. Services may allow binding, which means interacting with other components through a client interface called *Binder*. This allows even components from other applications to interact with the service using inter-process communication. Like activities, service components must be declared in the application's manifest, with *name* again representing the only mandatory attribute. Service declarations allow using the *exported* attribute to make the service available to other packages besides the one declaring the service component. Like activities, the interaction with a service can be restricted with permissions (see Figure 3.8). Although service components can also specify intent filters to flag them as exported, they cannot be started or bound through an implicit intent. This behavior was introduced for security reasons in API level *21*.

```
<service
      android:name="example.exampleapplication.ExampleService"
      android:exported="true"
      android:permission="android.permission.READ_CONTACTS">
</service>
```

Figure 3.8: Declaring a service component

3.5.2 Starting a service

Similar to launching an activity, Android offers several methods to start a service component, where the basic method is *startService*. When using an explicit intent the service specified in the intent is started directly (see Figure 3.9), when using an implicit intent, the service component is chosen by resolving the intent against all intent filters declared in service components, with the latter option available only on systems running a version of Android lower than API level *21*. Once a service starts via *startService*, it will run indefinitely until *stopService* is called on the component or the service itself invokes the method *stopSelf*. The system prioritizes a running service higher when a component is bound to it, while the priority of a service running in background will be lowered by the system until it may be terminated. The service can be started again with the *START_STICKY* flag, which signals the system to restart the service when terminated in this fashion. To bind to a service, the service must implement the binder functionality as shown in Figure 3.10, while the component using the binding must implement the binder connection (see Figure 3.11) and invoke *bindService* with a suitable intent (see Figure 3.12). After binding to the service, the component can invoke public methods declared by the service. Similar to activity components, services can be secured by permissions, which a component must acquire before being allowed to start or bind to a secured service.

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                    "example.exampleapplication.ExampleService"));
startService(intent);
```

Figure 3.9: Starting a service

```
public class LocalBinder extends Binder {
      ExampleService getService() {
            return ExampleService.this;
       }
}

@Override
public IBinder onBind(Intent intent) {
      return new LocalBinder();
}
```

Figure 3.10: Service declaring a binder

```
private ServiceConnection connection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName className, IBinder service) {
        ExampleService.LocalBinder binder = (ExampleService.LocalBinder) service;
        service = binder.getService();
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
    }
};
```

Figure 3.11: Component declaring a connection to a service binder

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                    "example.exampleapplication.ExampleService"));
bindService(intent, connection, Context.BIND_AUTO_CREATE);
```

Figure 3.12: Binding to a service

3.6 Broadcast

Besides starting components as described above, intents can be used as broadcast messages to transport data from one component to another, to trigger certain operations by the operating system or receive a notification when such an operation has been performed.

3.6.1 Global receiver

To receive broadcast intents, the abstract class *BroadcastReceiver* and its method *onReceive* require implementation, which are executed each time an intent is delivered to the receiver and allow the component to obtain and process the incoming intent. To inform the system that a component is supposed to receive intents, the receiver must be declared in the manifest file or dynamically registered during runtime. A manifest-declared receiver allows the system to start an application if it is not already running when the broadcast is performed. If an application starts in this manner, the system may close it again after the code inside the *onReceive* method concludes, since the system no longer highly prioritizes the component hosting the receiver. The API level 26 receivers for implicit broadcasts can no longer be declared in the manifest; however there are exceptions to this restriction for broadcasting-system related events such as newly connected devices or the completion of the boot process [66]. To declare a static receiver, the *receiver* tag has to be added to the manifest, and setting the *exported* attribute to *true* will allow other packages to send explicit broadcasts to the receiver. Adding an intent filter exports the component as well and enables the receiver to be triggered by suitable implicit broadcasts, (see Figure

3.13). A dynamically registered receiver must be declared by calling *registerReceiver* in the respective context 3.14. The receiver is active as long as this context is valid, which means that if registered in an activity context, the receiver will run as long as this activity remains in the foreground. When registering in the application context, the receiver is active until the application is closed. A receiver can be unregistered when no longer needed by calling *unregisterReceiver*. If a receiver is not properly unregistered before the context is destroyed, an exception is thrown due to the occurring memory leak. All dynamically registered broadcast receivers declaring an intent filter are exported. Both static and context-declared receivers can specify permissions to ensure that only broadcasts which are sent with the respective permissions can execute the receiver.

```
<receiver
      android:name="example.exampleapplication.MessageReceiver"
      android:exported="true">
</receiver>

<receiver android:name="example.exampleapplication.MessageReceiver">
      <intent-filter>
            <action android:name="example.exampleapplication.EXAMPLE_ACTION"/>
      </intent-filter>
</receiver>
```

Figure 3.13: Register a receiver statically

```
MessageReceiver receiver = new MessageReceiver();
IntentFilter filter = new IntentFilter();
filter.addCategory(Intent.CATEGORY_DEFAULT);
filter.addAction("example.exampleapplication.EXAMPLE_ACTION");
registerReceiver(receiver, filter);
```

Figure 3.14: Register a receiver dynamically

3.6.2 Global broadcast

To start a global broadcast, a component must call the function *sendBroadcast* on an intent. If no target component is set for the broadcast, the intent is implicit and resolved by comparing the fields containing data to all registered broadcast intent filters, (see Figure 3.15). This is called a global broadcast, since it is delivered system wide to any subscribed receiver in random order, and no receiver knows which or how many other receivers received this broadcast. The broadcast can not be aborted by any receiver nor can a receiver process the result from the preceding receiver. If a target component is set, the intent is directly delivered only to the specified receiver, provided the component in question exists and is exported, in case the sender is part of another application package (see Figure 3.16).

```
Intent intent = new Intent();
intent.setAction("example.exampleapplication.EXAMPLE_ACTION");
sendBroadcast(intent);
```

Figure 3.15: Starting an implicit broadcast

```
Intent intent = new Intent();
intent.setComponent(new ComponentName("example.exampleapplication",
                    "example.exampleapplication.MessageReceiver"));
sendBroadcast(intent);
```

Figure 3.16: Starting an explicit broadcast

3.6.3 Ordered broadcast

Ordered broadcasts are global broadcasts sequentially delivered to each suitable receiver according to the priority value in the matching intent filter, where receivers with the same value are executed in arbitrary order. After processing the broadcast, each receiver can propagate the result to the next receiver or abort the broadcast. An ordered broadcast is started by invoking the function *sendOrderedBroadcast* on a suitable intent (see Figure 3.17).

```
Intent intent = new Intent();
intent.setAction("example.exampleapplication.EXAMPLE_ACTION");
sendOrderedBroadcast(intent);
```

Figure 3.17: Starting an ordered broadcast

3.6.4 Sticky broadcast

While global broadcasts are inaccessible after processing, a sticky broadcast is retained and accessible for a certain amount of time until removed, which can be used to keep information frequently required by multiple applications, such as the battery status, quickly accessible at any time. If a suitable receiver is registered while the sticky broadcast is active, the broadcast is delivered to the receiver upon registration. To start a sticky broadcast (as shown in Figure 3.18), the sender requires the permission *android.permission.BROADCAST_STICKY*.

```
Intent intent = new Intent();
intent.setAction("example.exampleapplication.EXAMPLE_ACTION");
sendStickyBroadcast(intent);
```

Figure 3.18: Starting a sticky broadcast

3.6.5 Local broadcast

The use of a local broadcast is advised when a broadcast is not expected to reach receivers outside of the sending package. Similar to a global receiver, a local receiver can be registered during runtime (see Figure 3.19). While the broadcast types discussed above involve inter-process communication, a local broadcast is only propagated to receiver inside the application in which the broadcast originated, which makes them more resource efficient since the resolution process does not involve all system wide registered receivers.

```
LocalBroadcastManager localBroadcastManager =
                    LocalBroadcastManager.getInstance(this);
localBroadcastManager.registerReceiver(receiver, filter);
Intent intent = new Intent();
intent.setAction("example.exampleapplication.EXAMPLE_ACTION");
localBroadcastManager.sendBroadcast(intent);
```

Figure 3.19: Starting a local broadcast

3.6.6 Sending with permissions

If the receiver of a broadcast has specified a permission, the broadcast must be started with the same permission to trigger the respective receiver, and the sending application requires the permission in question as well to start such a broadcast.

3.7 Pending intent

A pending intent allows passing a specific intent to another process without starting the intent. The receiver of this intent may launch the intent using the identity and permissions of the creating application. Pending intents enable an application to start an intent at a certain time or event after the application itself been closed. A pending intent is an activity, broadcast or service, has its setup defined by the creating application and may be directly passed to a system process such as the alarmmanager service (see Figure 3.20) or sent within a broadcast intent (see Figure 3.21).

```
PendingIntent pendingIntent = PendingIntent.getActivity(this, 1
      new Intent(), PendingIntent.FLAG_UPDATE_CURRENT);
AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC, 10000, pendingIntent);
```

Figure 3.20: Passing a pending intent to the alarmmanager

```
PendingIntent pendingIntent = PendingIntent.getActivity(this, 1
      new Intent(), PendingIntent.FLAG_UPDATE_CURRENT);
Intent intent = new Intent("example.exampleapplication.EXAMPLE_ACTION");
intent.putExtra("pendingIntent", pendingIntent);
sendBroadcast(intent);
```

Figure 3.21: Sending a pending intent in an implicit intent

3.8 Resolving intents

The system performs a matching process to identify which components will receive a certain intent, which must be as effective as possible since the inter-process communication is a central and vital part of the operating system. Unnecessary time-consuming operations lead to noticeable delays in the launch of components and delivery of messages, to avoid such issues, the Android OS stores all registered intent filters in data structures named *IntentResolver* to efficiently manage recipients of started intents.

3.8.1 Intent resolver

An intent resolver retains a number of data structures to effectively store registered intent filters, which are stored in a global list and added to other lists depending on the respective values specified in the intent filter. Besides the global filter list, six additional data structures retain intent filters associated with particular actions, mimetypes and URIs. When an intent filter is unregistered, it is removed from all lists.

3.8.2 Resolving process

To check whether a registered intent filter matches an intent, the relevant intent is passed to the resolver by invoking the method *queryIntent*. The resolver then evaluates the parameters set in the passed intent to decide which collections to query for suitable intent filters. In several iterations, the resolver collects the matching intent filter, where the resolver first queries based on the mimetype whether any such type has been resolved for the intent. Afterwards, all intent filters matching the intents scheme are added before adding all filters with the intent's action. After each collection iteration, the built list of filters is checked for cases where a suitable filter was already added in a previous iteration

or a filter from a package matches while the intent is only intended for the intent sender's own package. After matching the intent against all filter candidates by invoking each filter's *match* function, the matching filters are sorted by priority value defined in the filter and returned to the calling process.

3.8.3 Intent filter matching

Whenever a resolver passes an intent to the filter's *match* function, firstly the intents action is evaluated and only passes if the filters action list contains the intent's action value. Although an empty action might pass the test in a filter, the intent resolver's matching process will discard the match. The intent's scheme is then matched against the filter and passes if the filter declares the same scheme as the intent. If the intent does not contain a scheme, the filter matches only if it specifies no scheme or an empty string. Intents which specify *file*, *content* or an empty string as scheme value will pass the test, even if the filter declares no schemes. The data property of the intent is only evaluated if the scheme test has been passed, and the intent's URI must match one of the the filter's data authorities to pass. If the filter specifies no data authorities, the intent passes even if it contains a data URI; however, if the intent does not specify an URI, it only passes a filter with no data authorities. If an intent filter specifies no mimetypes, only an intent which also specifies none is allowed to pass. If the filter lists any mimetypes, the intent must match one of them to pass. Filters specifying all possible types (**/**) match all intents with a mimetype; intents defining **/** as type are allowed to pass if the filter contains at least one mimetype; and if a filter contains a partial mimetype such as *audio/**, all intents matching the base type pass the test. If the intent specifies a partial type, the filter must contain at least one entry matching this base type.To pass the category test, an intent filter must contain all the categories specified by the intent. Every intent automatically contains the *CATEGORY_DEFAULT* category, which means a filter requires the same if any implicit intents become accepted by the filter.

3.9 Android intent firewall

The *ActivityManagerService* identified a valid target component for a launched event, which is passed to the intent firewall (IFW) for inspection before delivery. Depending on the firewall configuration the intent is either blocked or allowed to be propagated to its destination component. The intent firewall was initially added to the Android system in version 4.4, and after receiving minor changes in version 5.1 its functionality has not been changed or upgraded.Configuring the firewall with modular rules enables motoring and regulating the global intent message traffic. After the *com.android.server.firewall* package is loaded during the boot process, the firewall begins to monitor the secure system directory for changes. Whenever a change is made to the directory, all contained *XML* files are checked for firewall rules, and the rules currently active in the firewall's configuration are dropped during the process and replaced by the newly parsed ones.

3.9.1 Firewall rules

Firewall rules are exclusively defined for activity, broadcast or service intents and must specify whether matching intents should be blocked or logged and allowed to be propagated. To be matched with intents, rules may hold three data structures: a list of intent filters, a list of intents' target component names and a list of additional filters nested in tree form as described below. While a firewall rule may contain multiple intent filters and target component names, at least one of either must be specified to be considered valid. Specifying additional filters is optional to allow more finely grained matching. Figure 3.22 shows the makeup of a basic firewall rule filtering for a destination component and for intents with a specific action. A finer filtering is used in the rule as shown in Figure 3.23, where after the initial match of the defined component, the intents action is checked to ensure it starts with a certain value. In addition, the intent must define a specific port value range or may not carry a specific category to pass the filtering. Firewall rules may employ the following additional filter types:

Logical filter
Logical filters can nest one or multiple filter types to combine the matching results. The *and-filter* holds a list of child filters, and during the matching process all nested filter must match, to allow the *and-filter* to match. The *or-filter* can nest a list of filters, one of which must match for the *or-filter* to match. The *not-filter* holds one child filter and inverts the matching result of this filter. The root filter holding all additional filters in a firewall rule is treated as an *and-filter*, and therefore all containing filter must match.

Category filter
The *category-filter* stores the name of a single category which must be matched by the intent to pass the test and allows distinguishing between intents matching an intent filter specifying several categories.

Port filter
Using a *port-filter* allows filtering for intents with a specific port or port value in a specific range and defining a *port-filter* without a port value will match all intents specifying any port value.

Sender filter
The *sender-filter* specifies if the sender of an intent is either the system or has the same *UID* or signature as the receiver.

Sender permission filter
This filter allows checking for a specific permission required by the intent's sender to allow the intent to match.

String filter
The *string-filter* holds an intent property name and string value, which must be partially or fully matched by the respective intent property. A pattern or regular expression can also be specified instead of a specific value, if no matching type is passed, the filter checks for the existence of the specified property.

```
<rules>
  <activity block="true" log="true">
    <component-filter name="example.exampleapplication.ExampleActivity"/>
      <intent-filter>
        <cat name="android.intent.category.DEFAULT"/>
        <action name="example.exampleapplication.EXAMPLE_ACTION"/>
      </intent-filter>
  </activity>
</rules>
```

Figure 3.22: Basic rule with intent filter and component name filter

```
<rules>
  <activity block="false" log="true">
    <component-filter name="example.exampleapplication.ExampleActivity"/>
    <and>
      <action contains="exampleapplication"/>
      <or>
        <not>
          <category name="example.exampleapplication.ExampleCategory"/>
        </not>
        <port min="2100" max="2103"/>
      </or>
    </and>
  </activity>
</rules>
```

Figure 3.23: Fine-grained rule filtering for category and port values

3.9.2 Rule resolving

To find matching rules, the firewall builds upon the intent resolver mechanism described above. Depending on the type of the incoming intent, the resolver holding activity, broadcast or service rules are queried for matching intent filter and target component names. An intent must match at least one intent filter or component name filter for the rule to be considered an initial match. Rules containing several matching intent filter or target components are only added once to the list of matching rules. In a second matching phase each rule in this list is checked for matching additional filters by calling the *match* function of the root filter. This filter is regards an *and-filter* which will match if none of its nested child filters mismatch, which means that it will also match if no child filters are specified.

3.9.3 Shortcomings

The current implementation of the intent firewall system entails shortcomings. Since files containing firewall rules must be deployed into a folder in the system's root directory, the file system must be booted in *write mode*. Depending on the ROM in use, regular

operation requires the device's file system to be rebooted into *read only* mode after deploying the rule files. Because rules are automatically parsed and activated once deployed, this process error prone since one faulty rule can prevent the device from booting correctly. Furthermore, the current architecture prevents users from composing rules which combine intents and component filters or which filter intents based on their senders' package name. Finally, the output of successfully matched intents is sparse and written directly to the system log file, which is inconvenient for usage.

3.10 Intent-based attacks

The implementation of Android's *IPC* system contains several weaknesses which can be exploited by malicious applications to eavesdrop on communication or manipulate other applications by misusing intents. Some attacks described below can be prevented by application developers abiding by the best practice guidelines, and by application users being careful and vigilant.

Activity hijacking
As described above, activities can be launched by implicit intents to allow the system to search for suitable target applications, which are presented to the user for selection. This process may be abused by malicious applications by providing, or pretending to provide, greater functionality compared to the other candidates in order to trick the user into choosing the malicious application. The use of a misleading name for the attacking activity might cause the user to be unaware when choosing an unintended activity component. Once started, the malicious activity can perform several attacks, as outlined in Figure 3.24. In a basic form, the attacker can read any payload carried by the intent ①. In a more complex scenario, the malicious activity could disguise itself using an identical user interface as the benign application's activity to trick the user to input sensitive data, which could then be used in a further attack ②. Furthermore, the attacker can relay the user to another activity using an explicit intent, even the one it previously impersonated, and pass a changed data payload with the intent ③. If the initial sender of the intent expects a result from the invoked activity, the attacker can use this opportunity to pass malicious data back to the caller ④. The success of this attack relies on the user's chosen activities which are allowed to execute a certain intent. A malicious activity may misuse sensitive intents over a period of time without notice by the user. Application developers may mitigate the attack's impact by not passing sensitive data via implicit intents or by protecting them using strong permissions, while data received from other applications must be thoroughly sanitized.

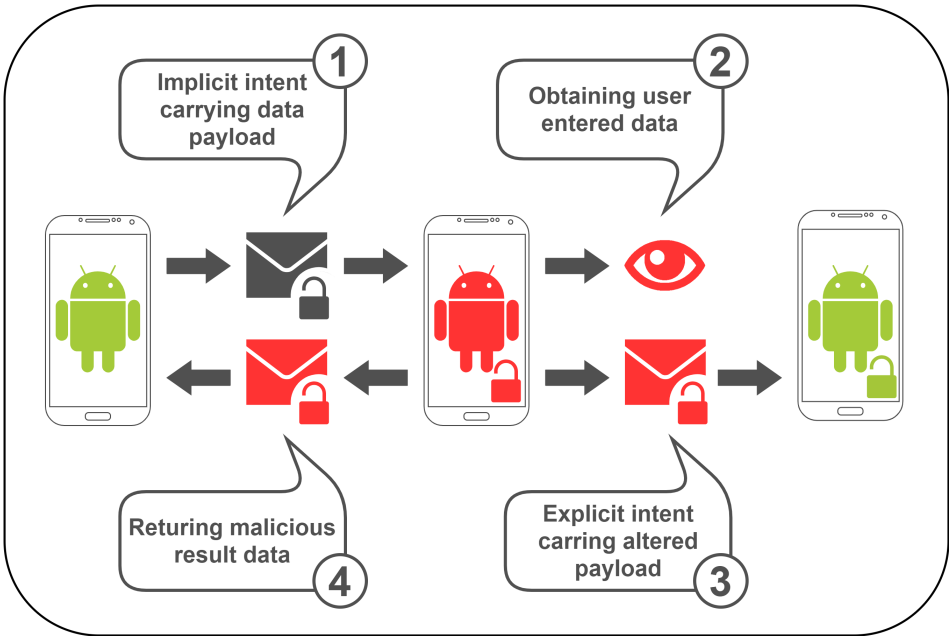

Figure 3.24: Activity hijacking

Activity injection

Exposing activities allows applications other than the declaring one to launch their components (see Figure 3.25), which is dangerous since an activity intended only for internal use, may not expect to receive input from another context and therefore may not sanitize received data before use. In addition, simply starting the activity may unexpectedly alter the state of the application ①, and the invoked activity may incautiously return sensitive data to any calling component ②. The attacker could also invoke an exposed activity of an exploitable application to trick the user into believing this activity belongs to the attacker's own application. The user would then perform operations, such as changing settings, in a benign application while believing these actions to be performed in the malicious application ③. This attack may be effectively countered by application developers avoiding exporting activities which are not designed to handle communication with other applications [2]. Since an intent filter automatically exports an activity, components should not handle sensitive internal logic when also designated to participate in outside communication, and thorough input data sanitation should occur when receiving intents. The use of permissions on an activity can prevent its launch by applications without these permissions.

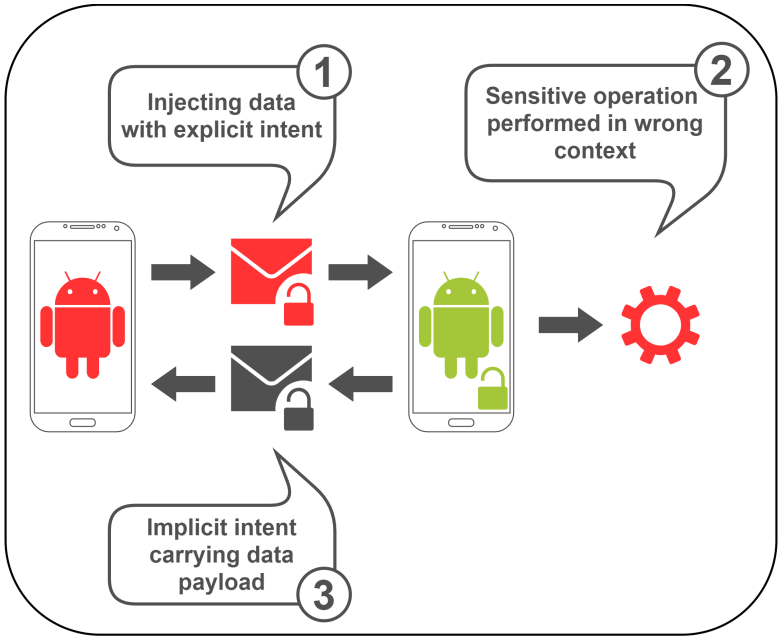

Figure 3.25: Activity injection

Broadcast eavesdropping

Since implicit broadcasts utilize a publish-subscribe pattern, every implicit global broadcast intent is delivered to every suitable broadcast receiver registered before the broadcast started. This behavior is intended but can be misused to eavesdrop on messages by malicious receivers. An attacker can register a broadcast receiver with various intent filters specifying a wide range of values to capture as many broadcasts as possible (see Figure 3.26). To prevent broadcast eavesdropping, implicit global broadcasts should only be used if the sent intent is supposed to reach destinations outside its application's context, otherwise local broadcasts offer a more efficient and secure alternative. Furthermore, if intents carrying sensitive data are sent globally, they should be made explicit or protected by strong permissions.

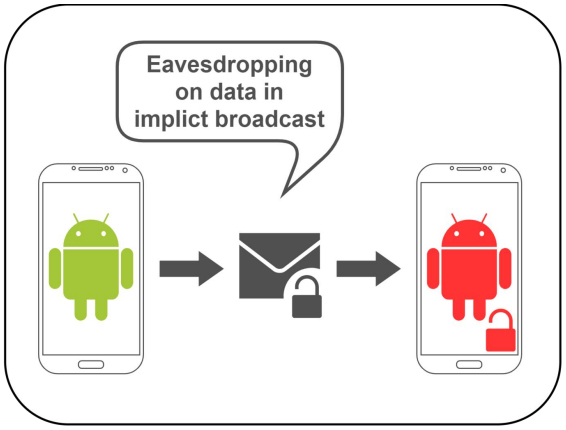

Figure 3.26: Broadcast eavesdropping

Sticky broadcast eavesdropping
Sticky broadcasts pose a security risk since they are not dismissed after the initial broadcast is completed but rebroadcasted to newly registered receivers as outlined in Figure 3.27. Since they can not be protected by permissions like normal broadcasts can, sticky broadcasts have been declared *deprecated* since API level *21*; if employed nonetheless, they should not be used to broadcast sensitive data.

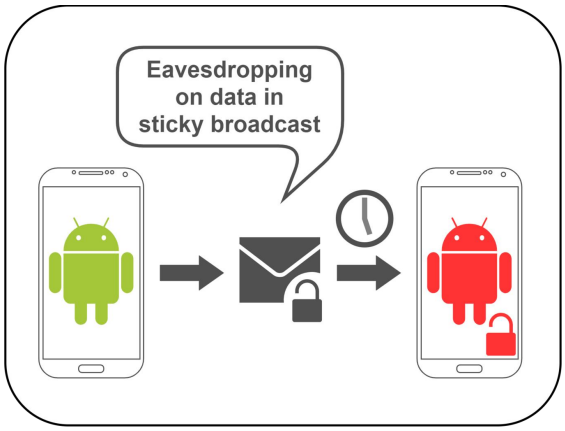

Figure 3.27: Sticky broadcast eavesdropping

Ordered broadcast interruption and result spoofing
Ordered broadcasts are also vulnerable to attacks since they are propagated in a serial fashion to all communication participants, and thus a malicious receiver can register with the highest priority to receive the intent first and can abort the broadcast (see Figure 3.28). Furthermore, the result value as well as result data can be set by each receiver before propagating the broadcast to the next one, which can be used by an attacker to distribute malicious data to the subsequent receiver as shown in Figure 3.29. Since the

mechanism of ordered broadcast intents allows an application to interrupt the broadcast and return an arbitrary result, each application developer must be aware of this process and employ input sanitation when using this type of broadcast.

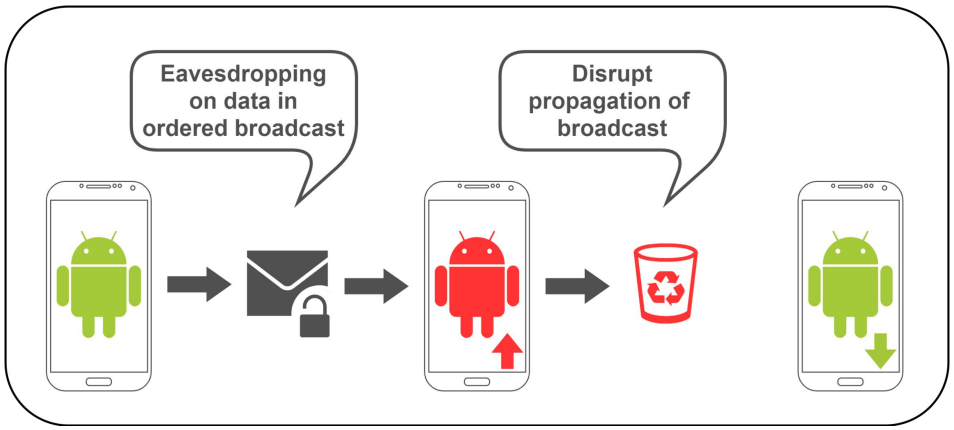

Figure 3.28: Ordered broadcast interruption

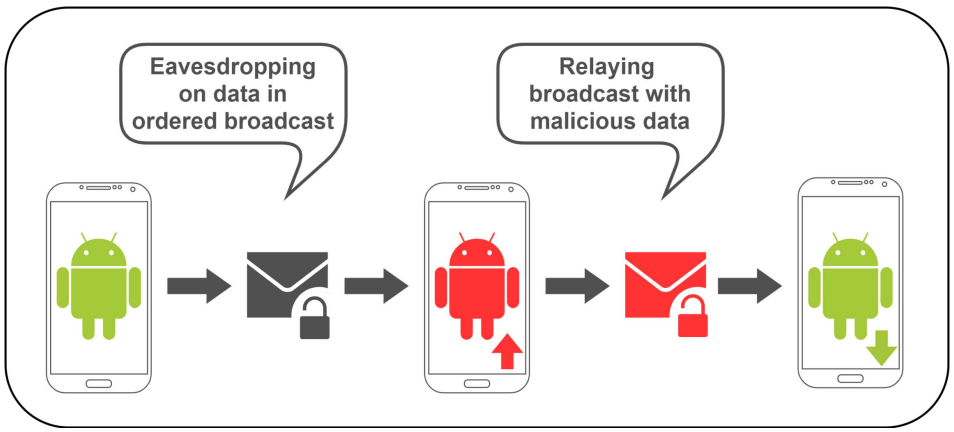

Figure 3.29: Ordered broadcast result spoofing

Broadcast injection

Exported broadcast receivers can be addressed from outside the respective application through implicit or explicit intents. If the application has exposed the receiver unintentionally or is not thoroughly sanitizing the data input, an attacker can inject data or trigger operations inside the exploitable application (see Figure 3.30). Similar to protecting activity components, broadcast receivers offer a smaller attack surface when only handling external communication, thus sanitizing received data before use, while internal communication is performed via local broadcasts.

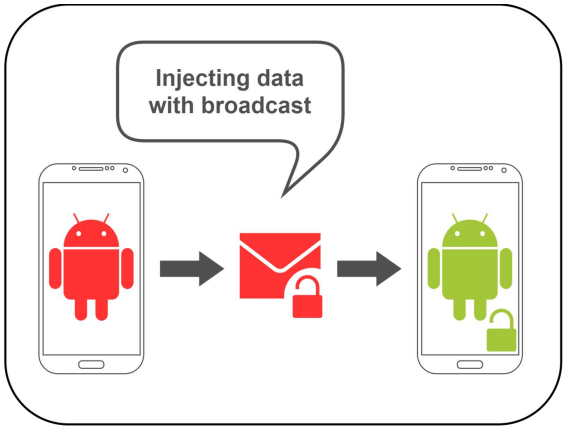

Figure 3.30: Broadcast injection

Service hijacking
Service components can only be started and bound by explicit intents since API level *21*, making it more difficult to trick an application to invoke another service component than intended. A vulnerable application could still be tricked if the package name of the target service component is set dynamically and an attacker manages to alter this value via the exploits described above (see Figure 3.31). Such an attack would be especially problematic, since service components are designed to process business logic which may involve sensitive data and to handle asynchronous tasks such as server communication. Furthermore, unlike activities, services function in the background, unnoticed by the user. If an unsuspecting component connects to the malicious service, the data in the intent could be read (see Figure ①). If the targeted component binds to the service and the anticipated methods are offered by the malicious service, sensitive data could be obtained by the attacker and malicious values could be passed to the invoking component (see Figure ②). To prevent such attacks, application developers must be aware when starting and binding to services which are not hard coded but instead resolved dynamically by the application logic.

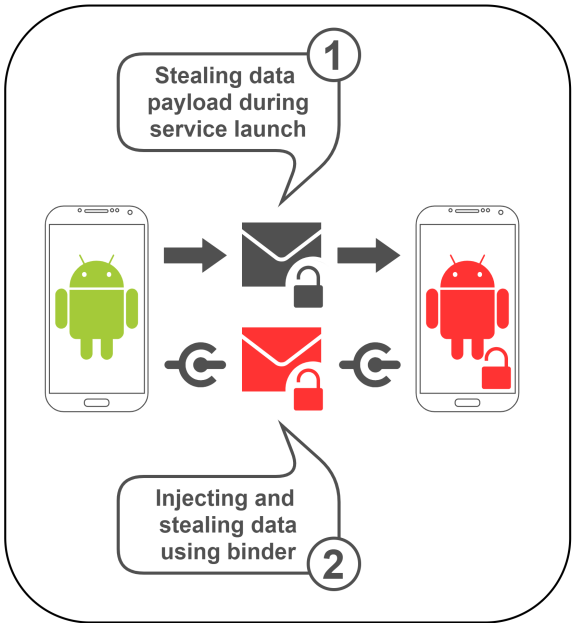

Figure 3.31: Service Hijacking

Service injection
Similar to activities, service components face the risk of abuse by malicious applications if exposed intentionally or otherwise. An attacker can execute a variety of problematic operations, depending on the service's function and its exposed methods (see Figure 3.32). Malicious values can be injected if the service uses data from the intent's payload ①. If the service allows binding, the attacker may invoke public methods to pass and receive data, depending on the method's implementation ②. The service may also be stopped by a suitable intent in the same manner it starts. To prevent such attacks, services should only be exported when they are intended and equipped to serve external applications and should thoroughly sanitize input data. Like activities and broadcast receivers, services should not bundle internal and external functions into one component, while services designated for internal functionality should not use intent filters to avoid automatic exportation.

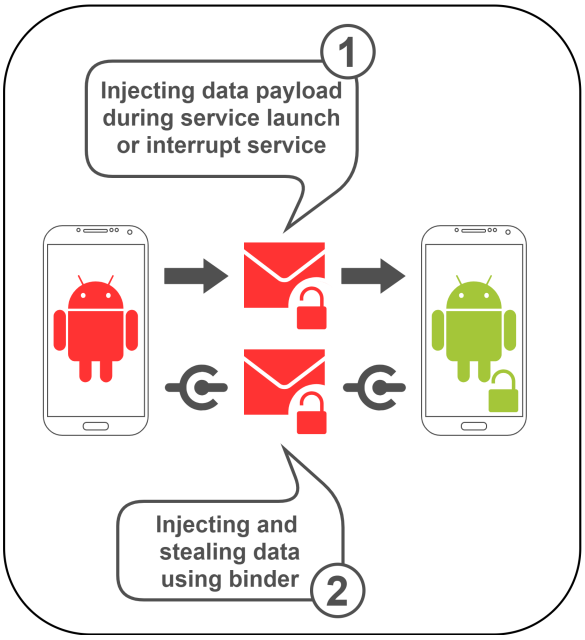

Figure 3.32: Service injection

Pending intent abuse

Pending intents allow applications to perform operations with another package's identity and permissions and thus present a target for misuse. Following the example of [10], Figure 3.33 shows a malicious application obtaining a pending intent created from an implicit base intent, which is sent via implicit broadcast intent. Because the base specifies no values, the malicious application may use the pending intent to execute arbitrary actions using the permissions of the intents creator application. The vulnerable application holds the permission to make phone calls, unlike the malicious application, which however can now trigger phone calls using the permission of the pending intent's creator (see Figure 3.34).

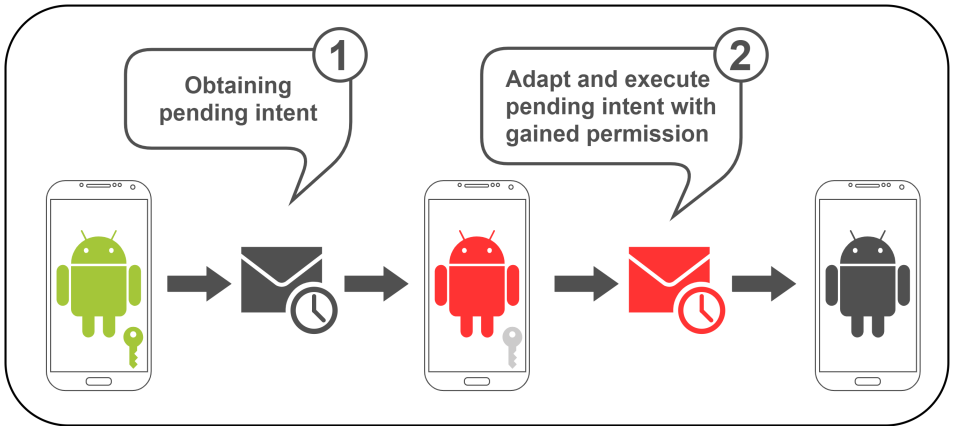

Figure 3.33: Using an pending intent to perform privileged operations

```
// vulnerable application
Intent baseIntent = new Intent();
PendingIntent pendingIntent = PendingIntent.getActivity(this, 1,
       baseIntent, PendingIntent.FLAG_UPDATE_CURRENT);
Intent implicitWrappingIntent = new Intent(Intent.ACTION_SEND);
       implicitWrappingIntent.putExtra("pendingIntent", pendingIntent);
sendBroadcast(implicitWrappingIntent);

// malicious application
Bundle extras = intent.getExtras();
PendingIntent pendingIntent = (PendingIntent) extras.get("pendingIntent");
Intent newIntent = new Intent(Intent.ACTION_CALL, Uri.parse("tel:123456789"));
pendingIntent.send(context, 2, newIntent, null, null);
```

Figure 3.34: Using an pending intent to misuse privileges

As discussed above, the *IPC* system offers several attack vectors for malicious applications to obtain sensitive data, manipulate other applications or gain unauthorized privileges. Vulnerability research shows that the recommendations discussed above best enable preventing or mitigating the repercussions of such attacks [29]. Application developers are advised to designate their components as internal and external, with the latter viewed as inherently vulnerable and focus on input sanitation while employing strong permissions for communication, while also supervising which data are allowed to leave the applications context [1, 3, 4, 12, 27, 29, 33]; however a user has no indication if this has been the case when installing an application.

CHAPTER 4

Enhanced intent firewall

4.1 Enhanced intent firewall

To remedy the issues of the original intent firewall implementation as described in 3.9, the *Enhanced intent firewall (EFW)* application builds on its design and aims finer-grained filtering, modular extension points and easier handling. The *EFW* is divided into two parts: the frontend application which handles user interaction and the backend which contains the firewall logic. While the frontend runs in the application's own context, the backend employs the *Xposed* framework to run in the operating systems context to alter the functionality of the native Android intent firewall.

4.1.1 Xposed framework

The *Xposed* framework allows user-created applications to perform operations or alter the Android operating system's behavior in a manner which is usually only possible for system applications. The use of the *Xposed* framework requires a rooted device with the framework installed along with the *Xposed* installer application to manage *Xposed* modules, which are user-created Android applications utilizing the framework. *Xposed* offers convenient methods to access class variables and hooking functions to execute custom code before, after or instead of the original function's code. During the devices boot process, hooks declared in active modules are placed on their respective target functions. All code executed in a function hook will run in the context of the hooked package. To use the *Xposed* framework, an Android application must declare which class acts as an entry point for the framework and the minimum version of the framework required to use the module. The fully qualified class name of the entry point class must be provided in a file *named xposed_init* which has to be placed inside the assets folder of the module application. The framework version must be specified in the *xposed* tag in the manifest file, which is *30 (Android API 5.1)* for the *EFW* application.

4.1.2 Hooked methods for intent interaction

To interact with intents reaching the intent firewall, function hooks are placed on the respective methods responsible for handling intents, as listed in Figure 4.1. All global intents with a valid target component are passed to one of these methods depending on their type before delivery to their destination. The original functionality of the three methods has been completely replaced by custom code in which the *Enhanced intent firewall* is invoked to handle the intents. As before, the methods listed in the table will return *true* to allow intents to be propagated to their destination and *false* if the operating system should discard them.

Class	Method
com.android.server.firewall.IntentFirewall	checkStartActivity
com.android.server.firewall.IntentFirewall	checkService
com.android.server.firewall.IntentFirewall	checkBroadcast

Figure 4.1: Hooked methods of the intent firewall package

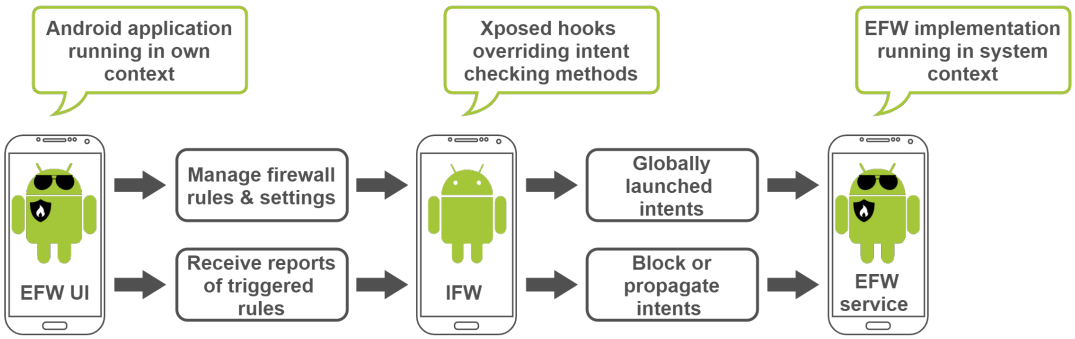

Figure 4.2: Enhanced intent firewall

4.1.3 Firewall rules

Similar to the original intent firewall implementation, the *Enhanced intent firewall* defines a structure in which firewall rules must be specified. One difference from *IFW* is that *EFW* rules are not written in *XML* files and copied into the respective system folder but rather are uploaded via the *EFW* user interface in *JSON* format. Figure 4.3 shows the structure of *EFW* firewall rules. As before, rules must specify a rule level, rule type and set of filters to be matched with candidate intents. The rule level can be *log* to register an occurrence of a matching intent or *block* to prevent a matching intent from being propagated, and the rule type requires specification as *activity, broadcast* or *service*. Although the *Enhanced intent firewall* knows the type *all* which is used in rules

to prevent a package from sending or receiving global intents, this type is reserved for system-generated rules, as discussed below. A major difference from the rules used in the original *IFW* regards how filters are applied to intents. As discuses in 3.9, *IFW* rules must match an intent filter or destination component filter before performing other filter evaluation. To allow a wider combination of filters in which single intent values, including the destination component name, can be filtered during a single matching phase, *EFW* rules retain only a single filter, in which all other filter types can be nested. The filter types which can be specified in *EFW* rules are listed below.

```
{
    rules:[
        {
            ruletype: activity | broadcast | service
            rulelevel: block | log
            filter: {}
        }
    ]
}
```

Figure 4.3: General setup of a firewall rule

Logical filter The logical filter types *and-filter*, *or-filter*, *not-filter* are used the same as in the original implementation to concatenate the results of nested filter (see Figure 4.4).

```
{
    filtertype: and | or
    filter: []
}

{
    filtertype: not
    filter: {}
}
```

Figure 4.4: Filter types and, not, or concatenate nested filter

String filter The *string-filter* was extended to match sender package names the same as other intent data fields are handled. After removing the intent filter as filter type, the *string-filter* becomes the main intent value filter since it allows matching single intent data properties independent from each other and facilitates different comparing methods such as patterns and regular expressions (see Figure 4.5).

```
{
    filtertype: equals | contains | pattern | regex
    field: senderpackage | receiverpackage | action | packagecomponent |
           component | category | mimetype | data | scheme | ssp | host | path
    value: filtervalue
}

{
    filtertype: exists
    field: senderpackage | receiverpackage | action | packagecomponent |
           component | category | mimetype | data | scheme | ssp | host | path
    value: true | false
}
```

Figure 4.5: Filter type string

Port filter To check for the existence of a port value in intents the port filter can match specific port value ranges as well as intents with no port value (see Figure 4.6).

```
{
    filtertype: port
    equals: filtervalue
}

{
    filtertype: port
    min: filtervalue
    max: filtervalue
}
```

Figure 4.6: Filter type string

Disbanded filter types

The type *category-filter* was decommissioned since its functionality is now provided by the *string-filter*. Furthermore, the *sender-filter* and the *permission-filter* were discarded since they rely on internal functions of the system's permission system which are not accessible without causing technical side effects.

System-generated rules

The rules generated by the *EFW* itself regard a special type since they do not specify a particular intent type but completely block all intent traffic of a package. These rules are activated and deactivated by the firewall depending whether the package in question is currently installed on the monitored device. A system-generated rule can be deleted by the user but can not be manually altered or deactivated.

4.1.4 Intent processing

As discussed above, each time an intent reaches the Android intent firewall, the method hooks relay the passed intents and other parameters to the *Enhanced intent firewall* where they are processed in five phases.

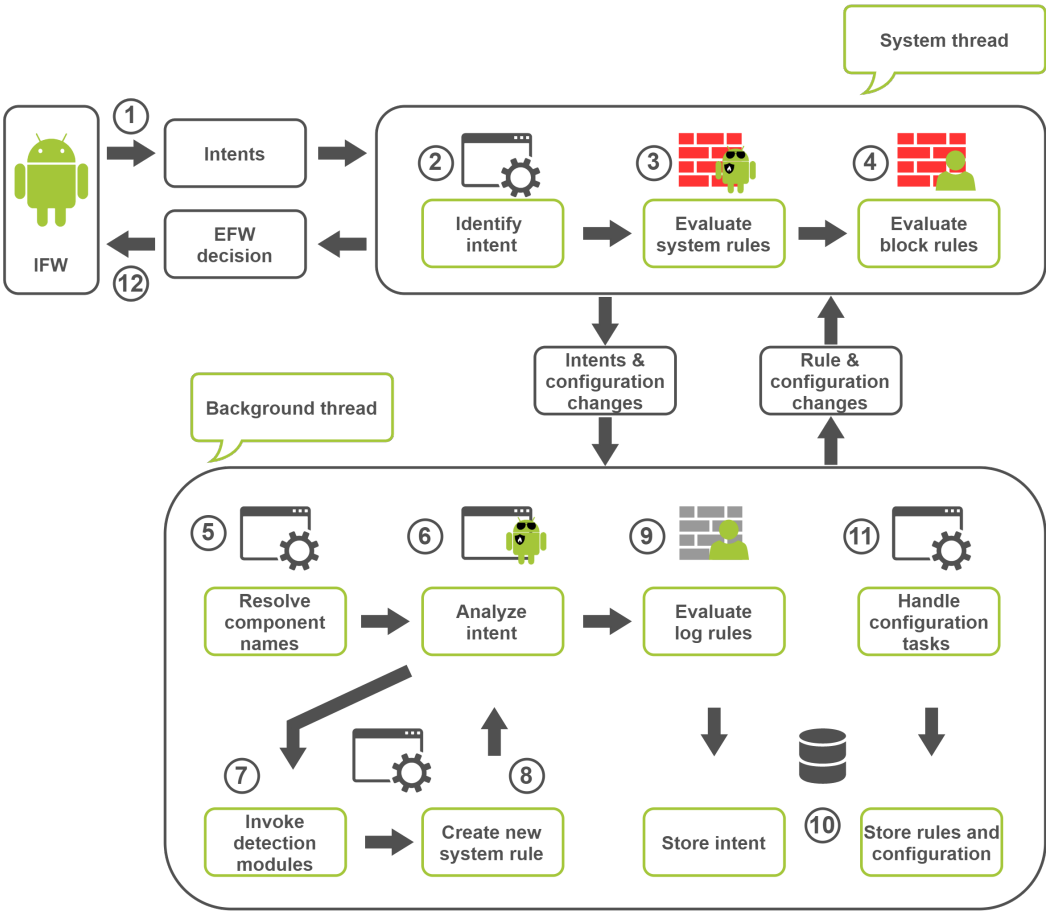

Figure 4.7: Enhanced intent firewall detail

Phase 1 - Intent identification

After receiving an intent, ① in Figure 4.7, the *Enhanced intent firewall* assesses how the intent must be handled depending on the type of the intent and the current firewall-settings ②. For the type activity and service the decision to process the intent is solely based on whether the firewall is currently running, in which case the intent and its respective meta data are culled from the passed function parameters and handed to the next phase of the evaluation process. The activity and service intents started by the *EFW* package are permitted without further processing. Broadcast intents require a more sophisticated evaluation since the *Enhanced intent firewall* application uses them to communicate between the user interface and the backend running in different processes

as well as to trigger the one-time initialization of the firewall after a system reboot. As long as the firewall has not been initialized, each broadcast intent is checked for the *BOOT_COMPLETED* action, which triggers the initialization process. Afterwards, each intent is handed to the next processing phase while the firewall is active unless the sender or receiver of the intent is the *EFW* application. When the intent is a command from the firewalls user interface, it is handed directly to the *EFW* service ⑪, where it is processed, while otherwise the intent's sender is vetted for security reasons, as discussed below.

Phase 2 - System rule evaluation
The first set of rules applied to incoming intents are created by the *EFW* application after classifying a package as malicious due to its intent usage. For this purpose, the firewall service retains a list of package *UIDs* and rules associated with the package to determine whether the intent in question should be blocked ③, before the intent and matching rule are handed to *phase 4*.

Phase 3 - User rule evaluation
If the intent was not blocked in the previous phase, user-defined block rules are applied corresponding to the intent type ④. Once an intent is matched by a rule, the remaining rules are not further evaluated and the intent is stopped from propagating, and if no rule matches the intent, it is allowed to be delivered to its destination. In either case, the intent along any matching rules are passed to the next phase.

Phase 4 - Background processing
The time critical phase of the evaluation is completed and intents entering this phase have been blocked or propagated to their destination components ⑫. All further processing is done in a separate thread to allow the *EFW* service to receive the next intent passed to the *IFW* interface. Before the evaluation progresses, depending on the intent's type, the package names of both sender and receiver must be determined from the *UIDs* of the sending and receiving components ⑤. If the intent was blocked during *phases 2* or *3*, it is handed directly to *phase 6* and is otherwise screened for signs of malicious behavior in *phase 5*.

Phase 5 - Detection of malicious intents
The analyzer of the *EFW* is modularly designed to allow registering separate detection algorithms. When receiving an intent, the analyzer checks which detection module is active ⑥ and sequentially passes the intent to each ⑦. The intent analysis and creation of firewall rules are handled inside of each module ⑧. The findings of the intent analysis described in Chapter 5 discuss also the implementation of such a detection module 5.6.

Phase 6 - Log Rule evaluation
In a final step the intent is matched with user-specified logging rules ⑨, before storage in the *SQLite* database ⑩.

Figure 4.11 shows the settings activity of the *EFW* application, which allows controlling the status of the firewall, while the Figures 4.12 and 4.13 list the firewall rules and details of a selected rule. When a rule is triggered by an intent, a notification alerts the user as

shown in Figure 4.14, and the details of the event are shown in the summary log, Figure 4.15.

4.1.5 Communication and security

The communication between the user interface running in the application context and the *EFW* backend service running in the system context depends on the same *IPC* functionality monitored by the firewall. To avoid unnecessary intent traffic, the firewall backend retains the status of the user interface, which sends an intent when opened or closed. With the sign-in intent, the user interface triggers a single push update, which supplies the stateless frontend with information regarding stored firewall rules, currently active settings and summarized occurred events. These events include triggered firewall rules; system events such as errors; findings of suspicious intents; and the subsequent automatic creation of firewall rules. While the user interface is active, the backend pushes the summarized information to the frontend in 30-second intervals. These data are displayed as a notification while the user interface is closed. To prevent other applications from eavesdropping on the intent communication between the front- and backends or from injecting intents into the *EFW* application, all broadcast intents started by the firewalls user interface are discarded after being consummated by the firewall backend in the function hook responsible for handling broadcast intents. Intents targeting the *EFW* package are only propagated if the sender is a system package, such as intents started by the firewall backend service.

4.2 Intent collector

Learning which uses of inter-process communication are dangerous requires observing both malicious and benign applications during runtime. The malware sample set of *Argus lab* [67] was analyzed for this purpose along with varied applications from the *Google Playstore*. The above *EFW* implementation was altered to collect the intent traffic of the respective sample applications, and to automate the sampling process, an additional application was created to orchestrate the data collection and perform optimization and post-processing on the sampled data.

4.2.1 General architecture

The *Intent collector* application is based on the *EFW* implementation and thus shares the same architecture. Instead of hooking all methods listed in 4.1, the *Intent collector* only hooks the *checkBroadcast* method to facilitate communication between the application's front- and backend. To monitor the global intent traffic, the *Intent collector* hooks various methods of the system and instead of a user interface, the application declares two empty activity components, which start or stop the sampling process when launched.

4.2.2 Hooked classes for intent sampling

Unlike the function hooks used in the *EFW* implementation, where the custom code is executed instead of the original method's code, the *Intent collector* application executes the original code of the hooked methods after the extraction of the parameters passed to the function call. This approach enables accessing the launched intents without altering the functionality of the *IPC* system or disrupting the intent traffic. As discussed in 3.8, global intent communication includes a resolution process in which the system seeks components suitable for intent delivery. For explicit intents, the respective target component must exist on the device when the intent is started, while the resolution process of implicit intents is more complex. In both cases, the intent is discarded in the absence of a suitable target component. Challenges arose when choosing a point in the resolution process at which the intents are being sampled. Sampling intents at the beginning of the resolution process allows capturing all sent intents; however, there is an absence of data regarding which components would have received an implicit intent. Performing the sampling process at the end of the resolving process this data would be accessible at the disadvantage that intents without a suitable receiver would have been discarded already. The latter option can be accomplished by accessing the intents at the *IFW's* interface similar to the *EFW* implementation. Since this option would yield only a fraction of the total volume of started intents for sampling, the earliest point in the resolution process was chosen instead. Table 4.8 depicts the methods and declaring classes which were hooked to siphon the launched intents. All methods were hooked before any of the native code was executed to ensure none of the parameter objects were changed. After sampling the data, the methods were allowed to continue with the original values.

Class	Method
com.android.server.am.ActivityStackSupervisor	startActivityLocked
com.android.server.am.ActivityManagerService	broadcastIntentLocked
com.android.server.am.ActiveServices	retrieveServiceLocked

Figure 4.8: Hooked methods of the system

4.2.3 Data collection process

Starting the *Intent collector* application launches the main activity component, which signals the backend to start the sampling process (see ① in Figure 4.9). When the hooked methods are invoked, the respective parameter values are collected and passed to the background thread ②. Unlike the *EFW* implementation, no rule evaluation or other processing of the intents are performed, and the data collection ceases once the respective activity is launched ③.

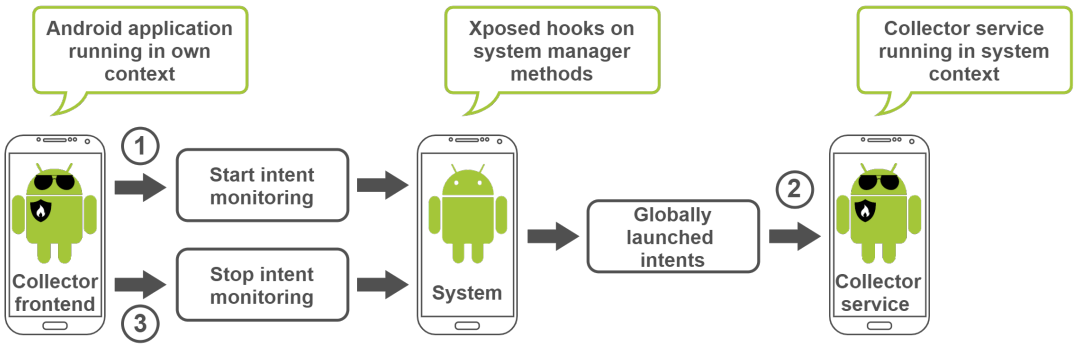

Figure 4.9: Intent collector setup

4.3 Data collection pipeline

To automate the data collection and orchestrate the tools such as the *Intent collector* application, a data collection pipeline was implemented in Java SE 8, which executes each sampling stage on every sample file, as outlined in Figure 4.10, to compile a data set containing the sample's *IPC* traffic for further analysis.

4.3.1 Virtual device preparation

To perform the data collection in a controlled environment, a virtual machine running Android 5.1 was used to execute the sample applications. *Genymotion Android emulator for desktop* [68], which relies on *Oracle VM VirtualBox*, was chosen for this task since it provides a command line interface which allows programmatical interaction with the created virtual devices. To ensure that each sampling iteration was performed under the same conditions, a template virtual device was created and was used to clone a new instance for each data collection procedure. The template virtual machine was prepared by installing the *Genymotion ARM translation patch (version 1.1)*, which makes the *x86*-based *Genymotion emulator* compatible with applications including *ARM* code. The *Xposed* installation was then performed by installing *SuperSU root (version 2.46)* [69], the *Xposed installer apk (version 3.1.1)* and the *Xposed framework (version 89)* [54]. Finally, the *Intent collector* application was installed and activated as a *Xposed module* on the virtual device.

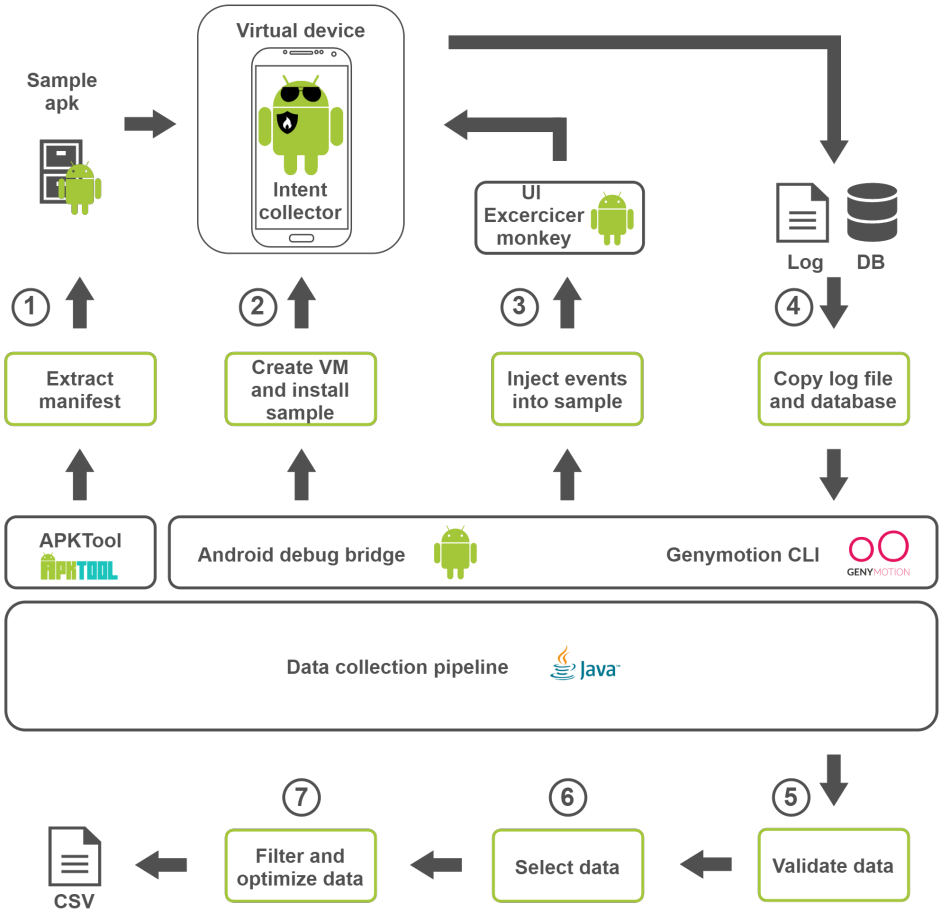

Figure 4.10: Data collection pipeline

4.3.2 Stage 1 - Sample preparation

As described in 5, all application samples were present in the form of a single apk file. To allow the *Data collection pipeline* to ingest the samples in a structured manner, the apk files required placement in a fixed folder structure. Each malware family was placed in a designated directory with separate folders for each malware variety, which in turn contained separate folder for each sample apk. Samples obtained from the *Google Playstore* are similarly organized with each sample placed in its designated folder which was placed in a directory representing the application's *Playstore* category. The pipeline application then performed the sampling process for each sample folder, and the respective apk file required disassembly to allow ample analysis. This step was performed by employing the *ApkTool (version 2.4.0)* [70] library which allows to disassembling and rebuilding Android binary files (see ① in Figure 4.10). The option to disassemble the source code was omitted since *Apktool* allows specifying which parts of the binary should be unpacked. This decision aimed to reduce disassembling effort since the sample's source

code was not needed for analysis. The output of the reverse engineering process was piped into a temporary directory inside the respective sample folder. After disassembly, the pipeline checked whether the *AndroidManifest.xml* file of the sample was successfully extracted; if not, the pipeline retried the extraction before skipping processing the sample. With the manifest file successfully obtained, the file was copied into the sample directory directly and the temporary unpacking directory was deleted. The manifest file was then parsed and the package name extracted and stored in a text file created inside the sample directory. Afterwards, the pipeline ensured that each successfully processed sample folder contained the files required for further analysis: the sample apk, extracted manifest and file containing the package name.

4.3.3 Stage 2 - Data collection

After preparing a sample, the data collection was performed using the *Genymotion* command line interface and Android's *adb* tool. Each process began with a timeout value, which automatically terminated the process after the respective time was elapsed if the process fails to finish on its own. After terminating a command, the return code was checked to ensure that each step was performed before moving to the next. The processing of the sample was aborted if an error occurred following a command execution. The return values of the execution processes were logged for later analysis. The first step of the stage regarded retrieval of the sample's package name from the associated *packagename.txt* file. Before creating a new instance of a virtual device, the pipeline ensured that all instances of previous sampling iteration were destroyed, and afterwards the pipeline created a new copy of the device template VM and started the new virtual device. Following the device's boot process, the data collection commenced using Android's *debug bridge CLI* to launch the *Intent collector* application. The same process was used to install and launch the sample application using the package name retrieved from the associated *packagename.txt* file (see ② in Figure 4.10). Android's UI Exerciser Monkey was employed to simulate interaction with the sample application to tailor the simulated events generated and injected into the target application ③. To ensure all samples received the identical order and type of events, the seed value *1535075680546* was randomly created by the *UI Exerciser Monkey* during the initial sampling process and was used for subsequent iterations. The distribution of event types was also fixed with 15% each for touch, motion, trackball and basic navigation events, as well as 20% each for general navigation and activity launch events. A 1,000 ms delay between each of the 3,000 injected events allowed the application to respond to the event. Furthermore, the option to terminate *UI Exerciser Monkey* was activated for cases when the sample application suffered an unrecoverable crash during the process. After the *UI Exerciser Monkey* finished injecting events or the process was terminated after encountering a timeout, the pipeline closed the sample application and its running processes. To signal the *Intent collector* to stop recording *IPC* traffic and close the database connection and log files, the respective activity was triggered via an *adb* command. Finally, after the *Intent collector* was closed, the database and log file were pulled from the virtual device and copied into the respective sample folder, ④. Following each data collection iteration,

the used instance of the virtual device was terminated and deleted.

4.3.4 Stage 3 - Data validation

To determine the success of the data collection process on a particular sample, the respective log file and database were analyzed, (see ⑤ in Figure 4.10). The sample was discarded if the log file or database were unsuccessfully pulled from the virtual device. Otherwise, the log file was parsed to check whether any sampling stages produced errors; if so, the sample was omitted from further analysis; if not, the number of successful injected events into the sample application was calculated. Samples with less than 2,000 successful injected events were discarded as well. Lastly, the database was analyzed, to ensure that the sample application was involved in any inter-process communication. If no intents were started or received by the application, the sample was excluded from further analysis.

4.3.5 Stage 4 - Data selection

The pipeline steps *Sample preparation*, *Data collection* and *Data validation* were performed on batches of ten malware samples and three *Playstore* samples respectively. The maximum number available was used in cases when fewer then ten samples were available for a certain malware variety. The number of processed samples for a certain variety was calculated after each batch, and if at least three fulfilled the requirements described above, the sampling of the respective malware variety or *Playstore* category was considered finished. In the absence of available samples for a particular malware variety, the maximum number of successful samples were chosen for further analysis. This was also employed when the threshold of 50 analyzed samples for a variety was passed without yielding at least 3 successfully processed samples, and if more than three samples of a category were available, three were chosen at random (see ⑥ in Figure 4.10). The number of sampled applications and of those which are successfully and unsuccessfully processed are listed in 5.1.1.

4.3.6 Stage 5 - Data optimization

Before analysis, the collected intents required separation from noise data, since only *IPC* traffic involving a sample application was relevant for the analysis. Furthermore, since the data collected in each sampling iteration were stored in a separate database and therefore unfeasible for efficient analysis, the data required merging into a single database each for malware and *Playstore* samples. During this process, the sampled data were optimized by connecting the intent destination component names with the associated package names for more efficient querying. After preparing the cleaned and optimized sample data, a final step sorted and organized the data by querying two sample databases to separate the IPC traffic regarding the type of intents as well as sender and receiver packages. The data returned by these queries were dumped into *CSV* data files, which presented the basis for further analysis (see ⑦ in Figure 4.10).

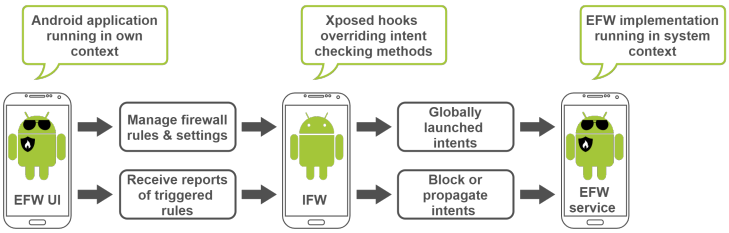

Figure 4.11: Controls of the firewall applications

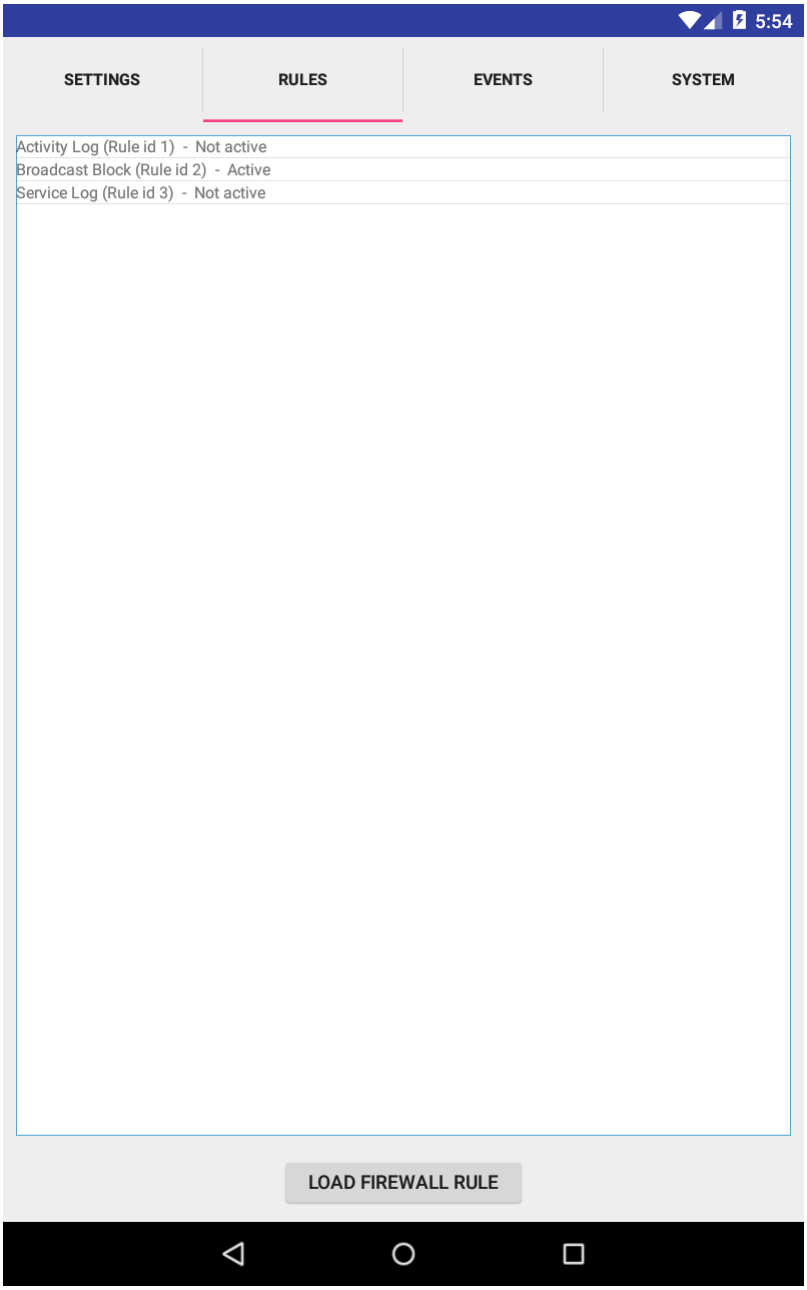

Figure 4.12: Overview of available firewall rules

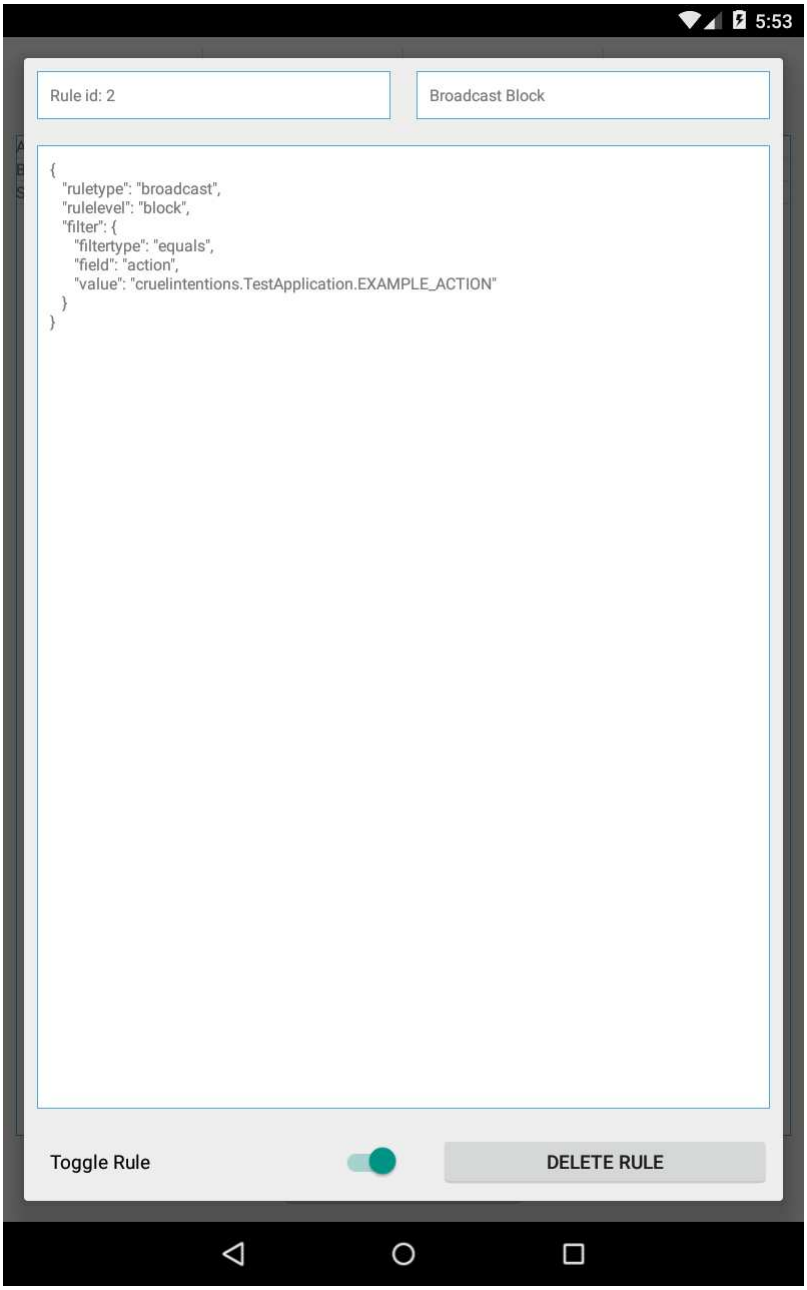

Figure 4.13: Detail view of a firewall rule

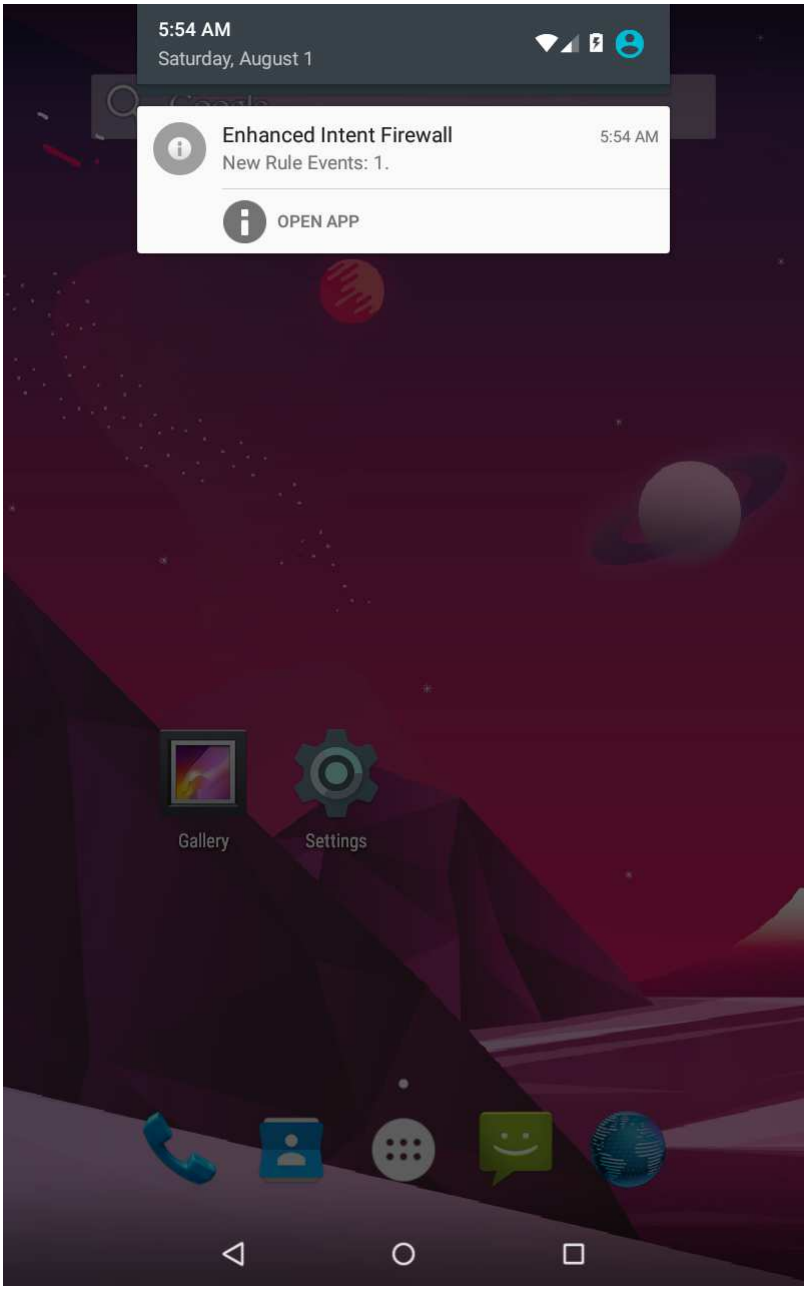

Figure 4.14: Notification of a rule event

Figure 4.15: Detail view of a rule event

CHAPTER 5

Evaluation

Analyzing a sufficiently large collection of recent malware samples was needed to learn which IPC calls are performed by malware on the Android platform, and reference samples of non-malicious applications also required evaluation to determine which intent traffic should be considered malicious. These two datasets were similarly processed and analyzed as described in 4.3 draw meaningful conclusions from the analysis.

A set of malicious applications, the *Android Malware Dataset*, was shared by the *Argus Cyber Security Lab* [71] for this project and consists of 24,553 samples which have been studied and categorized into 135 different varieties of 71 malware families [72]. The set contains benign applications altered to contain malicious components as well as standalone malware. Figure 5.1 shows the number of samples available for each malware and the type of malicious payload characterizing the respective family. Comparable analysis data of a reference dataset was needed to interpret the outcome of the malware analysis. Because every uploaded apk is screened for malware signatures before becoming downloadable in the official *Google Playstore*, it can be assumed that the applications in this market place do not contain malware. [73] estimated the prevalence of infected applications in the *Google Playstore* at 0.02 percent. Samples were randomly selected for analysis from 30 application categories, as listed in Figure 5.3.

Malware family	Type	# Varieties	# Samples
Airpush	Adware	1	7843
AndroRAT	Backdoor	1	46
Andup	Adware	1	45
Aples	Ransom	1	21
BankBot	Trojan-Banker	8	648
Bankun	Trojan-Banker	4	70
Boqx	Trojan-Dropper	2	215
Boxer	Trojan-SMS	1	44
Cova	Trojan-SMS	2	17
Dowgin	Adware	1	3385
DroidKungFu	Backdoor	6	546
Erop	Trojan-SMS	1	46
FakeAngry	Backdoor	2	10
FakeAV	Trojan	1	5
FakeDoc	Trojan	1	21
FakeInst	Trojan-SMS	5	2172
FakePlayer	Trojan-SMS	2	21
FakeTimer	Trojan	2	12
FakeUpdates	Trojan	1	5
Finspy	Trojan-Spy	1	9
Fjcon	Backdoor	1	16
Fobus	Backdoor	1	4
Fusob	Ransom	2	1277
GingerMaster	Backdoor	7	128
GoldDream	Backdoor	2	53
Gorpo	Trojan-Dropper	1	36
Gumen	Trojan-SMS	1	145
Jisut	Ransom	1	560
Kemoge	Trojan-Dropper	1	15
Koler	Ransom	2	69
Ksapp	Trojan	1	36
Kuguo	Adware	1	1199
Kyview	Adware	1	175
Leech	Trojan-SMS	1	128
Lnk	Trojan	1	5
Lotoor	HackerTool	15	329

Figure 5.1: Argus Lab malware samples - part 1

Malware family	Type	# Varieties	# Samples
Mecor	Trojan-Spy	1	1820
Minimob	Adware	1	203
Mmarketpay	Trojan	1	14
MobileTX	Trojan	1	17
Mseg	Trojan	1	235
Mtk	Trojan	3	67
Nandrobox	Trojan	2	76
Obad	Backdoor	1	9
Ogel	Trojan-SMS	1	6
Opfake	Trojan-SMS	2	10
Penetho	HackerTool	1	18
Ramnit	Trojan-Dropper	1	8
Roop	Ransom	1	48
RuMMS	Trojan-SMS	4	402
SimpleLocker	Ransom	4	173
SlemBunk	Trojan-Banker	4	174
SmsKey	Trojan-SMS	2	165
SmsZombie	Trojan-SMS	1	9
Spambot	Backdoor	1	15
SpyBubble	Trojan-SMS	1	10
Stealer	Trojan-SMS	1	25
Steek	Trojan-Clicker	1	12
Svpeng	Trojan-Banker	1	13
Tesbo	Trojan-SMS	1	2
Triada	Backdoor	1	210
Univert	Backdoor	1	10
UpdtKiller	Trojan	1	24
Utchi	Adware	1	12
Vidro	Trojan-SMS	1	23
VikingHorde	Trojan-Dropper	1	7
Vmvol	Trojan-Spy	1	13
Winge	Trojan-Clicker	1	19
Youmi	Adware	1	1301
Zitmo	Trojan-Banker	2	24
Ztorg	Trojan-Dropper	1	20

Figure 5.2: Argus Lab malware samples - part 2

Google Playstore category
Art and Design
Beauty
Books
Car
Comics
Customizing
Dating
Efficency
Entertainment
Finance
Food
Health
House and garden
Learning
Lifestyle
Map
Medicine
Messaging
Music
News
Office
Parents
Photography
Shopping
Social
Sport
Tickets and Events
Tools
Travel
Videoplayer
Wetter

Figure 5.3: Google Playstore sample categories

5.1 Results of data collection

Following the process discussed in 4.3, both sets of applications were sampled to obtain sufficient data for analysis. As shown in Figure 5.4, of the 24,553 malware samples available, 1,544 were processed to obtain data from at least one sample of each malware variety, with 39.89% of the sampling procedures resulting in a positive outcome, yielding data from 616 samples, covering 114 (84.44%) of the different varieties and 61 (85.91%) of the different malware families respectively. The threshold of 3 samples was reached, with 14 varieties yielding 2 and 10 varieties yielding 1 successfully sampled apk, as listed in Figure 5.5. No usable data could be obtained for 21 malware varieties and 10 malware families. As shown in Figure 5.6, processing 155 samples from the benign *Playstore* set yielded 97 successfully sampled applications, providing data from at least three samples of each *Playstore* category. Figure 5.7 lists the errors occurring during the sampling of the unsuccessful processed 928 malware and 58 benign samples. Errors within the sampled application represented the main reason for unsuccessful sampling attempts and accounted for 64.66% and 65.62% of all occurred errors, followed by unsuccessful installation attempts of the sample package on the virtual device with 17.56% and 20.69% respectively. In 4.2% and 13.79% cases the *UI Exerciser Monkey* reported an error while interacting with the sample application while incorrect activity names were specified by the sample in 6.03%. Furthermore, all samples were discarded if less than 2,000 events were injected (2.05%) or if no *IPC* traffic involving the sample application could be monitored (4.31%). In 11 cases (1.19%), the *Intent Collector* application did not behave correctly. From the 616 malicious and 97 benign samples successfully processed, 308 and 90 respectively were randomly selected for analysis.

Status	# Samples		# Varieties		# Families	
All samples	24553	100.00%	135	100.00%	71	100.00%
Processed Samples	1544	6.28%	135	100.00%	71	100.00%
Unsuccessfull processed *	928	60.11%	21	15.56%	10	14.09%
Successfull processed *	616	39.89%	114	84.44%	61	85.91%
Detailed analysis **	308	50%	114	100.00%	61	100.00%

* of processed samples
** of successfull processed samples

Figure 5.4: Processed malware samples

Successful processed	# Samples		# Varieties	
One sample per variety	10	1.62%	10	8.77%
Two samples per variety	28	4.54%	14	12.28%
Three or more samples per variety	578	93.83%	90	78.94%
Total	616	100.00%	114	100.00%

Figure 5.5: Successful processed malware samples per variety

Status	# Samples		# Categories	
All samples	N.A.	100.00%	35	100.00%
Processed samples	155	N.A.	30	85.71%
Unsuccessfull processed *	58	37.41%	30	100.00%
Successfull processed *	97	62.59%	30	100.00%
Detailed analysis **	90	92.78%	30	100.00%

* of processed samples
** of successfull processed samples

Figure 5.6: Processed Playstore samples

Error during processing	Malware samples		Play store samples	
Error installing sample	163	17.56%	12	20.69%
Less than 2000 events injected into sample	19	2.05%		
Error in analysis application	11	1.19%		
Error in sample application	600	64.66%	38	65.52%
No activities found in sample application	56	6.03%		
No intents of sample application collected	40	4.31%		
Unable to connect to activity manager	39	4.20%	8	13.79%
Total	928	100%	58	100%

Figure 5.7: Errors during sampling

5.1.1 Collected intents

The 308 malware samples chosen for detailed analysis include 392,343 globally sent intents which were started or received by one of the sample applications. For the benign samples, the overall number of captured intents is 19593, as shown in Figure 5.8. In both sample sets, the majority regards activity intents with 353,933 (90.21%) and 16,561 (85.53%) respectively. In the malicious dataset, broadcast intents constitute the second greatest portion with 37,041 (9.44%), which for the benign dataset regards service intents with 2,608 (13.31%). The smallest fraction of intents in the malicious sample set are service-related intents with 0.35% or 1,369 in total and broadcast intents with 424 (2.16%) for the *Playstore* set. As discussed in 4.3, the intents were grouped based on their type, origin and destination components for the structured analysis of the sampled data.

Intent type	Malware samples		Playstore samples	
Activity	353933	90.21%	16561	84.53%
Broadcast	37041	9.44%	424	2.16%
Service	1369	0.35%	2608	13.31%
Total	392343	100%	19593	100%

Figure 5.8: Overview of sampled intents

Activity intents

The captured activity intents were divided into six types as listed in 5.9, with the first group labeled A1 containing intents explicitly started by a sample to launch an activity of a system package. Groups A2 and A4 include intents sent by a sample application to a third party package or its own. Intents in the group A3 were also launched by a sample package, but these intents are not explicit and therefore do not specify a target component. Since the intents were sampled before the resolving process, the actual destination component of the intents in this category are unknown, as described in 4.2.2 The groups A5 and A6 contain intents received by a sample application from a system package or the *UI Exerciser Monkey*. As shown in Figure 5.9, malicious samples primarily sent intents to system packages, which accounted for 71.49% of all activity intents, followed by intents sent to their own package (16.58%). The third sizable portion (11.7%) of the activity intent traffic includes intents received by the sample from unresolved packages, and the remaining 0.21% *IPC* traffic comprises intents from categories A2, A3 and A5. For the benign *Playstore* sample set, the largest number of intents (87.32%) were those generated by the *UI Exerciser Monkey* during sampling, while the intents received from the system was comparably low at 0.02%. The largest group of started intents are those sent to their own application package, while intents sent to other packages comprise 2.2% of all sampled intents.

Intent category		Malware samples		Playstore samples	
A1	Sample package to system package	253032	71.491%	70	0.423%
A2	Sample package to third party package	727	0.205%	269	1.624%
A3	Sample package to unresolved package	41	0.012%	29	0.175%
A4	Sample package to same sample package	58713	16.589%	1727	10.428%
A5	System package to sample package	2	0.001%	4	0.024%
A6	UI Exerciser Monkey to sample package	41418	11.702%	14462	87.326%
	Total	353933	100%	16561	100%

Figure 5.9: Distribution of sampled activity intents

Broadcast intents

As shown in Figure 5.10, implicit broadcast intents with unknown receiver packages are used by both malicious and benign sample sets. Grouped as category B1, this type of intent accounts for 95.58% of all broadcasts used by malicious samples and 58.9% for benign ones. The remaining sent broadcast intents were explicit and aimed at the same package which started them, amounting to 4.42% and 44.1%.

Intent category		Malware samples		Playstore samples	
B1	Sample package to unresolved package	35403	95.58%	237	55.9%
B2	Sample package to same sample package	1638	4.42%	187	44.1%
	Total	37041	100%	424	100%

Figure 5.10: Distribution of sampled broadcast intents

Service intents

For the malware and *Playstore* sample sets, service intents aimed to start or interact with a component declared in their own package, representing 90.43% and 96.97% of all service intents (see Figure 5.11), followed by implicit intents targeting an unresolved service component, with 9.42% and 2.15% respectively. Lastly, 0.15% and 0.88% of the intents targeted service components declared in third-party applications.

Intent category		Malware samples		Playstore samples	
S1	Sample package to third party package	2	0.15%	23	0.88%
S2	Sample package to unresolved package	129	9.42%	56	2.15%
S3	Sample package to same sample package	1238	90.43%	2529	96.97%
	Total	1369	100%	2608	100%

Figure 5.11: Distribution of sampled service intents

5.2 Evaluation of activity intents

This section discusses activity-related intents, describing their composition and function while assessing their maliciousness.

5.2.1 Intents sent to system packages

ID	Intent	Malware		Playstore	
A1_1	A: android.intent.action.CHOOSER C: com.android.internal.app.ChooserActivity	24	0.009%	41	58.6%
A1_2	A: android.intent.action.PICK_ACTIVITY C: com.android.settings.ActivityPicker	2	0.0007%		
A1_3	A: android.intent.action.VIEW C: com.android.internal.app.ResolverActivity			1	1,4%
A1_4	A: android.intent.action.MAIN C: com.android.settings.WirelessSettings	3	0.0011%		
A1_5	A: android.settings.SETTINGS C: com.android.settings.Settings	3	0.0011%	14	20%
A1_6	A: android.settings.APPLICATION_DETAILS_SETTINGS C: com.android.settings.applications.InstalledAppDetails	19	0.0075%	14	20%
A1_7	A: android.settings.MANAGE_APPLICATIONS_SETTINGS C: com.android.settings.Settings$ManageApplicationsActivity	2	0.0007%		
A1_8	A: android.settings.SECURITY_SETTINGS C: com.android.settings.Settings$SecuritySettingsActivity	2	0.0007%		
A1_9	A: android.app.action.ADD_DEVICE_ADMIN C: com.android.settings.DeviceAdminAdd	252977	99.978%		
	Total	353932	100%	70	100%

Figure 5.12: Intents starting a system package activity

App chooser dialog
The intents A1_1 to A1_3 each another intent as a payload to be opened by a fitting activity and to allow the user to choose this activity, the wrapping intent creates a chooser dialog to display all activities able to handle the payload intent. The chooser dialog created by intent A1_1 starts the selected activity, and then the dialog created by intent A1_2 passes the chosen activity class to the sender of the wrapping intent to allow the component to react to the user's choice. Intent A1_3 shows a selection activity which behaves similarly to A1_1, although this approach allows the setting of an preferred activity to handle the wrapped intent in future launches.

System settings
The intents A1_4 to A1_8 launch an activity associated with the system settings. All the intents open a specified activity component, while A1_8 also requires passing the name of an installed package and a data URI to show the detailed settings for this specific package.

Device Admin
Applications may use the operating system's device administration *API* to enforce security policies on a device, which grant control over system-level features such as password-related settings, data encryption or the remote installation of applications. Before the user can exert this control, they must grant the application permission to do so. With the intent A1_9, the respective settings activity is shown which asks the user to enable an application as device administrator. The package name of the application in question must be passed with the intent, and an optional explanation can be passed to give the user additional information regarding the request. As shown in Figure 5.13, the distribution of the quantity in which this intent is employed by sample applications greatly varies, with some packages launching several thousand of this particular intent while most samples do not employ this intent at all. Since the rapid consecutive opening of the admin request activity creates a screen lock, the application aims to coerce the user to grant the requested administration privileges. In addition, the message passed with the intent informs the user that granting administrative privileges is necessary for the application to properly function. The numbers listed in Figure 5.14 indicate that the launch of a single intent is not triggered by user interaction but by an automated procedure started at application launch, confirming the assessment of [72] for this type of malware.

Malware	Variety	Sample 1	Sample 2	Sample 3
BankBot	5	9972	9808	9976
	6	9969	9802	9332
	7	9332	9366	9275
	8	9808	9687	
Koler	1	6443	6472	6456
Obad	1	1929		
RuMMS	1	9353	9183	9321
	2	9332	9235	9042
	3	8606	8223	7609
	4	9320	9147	9312
SimpleLocker	2	2911	2685	3032
SlemBunk	2	7763		

Figure 5.13: Device administration requests per sample

# Intents	# Samples
0	259
1 - 20	13
50 - 100	3
100 - 200	2
> 1900	31

Figure 5.14: Distribution of samples using the device administration request

5.2.2 Intents sent to third party packages

ID	Intent	Malware		Playstore	
A2_1	A: android.intent.action.VIEW C: com.android.browser.BrowserActivity	382	52.54%	199	73.97%
A2_2	A: android.intent.action.VIEW C: com.android.contacts.activities.PeopleActivity	2	0.27%		
A2_3	A: android.intent.action.VIEW C: com.android.packageinstaller.PackageInstallerActivity	85	11.69%		
A2_4	A: android.intent.action. DELETE C: com.android.packageinstaller.UninstallerActivity	8	1.1%		
A2_5	A: android.intent.action.DIAL C: com.android.dialer.DialtactsActivity	55	7.56%		
A2_6	A: android.intent.action.GET_CONTENT C: com.android.documentsui.DocumentsActivity	1	0.13%	10	3.71%
A2_7	A: android.intent.action.INSERT C: com.android.contacts.activities.ContactEditorActivity	1	0.13%		
A2_8	A: android.intent.action.INSERT_OR_EDIT C: com.android.contacts.activities.ContactSelectionActivity	1	0.13%		
A2_9	A: android.intent.action.MAIN C: com.android.browser.BrowserActivity	11	1.51%		
A2_10	A: android.intent.action.MAIN C: com.android.dialer.DialtactsActivity	19	2.61%		
A2_11	A: android.intent.action.MAIN C: com.android.calendar.AllInOneActivity	2	0.27%		
A2_12	A: android.intent.action.MAIN C: com.android.development.Development	1	0.13%		
A2_13	A: android.intent.action.MAIN C: de.robv.android.xposed.installer.WelcomeActivity	2	0.27%		
A2_14	A: android.intent.action.MAIN C: com.android.customlocale2.CustomLocaleActivity	2	0.27%		
A2_15	A: android.intent.action.PICK C: com.android.contacts.activities.ContactSelectionActivity	17	2.33%		
A2_16	A: android.intent.action.PICK C: com.android.gallery3d.app.GalleryActivity			1	0.37%
A2_17	A: android.intent.action.RINGTONE_PICKER C: com.android.providers.media.RingtonePickerActivity			2	0.74%

Figure 5.15: Intents starting a third party package activity

ID	Intent	Malware		Playstore	
A2_18	A: android.intent.action.SET_WALLPAPER C: com.android.launcher3.LauncherWallpaperPickerActivity			1	0.37%
A2_19	A: android.service.wallpaper.CHANGE_LIVE_WALLPAPER C: com.android.wallpaper.livepicker.LiveWallpaperChange	1	0.13%		
A2_20	A: android.service.wallpaper.LIVE_WALLPAPER_CHOOSER C: com.android.wallpaper.livepicker.LiveWallpaperActivity	73	10.04%		
A2_21	A: android.intent.action.SENDTO C: com.android.mms.ui.ComposeMessageActivity	46	6.32%		
A2_22	A: android.search.action.GLOBAL_SEARCH C: com.android.quicksearchbox.SearchActivity	2	0.27%		
A2_23	A: android.media.action.IMAGE_CAPTURE C: com.android.camera.CameraActivity			51	18.95%
A2_24	A: android.speech.tts.engine.CHECK_TTS_DATA C: com.svox.pico.CheckVoiceData			1	0.37%
A2_25	A: android.speech.tts.engine.INSTALL_TTS_DATA C: com.svox.pico.DownloadVoiceData			4	1.48%
	Total	727	100%	269	100%

Figure 5.16: Intents starting a third party package activity

Web browser

Intent A2_1 specifies the action *VIEW* and web browser application as targets to display a webpage which is passed in the form of a data URI. This intent is common for the malware and *Playstore* sample sets and malicious applications use this intent in 342 of 382 instances to display advertising webpages, while for benign samples this is the case in only 25 usages, all of which use an official *Google* advertising *API*, unlike the malware samples. In 25 cases, malicious samples pass device information such as *phone number*, *IMEI* or *operating system version* as URL parameters to the contacted server. This information can be used to identify the device and further action by the malicious application, as it is instructed by the contacted control server [72].

Contact management

The intents in this category interact with the device's stored contacts. Similar to A2_1, intent A2_2 uses the action *VIEW* to display the specified component, where in this case the contact overview activity lists existing contacts. A2_8 displays the same list of contacts, and unlike *VIEW* which has no return value, the action *INSERT_OR_EDIT* will return the selected or newly created contact to the invoking component in the form of an URI. A2_7 similarly returns the created contact as data URI but directly opens the activity containing the contact creation form. Intent A2_15 employs the *PICK* action and a data URI, in this case identifying the list of stored contacts, to display a set of data from which the user may select an item, which again is returned as URI.

Package management

The intent A2_3 opens a dialog to ask the user for confirmation to remove an installed application package from the device. The name of the package must be passed via the intents data URI property. Similar to A2_3, the intent A2_4 opens a dialog to request confirmation from the user to make changes to the devices application packages. The dialog lists the name of the package supposed to be installed on the device and the permissions requested by the package, and the intent must pass the package name as data URI. The installation of a package regards an ordinary operation, but this intent is misused to trick the user into loading malicious payloads on the device or into installing additional unwanted applications by disguising the operation as an urgent update for the host application [72].

Dialing activity
To launch the devices dialer activity, the intent A2_5 can specify a data URI with the scheme *tel* to start the activity with the passed phone number preselected or without additional data to show the dialer in an empty state. Both options are used by the sample.

Choose image
Using an intent with the *GET_CONTENT* action as with A2_6 allows passing a mimetype of data, enabling the user to select a resource corresponding to this type, which is then returned to the calling component. A2_16 specifies the action *PICK* and an URI pointing to a data directory, which also returns the URI of the selected item. Both intents define that only resources of type *image* may be chosen.

Open application
The intents A2_9 to A2_14 use the action *MAIN* to open the specified activity as the main entry point of the respective application, meaning that no additional data are expected to be specified in the intent.

Choose ringtone
A2_17 opens a dialog listing all currently available ringtones on the device to allow the user to choose one, which returns no data but sets the selected resource as the active ringtone.

Choose wallpaper
While A2_18 and A2_19 show a gallery containing images suitable as background wallpaper, A2_20 shows a preview of a wallpaper passed with the intent, allowing the user to confirm the selection to set the currently active wallpaper.

Send message
Intent A2_21 opens the message composing activity of the installed *MMS* application and intent can carry data to preset the message content and receiver for the message.

Global search
A2_22 opens the global search provider activity and similar to A2_21 can carry additional data such as the search query.

Image capturing
Intent A2_23 launches the main activity of the installed camera application.

Text to speech
The intents A2_24 and A2_25 interact with Android's text-to-speech engine. The first checks the installation status of the required components to use the service, while the second opens the Playstore application to install the package.

5.2.3 Sample package to unresolved package

ID	Intent	Malware		Playstore	
A3_1	A: android.intent.action.VIEW	36	87.8%	29	100%
A3_2	A: android.intent.action.SENDTO	5	12.2%		
	Total	41	100%	29	100%

Figure 5.17: Implicit intents starting an activity of a unresolved package

Display data
The implicit intent A3_1 specifies the action *VIEW* and a data URI, which allows the system to resolve a suitable component to display the data. The *Playstore* sample set employs this intent to specify URIs pointing to the marketplace, the *Google Playstore* website and to *PDF* files on the device's drive. Although the malware samples refer to applications in the marketplace, the target applications differ, as the benign samples use this intent to link to their own page in 80% of the cases or to a *Google* or *Facebook* application in the remaining 20%, while the malware samples refer to applications unrelated to the domain of the sample (58%) and to online gambling applications (37%). Only 5% of the intents link to the sample package's own marketplace page.

Send message
Similar to A2_21, A3_2 uses the action *SEND_TO* to compose a message, and the system resolves the used application to send emails by defining the type *mailto*. This action allows passing additional properties defining the title, text and receiver of the message with the intent. In all five instances where the malware sample set uses this intent, the values for these three properties are specified by the application. While the passed string values are encrypted, the content of the message contains device-specific information such as the *device name* and *operating system version*.

5.2.4 Sample package to same sample package

ID	Intent	Malware		Playstore	
A4_1	A: android.intent.action.MAIN C: Custom component	147	0.25%	150	8.68%
A4_2	A: android.intent.action.VIEW C: Custom component	14109	24.03%	34	1.96%
A4_3	A: Custom action C: Custom component	11	0.01%	39	2.25%
A4_4	C: Custom component	44446	75.7%	1504	87.08%
	Total	58713	100%	1727	100%

Figure 5.18: Intents starting an activity of the own package

Open activity

The intents A4_1 to A4_4 open specified activities, which in all cases regard the same sample package which started the intents. The action property becomes optional since the component is explicitly defined, and since no intents contain a data URI, the action *VIEW* behaves like *MAIN* by launching the activity component without being expected to pass additional data. Any custom-defined action in A4_4 may impact the internal logic of the component, but has no influence on which activity is launched. 56758 (96.67%) of all intents used by the malware samples were sent by the nine applications listed in 5.19, which shows the number of this type of intent launched by each sample. The resulting activities notify the user of urgent updates requiring root privileges in order to be added to the system's device administrator list. Identical to the behavior of A1_9, the intent's rapid launch aims to coerce the user into granting requested privileges to regain control of the device. The remaining 1,955 intents launched by the malware samples were unsuspicious along with the 1,720 intents launched by the benign samples.

Malware	Variety	Sample 1	Sample 2	Sample 3
BankBot	3	3948	5000	5038
SlemBunk	1	7164		
	2	7183	7159	
	4	7092	7070	7104

Figure 5.19: Samples launching their own activities repeatedly

5.2.5 System package to sample package

ID	Intent	Malware		Playstore	
A5_1	A: android.intent.action.MAIN C: Custom component	1	50%		
A5_2	C: Custom component	1	50%	4	100%
	Total	2	100%	4	100%

Figure 5.20: System intents starting an activity of a sample package

Open activity
The intents of types A5_1 and A5_2 were launched by the system to start a sample application activity. This behavior can be achieved by passing a pending intent to the alarm manager service to allow the system to invoke the intent at a later time, as discussed in 3.7.

5.2.6 UI Exerciser Monkey to sample package

ID	Intent	Malware		Playstore	
A6_1	A: android.intent.action.MAIN C: Custom component	41418	100%	14439	99.84%
A6_2	A: android.intent.action.VIEW C: Custom component			3	0.02%
A6_3	C: Custom component			20	0.13%
	Total	41418	100%	14462	100%

Figure 5.21: Intents sent by the UI Exerciser Monkey to start activities of sample packages

Open Activity
The intents A6_1 to A6_3 are generated by simulated user events of the *UI Exerciser Monkey* to start a sample package's activity.

5.3 Evaluation of broadcast intents

This section discusses all broadcast intents launched or received by a sample package.

5.3.1 Sample package to unresolved package

ID	Intent	Malware		Playstore	
B1_1	A: android.intent.action.MEDIA_SCANNER_SCAN_FILE			1	0.43%
B1_2	A: android.intent.action.CLOSE_SYSTEM_DIALOGS	34535	97.548%		
B1_3	A: com.android.launcher.action.INSTALL_SHORTCUT	14	0.039%		
B1_4	A: com.android.launcher.action.UNINSTALL_SHORTCUT	3	0.008%		
B1_5	A: Custom action	868	2.403%	236	99.57%
	Total	35403	100%	237	100%

Figure 5.22: Implicit broadcast intents started by a sample package

Media scanner
Broadcast intents such as B1_1 which use the action *MEDIA_SCANNER_SCAN_FILE* must pass a URI pointing to a media resource along with the intent. The resource is then scanned by the media scanner and afterwards added to the media database for later usage.

Change shortcuts
Broadcasts B1_3 and B1_4 interact with the application shortcut manager, where the former requests creating a new pinned shortcut while the latter requests removing an existing one. The name and shortcut icon as well as action of the shortcut in the form of an intent are passed as extra payloads of the broadcast. Both intents require the user's approval before a shortcut is added or removed.

Close system dialog
Intent B1_2 requests closing temporary system dialogs such as the recent tasks overview or notification messages. Although this intent accounts for 97.54% of all broadcasts in this category, only 9 samples launch the 34,535 intents. Figure 5.23 shows that three samples launch one of these intents each, while the remaining six use this intent several thousand times. The rapid broadcasting of this intent ensures that system dialogs are immediately closed after being opened, thus impeding the user's interaction with the system such as trying to uninstall an application. Notably, malware *Bankbot* uses this intent in conjunction with intent A4_4.

Implicit custom broadcast
Like with B1_5, implicit broadcast intents defining an action are received by all broadcast receivers which have previously subscribed to the respective action. Because custom-defined actions are not known to other applications, the intents could have been only intended for local usage and therefore sent through a local broadcast, as discussed in 3.10.

Malware	Variety	Sample 1	Sample 2	Sample 3
Aples	1	1	1	1
BankBot	3	3947	4999	5037
Koler	2	6866	6840	6843

Figure 5.23: Samples requesting the closing of a system dialog

5.3.2 Sample package to same sample package

Updating app widget
Intents using the action *APPWIDGET_UPDATE* such as specified by B2_1 are used to signal that an app widget component should be updated and the widget's *id* must be passed as extra payload with the intent.

Interaction with Google libraries
Intent B2_2 is used to send data to an app measurement receiver component of the *Google Firebase analytics* service.

Explicit custom broadcast
Broadcast intents B2_3 and B2_4 are received by the same package which started them, and if following the best practice guidelines discussed in 3.10, these intents should have been sent through a local broadcast.

ID	Intent	Malware		Playstore	
B2_1	A: android.appwidget.action.APPWIDGET_UPDATE C: Custom component			25	13.36%
B2_2	A: com.google.android.gms.measurement.UPLOAD C: Custom component			35	18.71%
B2_3	A: Custom action C: Custom component	38	2.31%	115	61.49%
B2_4	C: Custom component	1600	97.68%	12	6.41%
	Total	1638	100%	187	100%

Figure 5.24: Explicit broadcast intents started by a sample package

5.4 Evaluation of service intents

This section evaluates all intents used to interact with service components.

5.4.1 Sample package to third party package

Interaction with Google libraries
Intent S1_1 is used to start a service component of the *Google Analytics* library.

ID	Intent	Malware		Playstore	
S1_1	A: com.google.android.gms.analytics.service.START C: com.google.android.gms.analytics.service.AnalyticsService	2	100%	23	100%
	Total	2	100%	23	100%

Figure 5.25: Explicit service intents started by a sample package

5.4.2 Sample package to unresolved package

Intents discussed in this subsection are implicit and used to interact with service components. As discussed in 3.5.1 only applications using *API* level 20 or lower may use implicit intents in this manner.

Interaction with Google libraries
The intents S2_1 to S2_5 interact with service components of a *Google* library. S2_1 starts a service of the *Google Analytics* library , as it defines the same action as S1_1. Similar to A2_24 and A2_25, S2_2 is used to interact with Android's text-to-speech engine, while S2_3 is used by the *Custom tabs support* library. Intents S2_4 and S2_5 are used in conjunction with *Google Play* billing and ads service respectively.

Custom defined action
Intent S2_6 is defines a custom action for interacting with a package-defined service component.

ID	Intent	Malware		Playstore	
S2_1	A: com.google.android.gms.analytics.service.START	6	4.65%	2	3.57%
S2_2	A: android.intent.action.TTS_SERVICE			19	33.92%
S2_3	A: android.support.customtabs.action.CustomTabsService			4	7.14%
S2_4	A: com.android.vending.billing.InAppBillingService.BIND			5	8.92%
S2_5	A: com.google.android.gms.ads.identifier.service.START			26	46.42%
S2_6	A: Custom action	123	95.34%		
	Total	129	100%	56	100%

Figure 5.26: Implicit service intents started by a sample package

5.4.3 Sample package to Sample package

Package defined service
The intents S3_1 and S3_2 interact with services defined in the their own package.

Interaction with Google libraries
Intents S3_3 and S3_4 target a service component using the *Google Firebase analytics* service and unlike with S2_1 the destination service component is defined.

ID	Intent	Malware		Playstore	
S3_1	C: Custom component	1158	93.61%	1139	54.03%
S3_2	A: Custom action C: Custom component	79	6.38%	421	16.64%
S3_3	A: com.google.android.gms.measurement.UPLOAD C: Custom component			35	1.38%
S3_4	A: com.google.android.gms.measurement.START C: Custom component			934	36.93%
	Total	1237	100%	2529	100%

Figure 5.27: Explicit service intents started by a sample package

5.5 Countering attacks

The above evaluation of the *IPC* traffic, shows that malware employs intents to execute and facilitate malicious behavior, which can be identified by monitoring the intent traffic during runtime by the *EFW*. This section discusses how firewall rules can be configured to counteract these intents.

Component filter
To filter intents solely based on the packages involved in the communication, filter 5.28 matches all intents sent or received by the specified package, which can be used in a rule to monitor all traffic of the respective package without interfering with the *IPC* traffic. An application can be hindered from launch but still receive intents by only defining a sender package filter in a rule set in order to block matching intents. Blocking a package from receiving intents prevents this application's launch since all intents sent to the applications entry points would be blocked.

```
{
    filtertype: or,
    filter: [
        {
        filtertype: equals,
        field: senderpackage,
        value: malicious.package.name
        },
        {
        filtertype: equals,
        field: receiverpackage,
        value: malicious.package.name
        }
    ]
}
```

Figure 5.28: Filter matching sender or receiver package name

Device admin filter
To match device administration request intent A1_9, the filter 5.30 matches the action and component name of the device administration request intent when started by the specified package.

```
{
    filtertype: and,
    filter: [
        {
        filtertype: equals,
        field: senderpackage,
        value: malicious.package.name
        },
        {
        filtertype: equals,
        field: action,
        value: android.app.action.ADD_DEVICE_ADMIN
        },
        {
        filtertype: equals,
        field: component,
        value: com.android.settings.DeviceAdminAdd
        }
    ]
}
```

Figure 5.29: Filter blocking a device admin request intent

Browser URL filter

Filter 5.30 matches URIs containing a certain string when opened in the browser activity component, and the sender package is also matched. A rule with this filter setup can be used to block intents of type A2_1.

```
{
    filtertype: and,
    filter: [
        {
        filtertype: equals,
        field: senderpackage,
        value: malicious.package.name
        },
        {
        filtertype: equals,
        field: action,
        value: android.intent.action.VIEW
        },
        {
        filtertype: equals,
        field: component,
        value: com.android.browser.BrowserActivity
        },
        {
        filtertype: contains,
        field: data,
        value: advertising
        }
    ]
}
```

Figure 5.30: Filter blocking a device admin request intent

Settings broadcast filter

Figure 5.31 shows a filter matching intents using the action *CLOSE_SYSTEM_DIALOGS* sent by the specified package. When used in a broadcast rule this filter can block intents of type B1_2.

```
{
    filtertype: and,
    filter: [
        {
        filtertype: equals,
        field: senderpackage,
        value: malicious.package.name
        },
        {
        filtertype: equals,
        field: action,
        value: android.intent.action.CLOSE_SYSTEM_DIALOGS
        }
    ]
}
```

Figure 5.31: Filter blocking close settings intent

Market scheme filter

Specifying the scheme *market*, filter 5.32 matches all intents using the sender component containing a data URI with this scheme, such as intents of type A3_1.

```
{
    filtertype: and,
    filter: [
        {
        filtertype: equals,
        field: senderpackage,
        value: malicious.package.name
        },
        {
        filtertype: equals,
        field: scheme,
        value: market
        }
    ]
}
```

Figure 5.32: Filter blocking access to market

Once discovered, all malicious intents described above can be blocked via firewall rules, although intents started in rapid succession to screenlock the device pose a challenge. Creating a sender component filter before installing the application package would prohibit such an attack, although it would also prevent the application from starting, thus preventing detection if such an attack were performed by the application without launching the package. An automatic detection module which is specifically designed to detect this type of behavior could thus counteract this attack.

5.6 Screenlock attack detection module

This section discusses implementing a detection module aiming to discover and prevent attacks involving intents from being launched in rapid succession. If activated the detection module receives an intent during phase five of the intent processing (as described in 4.1.4). Because the algorithm's goal is to pinpoint packages with an irregular sending pattern, the module first checks whether the intent stems from a package which was already blocked from sending intents (see ① in Figure 5.33). This ensures that intents are excluded from analysis if they have been passed to the module between when an attack was discovered and the blocking of the malicious package. The analysis module is thus not slowed by unnecessary processing intents. Furthermore, intents launched by a system package are treated as trusted by the module and ignored in the analysis. For all other intents, the analysis calculates a five-second period before the moment the intent was received at the interface of the intent firewall, (② and ③). Afterwards, the module queries a map of intents which retains their occurrences respective receiving timestamp. The algorithm matches the specified action and destination component of the analyzed intent to find its prior usages by the sender package in question. This package is considered malicious if the query shows that 10 intents, including the one currently monitored, were launched by the same package during the calculated time window ④. The data structure containing the information regarding previously analyzed intents is cleaned during each query for a specific intent. To exclude intents originating in this package from further analysis, the internal *UID* block list was updated ⑤. Once a package is flagged as matching the analysis parameter of the module, a new firewall rule is created and blocking the package from sending future intents ⑥. The amount of 10 intents over a timespan of five seconds as a metric for maliciousness was chosen for two reasons. To detect the point where user-interaction, such as closing an unintentional launched activity, is no longer possible, a threshold must be low enough for a human user to respond. Even with the intents evenly clocked each 500 milliseconds, it would be difficult for a user to react at this pace. Although this behavior could be recognized by monitoring the occurrence of 2 intents in a window of one second, spreading of the observation over several seconds, grants a brief temporal increase in intent emergence. For instance, a repeatedly pressed button by a user causing a short increase in intent communication would not trigger the algorithm unless this behavior is continued for several seconds.

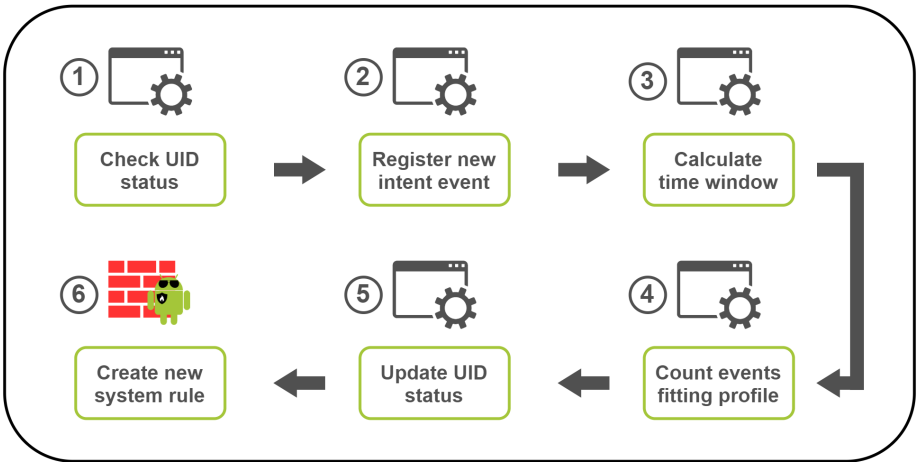

Figure 5.33: Makeup of the screenlock attack detection module

When the detection module blocks a malicious package, the user is informed by a notification (see Figure 5.43) as well as by a popup after reopening the application (see Figure 5.44). In Figure 5.45 the created system block rule can be inspected and deleted by the user, while the event log summarizes the blocked intents by the rule (see Figure 5.46).

5.7 Firewall evaluation

To evaluate the performance of the *Enhanced intent firewall*, a virtual machine was employed running Android version 5.1 with 1 GB RAM. The firewall application was configured to contain 500 identical rules, while an additional performance evaluation application was established to send a specific intent 1,000 times. The firewall rule contained 21 filters, with the first 20 designed to match the performance-test intents' values, while the last one was configured to reject the intents. This ensured that all filter were evaluated against the intent before the firewall could decide on how to handle the intent. Since the same makeup was used in all 500 rules, each performance test intent required evaluation against every rule. The same configuration of the performance test rule and intent were used for evaluating the activity, broadcast and service intents. The makeup of the filter used in the performance test rule is shown in Figure 5.35. Figure 5.34 shows the minimum, maximum and mean timespan required by the *EFW* to evaluate one performance test intent. The mean time for evaluating intents was for all three intent types was 2.2 milliseconds, while the maximum values for examining one intent at 15 milliseconds remained distinctively under the human perceivable delay of 100 milliseconds [74].

Parameter in milliseconds	Activity	Broadcast	Service
Minimum	0.884	0.51	1.328
Mean	2.257	2.265	2.296
Maximum	15.108	8.773	10.060

Figure 5.34: Performance of EFW rules

```
{   filtertype: and
    filter: [
        {   filtertype: equals
            field: senderpackage
            value: intent.test.application.package },
        {   filtertype: equals
            field: component
            value: intent.test.application.package.activity },
        {   filtertype: or
            filter: [
                {   filtertype: and
                    filter: [
                        {   filtertype: exists
                            field: action
                            value: true },
                        {  filtertype: equals
                            field: category
                            value: intent.test.cateory1 }]},
                {   filtertype: and
                    filter: [
                        {   filtertype: equals
                            field: action
                            value: intent.test.action2 },
                        {   filtertype: equals
                            field: category
                            value: intent.test.cateory2 }]}]},
        {   filtertype: equals
            field: scheme
            value: https },
        {   filtertype: or
            filter: [
                {filtertype: and
                    filter: [
                        {   filtertype: not
                            filter: { filtertype: port}},
                        {   filtertype: contains
                            field: host
                            value: foo }]},
                {   filtertype: and
                    filter: [
                    {   filtertype: equals
                        field: host
                        value: bar },
                        {   filtertype: not
                            filter: { filtertype: contains
                                      field: path
                                      value: qux}}]}]},
                        {   filtertype: not
                            filter: {  filtertype: equals
                                      field: senderpackage
                                      value: intent.test.application.package }}]}
```

Figure 5.35: Makeup of the filter in the performance test rule

5.7.1 Performance of screenlock attack detection

To test the efficiency of the above analysis, a sample from each of the 16 malware varieties performing a screenlock attack (as listed in 5.36), was installed on a system using the specifications described above. As Figure 5.36 shows, the mean number of intents sent per second by each of the samples considerably exceeds the minimum required to trigger the detection algorithm, making the time needed to detect the attack only dependent on the time needed by the module to process the first ten intents. Figure 5.37 lists the time measured for the analysis as the minimum, mean and maximum value for the ten intents analyzed by each sample, while Figure 5.38 lists the means for the three parameters shown in Figure 5.37. Assuming the mean value of 0.021 milliseconds, the consecutive analysis of ten intents would require 0.21 milliseconds while a conservative calculation using the average maximum of 0.135 milliseconds would result in a duration of 1.35 milliseconds. After the intent analysis identifies a malicious package, a firewall rule is created and deployed to block the *IPC* traffic of the respective package. Figures 5.39 and 5.40 show the time measured to create a new firewall rule as absolute values for each sample package as well as the minimums, means and maximums of these values. Although a significant difference in length can be observed with a minimum value of 2.136 milliseconds and maximum of 29.255 milliseconds, this step is only necessary when a malicious package is found and thus has overall low impact on the modules performance. The maximum values for the detection and blocking of an attack would lead to a conservative overall estimation of 30.9 milliseconds. Using 100 milliseconds as a threshold for human perception of reaction delays [74], the screenlock would be not perceivable by the user. Although control of the device would be remain in the user's hand, the malicious package would continue to rapidly send intents towards the intent firewall interface. As described in 4.1.4, rules which concern packages blocked by the firewall are evaluated at the earliest point possible to mitigate the impact of rapid intent traffic on the interface's responsiveness. Figure 5.41 shows the time needed to block a single intent by a system-created rule as the minimum, mean and maximum values for each sample as well as their overall averages (see 5.42). With an average mean value of 0.029 milliseconds and average maximum of 0.345 milliseconds, blocking an intent by a system-created rule is considerably faster than the evaluation of user-generated rules. Furthermore, the performance impact on the system by the *EFW* repelling the attack is negligible, while the user retains unrestricted control over the device to remove the malicious package.

Sample	# Intents per minute
BankBot (variety 3)	27.25
BankBot (variety 5)	838.63
BankBot (variety 6)	1636
BankBot (variety 7)	744.4
BankBot (variety 8)	833.07
Koler (variety 1)	28.98
Koler (variety 2)	28.98
Obad (variety 1)	68.84
RuMMS (variety 1)	956.95
RuMMS (variety 2)	802.8
RuMMS (variety 3)	750.89
RuMMS (variety 4)	913.95
SimpleLocker (variety 2)	1406.9
SlemBunk (variety 1)	10.22
SlemBunk (variety 2)	10.58
SlemBunk (variety 4)	9.67

Figure 5.36: Mean intents per second

Sample	Intent analysis time in milliseconds		
	Minimum	Mean	Maximum
BankBot (variety 3)	0.000089	0.017372	0.056308
BankBot (variety 5)	0.011269	0.021797	0.456393
BankBot (variety 6)	0.010901	0.016982	0.067927
BankBot (variety 7)	0.016641	0.019918	1.197159
BankBot (variety 8)	0.012914	0.021149	1.029068
Koler (variety 1)	0.020305	0.029270	1.086133
Koler (variety 2)	0.000089	0.020388	0.476262
Obad (variety 1)	0.017320	0.021097	0.092912
RuMMS (variety 1)	0.000053	0.016490	0.043900
RuMMS (variety 2)	0.010936	0.022293	0.075293
RuMMS (variety 3)	0.014958	0.017673	0.021841
RuMMS (variety 4)	0.015896	0.020236	0.791405
SimpleLocker (variety 2)	0.017813	0.021320	0.169727
SlemBunk (variety 1)	0.014746	0.021579	0.062652
SlemBunk (variety 2)	0.012965	0.021420	1.182842
SlemBunk (variety 4)	0.010226	0.026234	0.100280

Figure 5.37: Analysis time of intents

Parameter	Time in milliseconds
Mean of minimums	0.012939
Mean of means	0.021123
Mean of maximum	0.135000

Figure 5.38: Overall analysis time of intents

Sample	Block rule creation in milliseconds
BankBot (variety 3)	2.286
BankBot (variety 5)	17.026
BankBot (variety 6)	29.255
BankBot (variety 7)	14.628
BankBot (variety 8)	16.242
Koler (variety 1)	8.988
Koler (variety 2)	2.628
Obad (variety 1)	2.290
RuMMS (variety 1)	16.796
RuMMS (variety 2)	18.349
RuMMS (variety 3)	24.077
RuMMS (variety 4)	12.351
SimpleLocker (variety 2)	17.547
SlemBunk (variety 1)	2.136
SlemBunk (variety 2)	2.286
SlemBunk (variety 4)	2.308

Figure 5.39: Creation time of firewall rules

Parameter	Block rule creation in milliseconds
Minimum	2.136
Mean	14.628
Maximum	29.255

Figure 5.40: Overall creation time of firewall rules

Sample	Intent block time in milliseconds		
	Minimum	Mean	Maximum
BankBot (variety 3)	0.005981	0.013503	0.054227
BankBot (variety 5)	0.002925	0.013567	0.038814
BankBot (variety 6)	0.003025	0.015444	0.065348
BankBot (variety 7)	0.000035	0.029886	0.112814
BankBot (variety 8)	0.000035	0.043072	0.048926
Koler (variety 1)	0.002838	0.048258	3.466684
Koler (variety 2)	0.003076	0.034618	0.034618
Obad (variety 1)	0.000036	0.028195	0.195963
RuMMS (variety 1)	0.002955	0.005205	0.031754
RuMMS (variety 2)	0.000035	0.006677	0.097036
RuMMS (variety 3)	0.000036	0.031557	0.033209
RuMMS (variety 4)	0.000035	0.029582	0.029582
SimpleLocker (variety 2)	0.000036	0.113097	0.113097
SlemBunk (variety 1)	0.000035	0.014722	1.11558
SlemBunk (variety 2)	0.000035	0.045051	0.058247
SlemBunk (variety 4)	0.000036	0.004878	0.026048

Figure 5.41: Block time of intents

Parameter	Time in milliseconds
Mean of minimums	0.001322
Mean of means	0.029832
Mean of maximum	0.345121

Figure 5.42: Overall block time of intents

Figure 5.43: Notification of a newly created block rule

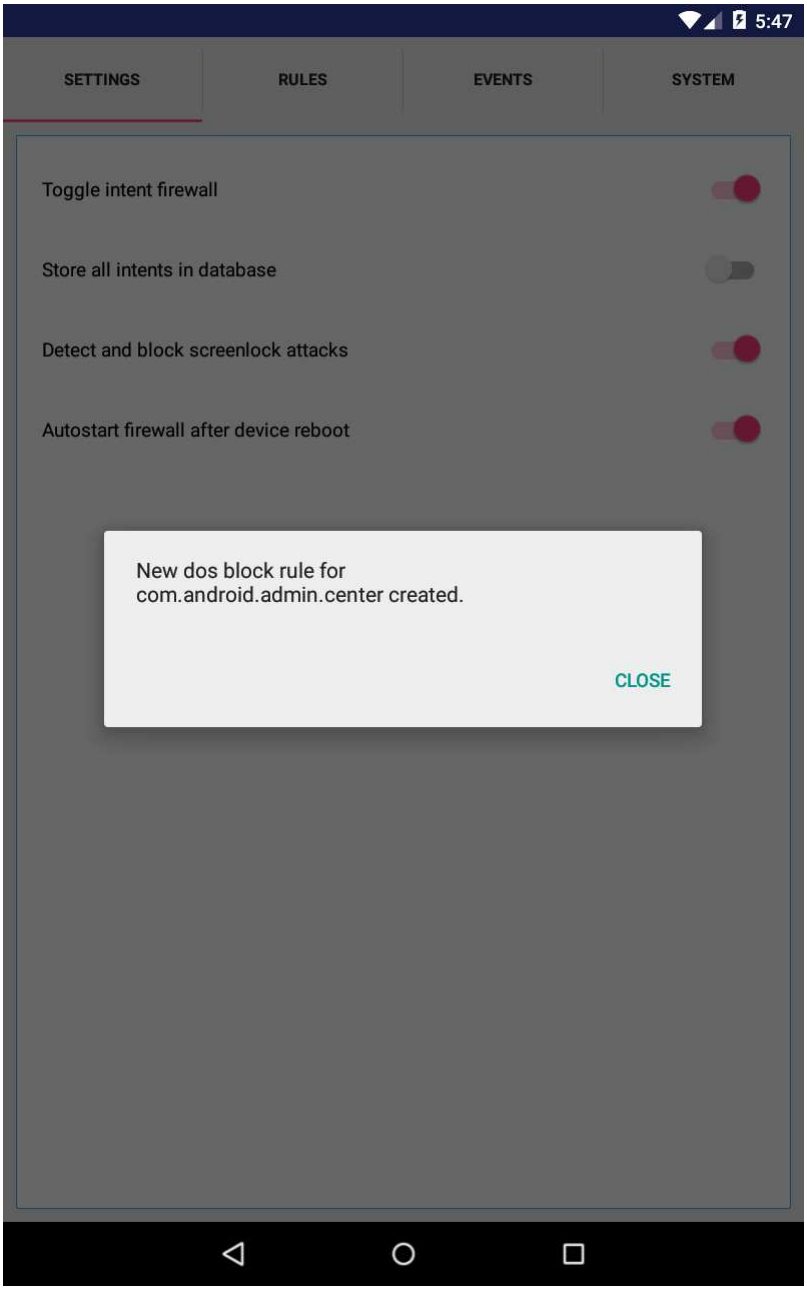

Figure 5.44: Notification of a blocked package

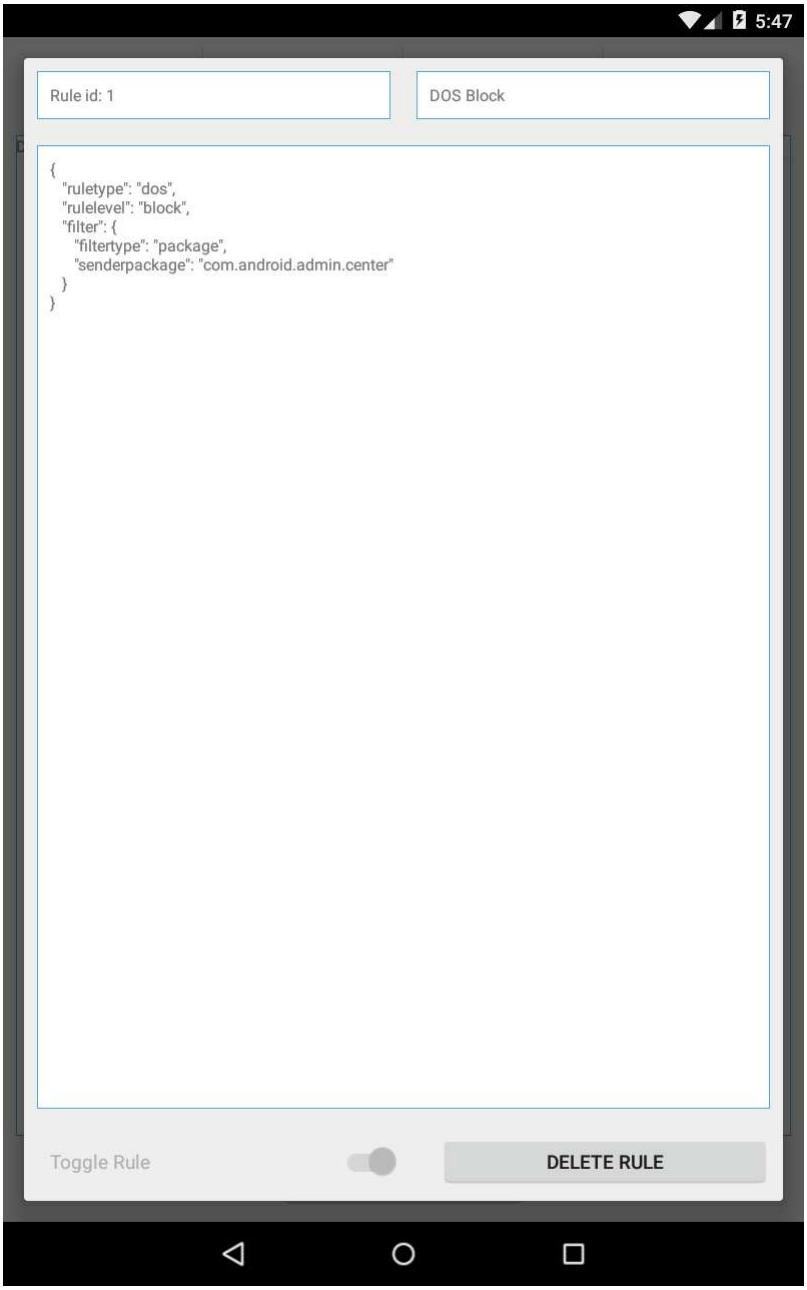

Figure 5.45: Detail view of system block rule

Figure 5.46: Detail view of a system block rule event

CHAPTER 6

Conclusion

The implementation of the inter-process communication *(IPC)* system on the Android platform enables misusage by malicious applications. The intent messages sent via the *IPC* system may be eavesdropped on or forged by malicious applications to prey on vulnerabilities in the implementation of benign apps. To mitigate the repercussions of such attacks, the Android OS allows users to monitor and block intent traffic based on rules through the system's *Intent firewall (IFW)*, although the capabilities and usability of this tool limit its versatility. By employing the *Xposed* framework, the *Enhanced intent firewall (EFW)*, which is based on the *IFW's* approach, demonstrated its capability to compensate the shortcomings of the original implementation without requiring changes to the operating system image beyond a rooted device. By monitoring the *IPC* traffic of known malicious applications, various attacks could be observed and countered by firewall rules without a discernible slowdown in the system's performance. Finally, the extensibility of the approach was shown by implementing a real-time detection module for screenlock attacks, which showed the feasibility of the tool for detecting more complex attacks. The screenlock detection algorithm only considers the frequency, action and component of the sent intent as metrics for maliciousness and does not consider other intent values. Evaluating data fields such as category or data URI allow improving the detection capability. Another possible application for future detection modules is to monitor the intent payload for suspicious values such as IMEI or entries from stored device contacts and to swap this sensitive data with dummy values when certain conditions apply, such as when these data values are passed to a server via URL parameters. Furthermore, the application could be improved through resorting options for created firewall rules to orchestrate rules matching the same intent from coarse to finer-grained filter rules. Finally, while the user interface shows summarized information regarding triggered rules, the database containing the verbose data is only accessible via a database dump. A UI-based export function would offer more convenient access to the collected data.

List of Figures

Bibliography

[1] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[2] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[3] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. pages 229–240, 10 2012.

[4] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[5] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, October 1996.

[6] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 18–34, London, UK, UK, 2000. Springer-Verlag.

[7] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. Composite constant propagation and its application to android program analysis. *IEEE Transactions on Software Engineering*, 42:1–1, 11 2016.

[8] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantic-based detection of android malware through static analysis. pages 576–587, 11 2014.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49, 06 2014.

[10] Sascha Groß, Abhishek Tiwari, and Christian Hammer. *PIAnalyzer: A Precise Approach for PendingIntent Vulnerability Analysis*, page 41–59. 08 2018.

[11] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in android. pages 69–80, 10 2012.

[12] A. Cozzette, K. Lingel, S. Matsumoto, O. Ortlieb, J. Alexander, J. Betser, L. Florer, G. Kuenning, J. Nilles, and P. Reiher. Improving the security of android inter-component communication. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 808–811, May 2013.

[13] Sébastien Salva and Stassia Zafimiharisoa. Apset, an android application security testing tool for detecting intent-based vulnerabilities. *International Journal on Software Tools for Technology Transfer*, 17:201–, 02 2015.

[14] Xianyong Meng, Kai Qian, Dan Lo, and Prabir Bhattachrya. Detectors for intent icc security vulnerability with android ide. pages 355–357, 07 2018.

[15] OWASP Mobile Security Project. Owasp mobile security project - top ten mobile risks.

[16] Atefeh Nirumand, Bahman Zamani, and Behrouz Ladani. Vandroid: A framework for vulnerability analysis of android applications using a model-driven reverse engineering technique. *Software: Practice and Experience*, 10 2018.

[17] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Detecting privacy leaks in android apps. *CEUR Workshop Proceedings*, 1298, 01 2014.

[18] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security*, 21:1–32, 04 2018.

[19] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. 01 2014.

[20] Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. 01 2015.

[21] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. 06 2014.

[22] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. 09 2014.

[23] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, pages 513–527, Cham, 2015. Springer International Publishing.

[24] Songyang Wu, Yong Zhang, Bo Jin, and Wei Cao. Practical static analysis of detecting intent-based permission leakage in android application. pages 1953–1957, 10 2017.

[25] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using covert. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 725–728, May 2015.

[26] Cong Tian, Congli Xia, and Zhenhua Duan. Android inter-component communication analysis with intent revision. pages 254–255, 05 2018.

[27] Rocco Salvia, Pietro Ferrara, Fausto Spoto, and Agostino Cortesi. Sdli: Static detection of leaks across intents. 05 2018.

[28] Yutong Chen. A static detection of inter-component communication vulnerability in android application. In *Proceedings of the 2019 International Conference on Computer, Network, Communication and Information Systems (CNCI 2019)*, pages 532–537. Atlantis Press, 2019/05.

[29] Ryo Sato, Daiki Chiba, and Shigeki Goto. Detecting android malware by analyzing manifest files. volume 36, page 23, 08 2013.

[30] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. 02 2014.

[31] Ke Xu, Yingjiu Li, and Robert Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11:1–1, 06 2016.

[32] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *Computers and Security*, 65:121–134, 2017.

[33] Mohamed El-Zawawy. A new technique for intent elicitation in android applications. *Iran Journal of Computer Science*, 2, 02 2019.

[34] Tristan Ravitch, E. Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. pages 1–10, 12 2014.

[35] Irina As˘avoae, Jorge Blasco, Thomas Chen, Harsha Kalutarage, Igor Muttik, Nga Nguyen, Markus Roggenbach, and Siraj Shaikh. Towards automated android app collusion detection. 04 2016.

[36] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim Elish, and Barbara Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. pages 189–198, 05 2017.

[37] Shweta Bhandari, Frédéric Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S. Roop, and Manoj Singh Gaur. Detecting inter-app information leakage paths. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *AsiaCCS*, pages 908–910. ACM, 2017.

[38] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 71–85, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Hongji Song and Hua Zhang. Drfuzzer: Detector of android app inter-component vulnerability. 01 2016.

[40] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[41] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. pages 118–128, 07 2015.

[42] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 393–407, USA, 2010. USENIX Association.

[43] H. Cai, N. Meng, B. Ryder, and D. Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2019.

[44] M. W. Afridi, T. Ali, T. Alghamdi, T. Ali, and M. Yasar. Android application behavioral analysis through intent monitoring. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–8, 2018.

[45] Raimondas Sasnauskas and John Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, page 1–5, New York, NY, USA, 2014. Association for Computing Machinery.

[46] Daniele Gallingani, Rigel Gjomemo, V.N. Venkatakrishnan, and Stefano Zanero. Static detection and automatic exploitation of intent message vulnerabilities in android applications. 2015.

[47] Bradley Schmerl, Jeffrey Gennari, Javier Cámara, and David Garlan. Raindroid: A system for run-time mitigation of android intent vulnerabilities [poster]. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, HotSos '16, page 115–117, New York, NY, USA, 2016. Association for Computing Machinery.

[48] B. F. Demissie, D. Ghio, M. Ceccato, and A. Avancini. Identifying android inter-app communication vulnerabilities using static and dynamic analysis. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 255–266, 2016.

[49] Shahrooz Pooryousef and Morteza Amini. Enhancing accuracy of android malware detection using intent instrumentation. In *ICISSP*, 2017.

[50] B. Khadiranaikar, P. Zavarsky, and Y. Malik. Improving android application security for intent based attacks. In *2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 62–67, 2017.

[51] J. Tang, X. Cui, Z. Zhao, S. Guo, X. Xu, C. Hu, T. Ban, and B. Mao. Nivanalyzer: A tool for automatically detecting and verifying next-intent vulnerabilities in android apps. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 492–499, 2017.

[52] John Jenkins and Haipeng Cai. Icc-inspect: supporting runtime inspection of android inter-component communications. pages 80–83, 05 2018.

[53] Amr Amin, Amgad Eldessouki, Menna Magdy, Nouran Abdeen, Hanan Hindy, and Islam Hegazy. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10, 10 2019.

[54] Xposed module repository, https://repo.xposed.info/.

[55] Mohamed El-Zawawy, Eleonora Losiouk, and Mauro Conti. Do not let next-intent vulnerability be your next nightmare: type system-based approach to detect it in android apps. *International Journal of Information Security*, 03 2020.

[56] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, page 340–349, USA, 2009. IEEE Computer Society.

[57] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. 2011.

[58] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146, Washington, D.C., August 2013. USENIX Association.

[59] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 514–525, 2016.

[60] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. Appguard – fine-grained policy enforcement for untrusted android applications. volume 8247, 09 2013.

[61] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: extensible multi-layered access control on android. In *ACSAC '14*, 2014.

[62] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. Asm: A programmable interface for extending android security. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 1005–1019, USA, 2014. USENIX Association.

[63] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 40–49, 2014.

[64] Carter Yagemann. Intentio ex machina: Android intent access control via an extensible application hook. 2016.

[65] Youn Kyu Lee, Jae young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 312–323. IEEE Press, 2017.

[66] Android API documentation implict broadcasts exceptions. https://developer.android.com/guide/components/broadcast-exceptions.

[67] Argus cyber security lab, https://www.arguslab.org/.

[68] Genymotion android emulator for desktop, https://www.genymotion.com/desktop/.

[69] Super su root, https://supersuroot.org/.

[70] Apktool - a tool for reverse engineering android apk files, https://ibotpeaches.github.io/apktool/.

[71] Argus cyber security lab, https://www.arguslab.org/.

[72] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *DIMVA*, 2017.

[73] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *Proceedings of the 19th Network and Distributed System Security Symposium NDSS 2012*, 01 2012.

[74] Keeping Your App Responsive. Android developers, 2020 http://developer.android.com/training/articles/perf-anr.html.