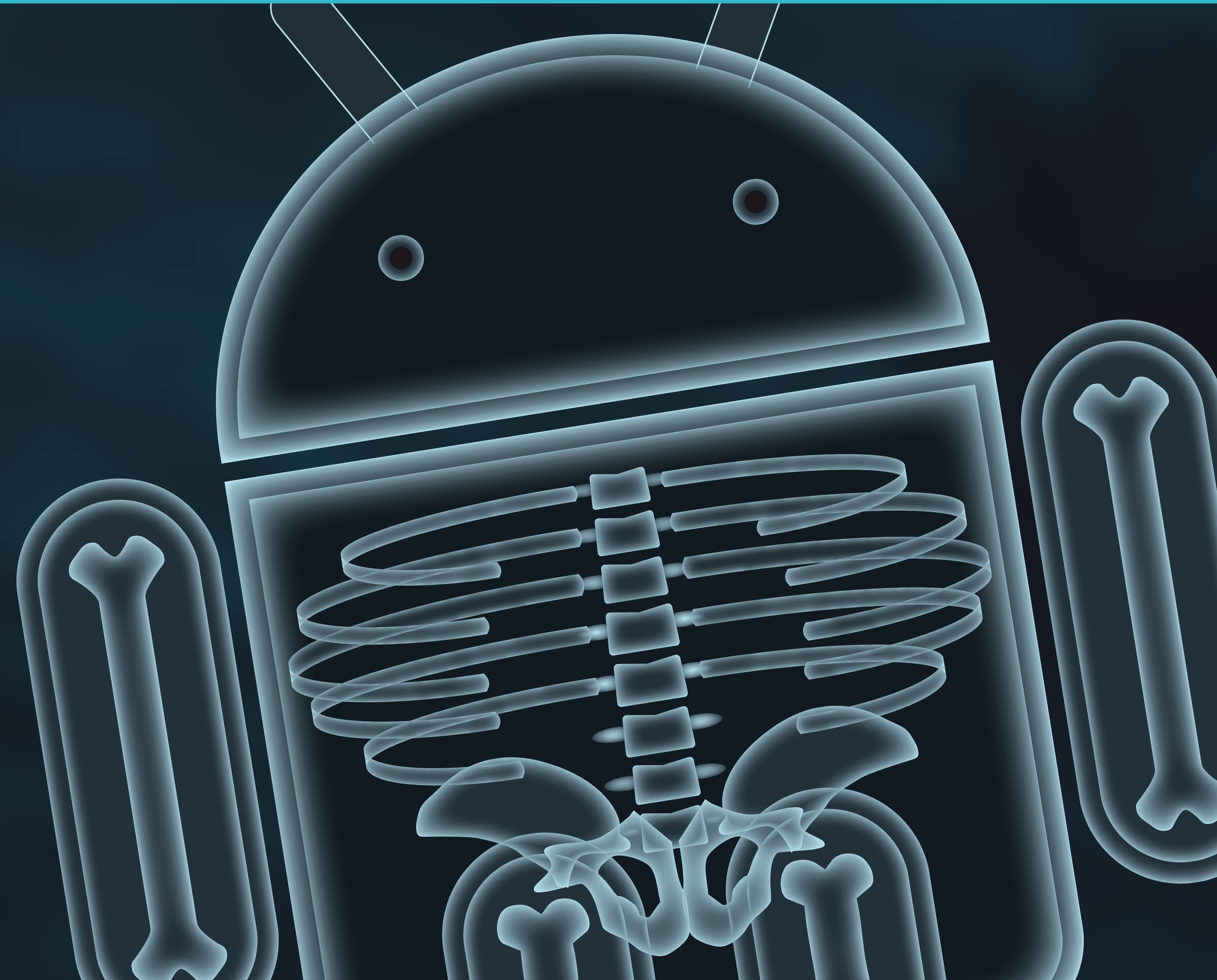
Fraunhofer
AISEC

FRAUNHOFER INSTITUTE
FOR APPLIED AND INTEGRATED SECURITY

# APPRAY:

## USERDRIVEN AND FULLY AUTOMATED ANDROID APP SECURITY ASSESSMENT

DENNIS TITZE, PHILIPP STEPHANOW, DR. JULIAN SCHÜTTE



Fraunhofer
AISEC

# App-Ray: User-driven and Fully Automated Android App Security Assessment

Dennis Titze, Philipp Stephanow, Julian Schuette
{titze,stephanow,schuette}@aisec.fraunhofer.de

December 22, 2013

# Summary

Android is currently the prevailing mobile operating system accompanied by a huge number of apps available at various online market platforms. To protect against malicious or vulnerable apps, Android comprises a permission-based security model and some, but yet opaque security checks conducted by Google Play. Under these conditions, assessing the security of an app according to user-specific requirements is hardly possible. Nevertheless, end users and professionals, such as IT administrators, need to understand apps' security implications prior to installation or rollout. To address this need, we present App-Ray, a novel security scanning framework, which analyses apps according to user-specific security requirements. The contribution of our paper is a method to refine such requirements to specific test criteria, and to automatically combine static and dynamic analysis methods for their evaluation. We demonstrate the feasibility of our approach by implementing a prototype and running user-specific analysis on 50 apps.

# Contents

# 1 Introduction

The paramount success of the Android smartphone operating system and its openness have lead to a huge number of apps available from online market platforms, such as Google Play. In these online markets, many apps contain vulnerabilities, data leaks, or threaten users' privacy by collecting personal information and tracking their behavior. Such security breaches do not necessarily spring from malicious intents of a developer. Flaws that lead to security issues often result from programming errors, caused by time pressure or lack of experience. Unfortunately, such flaws and subsequent vulnerabilities are usually not covered by the platform's security model. This problem also applies to very popular apps, which enjoy trust by the majority of users. Although the permission model of Android is able to prevent apps from using functionality they did not request, e.g., access to Internet resources, it does not support to check the usage of these resources in more detail, e.g., for legitimate URLs, proper use of Transport Layer Security (TLS), or the amount of transferred data. Users have no other possibility but to accept all permissions required when installing an app, even though an app may be over-permissive or contain undesired functionality. Therefore, a user cannot assess whether an app actually complies with her security requirements, even if it remains within the Android permission model.

Complicating the issue even further, nowadays private smartphones are often used for professional purposes. In this Bring-Your-Own-Device (BYOD) context, it has to be decided which business apps can be considered secure, for example, before distributing them over a private marketplace or simply recommending them to employees. Of course, these apps must not contain any malicious code, but it is also important that they do not leak any information, for instance, when being used in insecure networks. Furthermore, it has to be assessed if these apps comply with the overall security guidelines of the company and do not contain hidden functionality, such as taking pictures or recording audio input.

To address these challenges, we propose a framework called *App-Ray*. The two contribution of this paper are:

*User-specific configuration of detection mechanisms.* Starting from high-level security requirements, user-specific declaration of a security requirements catalog is derived and mapped to specific detection rules.

*Automated combination of static and dynamic detection techniques*. According to the detection rules specified by the user, static and dynamic detection mechanisms are automatically combined and executed. App-Ray allows for interweaving of different types of analysis techniques, which, to the best of our knowledge, is not supported by other tools.

The remainder of the paper is structured as follows: The next section gives a brief overview of related work. Section 3 introduces the combination of static and dynamic detection mechanisms, presents the architecture of App-Ray and details on user-specific configuration of malware detection techniques. In Section 4, the prototypical implementation is presented. Section 5 concludes this paper and provides an outlook on future work.

# 2 Related work

Work related to ours is concerned with automated static and dynamic analysis of android apps and unassisted software analysis in general. Also, our framework includes various existing tools for analyzing Android apps, such as apktool [4], a tool for unpacking Android application package (APK) files and inspecting their manifest file, and smali [5], an assembler/disassembler for dex bytecode. While these are helpful tools for creating a framework like App-Ray, they provide limited functionality and do not aim at a comprehensive and automated security inspection of apps as we do.

Dynamic analysis aims to identify vulnerabilities and data leaks at runtime, i.e., by executing and monitoring an application. A straight-forward approach is undertaken by [9] who executes an app within an emulator and simulates user interaction via the Google monkey service. The goal of this work is to observe the behavior of the app in terms of system calls and network traffic. [24] presents MADAM, a framework which monitors execution at different layers to identify malware.

Dynamic taint analysis takes on a more systematic, but heavyweight approach. A prominent solution is TaintDroid [15], a specially crafted Android image capable of detecting sensitive data which is about to leave the device via untrusted sinks. However, this approach suffers from two main drawbacks: Firstly, it requires significant modification of the Android middleware and kernel. Secondly, it does not actively search for vulnerabilities but rather enables the system to detect such when they occur. Dynamic tainting alone is thus not suited for a fully automated analysis of apps. Nevertheless, TaintDroid provides meaningful insights and has been picked up and extended by other tools such as droidbox [1].

One way to control the coverage of control flow paths is symbolic execution. Here, an app is not actually executed, but variables are rather filled by "symbolic" values and the analysis creates logical statements for each variable as it traverses the control flow paths. On the resulting set of statements, model checking can be applied in order to identify infeasible paths or input sets leading to a specific execution path. SymDroid, a symbolic execution framework for Android has been proposed in [21]. A slightly different approach, leveraging the S2E [12] framework for "partial" symbolic execution has been described in [22]. Although heavyweight in terms of required system modifications, symbolic execution is promising fully automated dynamic analysis as it helps identifying input values for relevant test cases.

Another approach to dynamically inspect apps is to observe their behavior at runtime, i.e., their inter-process communication and file access, but without explicitly tracing data flows as proposed in TaintDroid. Here, various authors have described different approaches (c.f [8, 25, 29, 30, 10]). However, applying behavior based inspection for a fully automated analysis is quite difficult. It requires a comprehensive data set based on interaction with the GUI to learn legitimate behavior and detect illegitimate apps.

While dynamic analysis is well suited to identify actual vulnerabilities and data leaks as they occur, static analysis aims at identifying possible flaws by inspecting the application without running it.

Considering static analysis, one early approach inspects the set of permissions required by an Android app and checks whether it contains critical combinations [16]. [14] investigates whether apps' permissions exceed those typically required for a specific type of application. More advanced approaches create call graphs (*interprocedural*) and control flow graphs (*intraprocedural*) and try to extract information about information flows and critical code patterns from it. In general, three frameworks for static code analysis are worth mentioning: Soot, Androguard, and Wala. Soot [23] is a framework for static analysis of Java bytecode, featuring a conversion of bytecode into four different representations at different abstraction levels. As one of the most comprehensive frameworks, it allows to implement custom analysis through various extension points. While Soot has been designed to operate on Java bytecode, extensions to support instrumentation of dex bytecode have been published recently [6]. Soot is relatively mature and its recent extensions for dex bytecode look promising, but its feature-richness comes at a cost in terms of complexity and overhead. In contrast to Soot, Androguard [13] focuses specifically on Android and operates directly on dex bytecode, skipping any error-prone translations to higher-level languages. It is thus leaner and can be extended by directly modifying its source code. Wala [3] is another static analysis framework which has originally been created for Java, similar to Soot. It provides the basis for AndroidLeaks [18], a tool to identify data leaks in Android applications. In contrast to our approach, the detection, i.e., the definition of data leaks, is not configurable. Static analysis is also deployed in [19] and [11] to detect *capability leak* vulnerabilities which can lead to so-called *confused deputy* attacks.

As static and dynamic analysis are complementary, combining both approaches is an obvious improvement. ProfileDroid [28] applies both approaches in order to automatically create profiles of apps, however without searching for security-relevant flaws, as we do. RiskRanker [20] is similar to our approach in that it identifies potential security flaws in apps by combining different detection mechanisms, but similar to the approach in [7] it does not consider user-specific security requirements to check for.

# 3 Framework design

The design goal of App-Ray is to create an extensible framework for a fully automated and user-driven security inspection of Android apps. The framework's components serve the following purposes:

- Specifying and refining security-related user requirements,
- orchestrating different analysis modules, and
- evaluating the results according to the previously defined requirements.

The two main challenges of the framework design are (1) the mapping of high-level security goals to specific detection rules and (2) automated orchestration of static and dynamic analysis components.

When testing an application, users simply choose a so-called *protection profile*, i.e., a set of security requirements to check the application against. The framework then breaks these requirements down to specific analysis tasks. This is done by the core component of the framework, the *analyzer*. The actual analysis functionality is implemented by *detection modules* which are dynamically added to the framework if they are needed to assess a certain requirement. Each detection module has to provide an interface for configuring and executing its analysis. Having derived the configuration values for all detectors involved in a test case, the analyzer puts all detectors in a scheduling list and sets them up by calling their configuration method. After that, the analyzer runs the analysis tasks by invoking the execute method of the previously configured detectors.

Detectors are first scheduled in phases by their type, referring either to *meta data analysis*, *static analysis*, or *dynamic analysis*. The results of each phase are collected in a common data structure and passed on to the following detection modules enabling subsequent detectors to use the results of previous ones.

Each phase is described separately in the following subsections. Furthermore, the integration of App-Ray into the threat assessment process is presented in Section 3.6.

## 3.1 Meta data analysis

The meta data analysis collects information about the app, such as the contained files, native libraries, as well as information from the `AndroidManifest.xml` file. Here, security relevant information is gained from the set of required permissions, entry points to the application in form of activities, content providers, and services. In addition to that, external services, which have been integrated into App-Ray, such as virustotal [2] for checking the app for known exploit signatures, are invoked and their return data is collected. Meta data analysis serves as an efficient way to get first insights into an app and to collect information paving the way for more detailed, subsequent inspection techniques.

## 3.2 Static analysis

In a next step, detectors for static analysis are invoked. Here, the actual dex bytecode of the application is inspected to detect patterns indicating malicious or vulnerable code. At this stage, detectors gather various information about the app, such as contained classes and methods, call graphs and data flow graphs, implemented interfaces, etc. Based on this information, App-Ray statically searches for potential information leaks, i.e., unwanted data flows sending private information (location, contact data, etc.) to untrusted APIs (sockets, browser intents, etc.). Other examples for potentially malicious code comprise attempts to place tapjacking attacks by means of fullscreen enlarged toast messages, attempts to record touch events via transparent `SYSTEM_ALERT` windows, as well as any attempt to invoke suspicious files, such as `su` (superuser), `busybox` (utilities), `/dev/input/event*` (touch events), `/dev/input/fb` (framebuffer), and others. In particular, one test has proven highly relevant in practice: to check how the interface `javax.net.ssl.TrustManager` is implemented [17]. On the one hand, this is required for certificate pinning, i.e., an improvement of the certificate verification relying on Android's built-in CAs. On the other hand, however, the default TrustManager is often overwritten to simply eliminate certificate checks, thus leaving the communication open to Man-in-the-Middle attacks. App-Ray checks for such cases by inspecting the bytecode of any implementation of the TrustManager interface to distinguish whether it is a certificate pinning or a "void" TrustManager implementation. Practical experience with our prototype strongly confirms the findings of [17], stating a significant amount of Android application are prone to Man-in-the-Middle attacks due to intentionally overwritten TrustManagers.

It is in the nature of static analysis that these findings may be inaccurate, and produce false positive or false negatives. False positives can occur, e.g., if a GUI with a transparent full screen `SYSTEM_ALERT` window is actually required for certain interactions. False negatives may result from attack patterns which have not been anticipated and thus are not checked for, or if applications circumvent detection, e.g., by constructing malicious code dynamically at run time.

## 3.3 Dynamic analysis

To improve detection capabilities, App-Ray adds a dynamic analysis step in which the application is executed within an emulated environment. Here, we inspect the runtime behavior of the application and validate potential leaks previously identified by the static analysis. Also, checks for further information are run, including: capturing network traffic and filtering for potential privacy breaches, user tracking, monitoring file access, and tracking information flows with dynamic taint analysis, as described in [27]. One challenge in dynamic analysis is to drive an application down relevant execution paths to cover all security-critical executions paths. App-Ray allows to record and replay user inputs to automatically simulate the execution of an app. Having applied this record and replay technique within our prototype to a variety of apps makes us confident that this approach is applicable to discover flaws in the applications. However, we acknowledge that symbolic execution depicts a solution to capture all possible execution paths from which input data for automated test cases could be derived. Of course, symbolic execution comes with its own drawbacks and challenges not to be discussed within this paper. Nevertheless, App-Ray supports the integration of such techniques and they will be part of our future work on automated security tests for Android.

## 3.4 Combination of static and dynamic tests

So far, meta data analysis, static analysis and dynamic analysis have been applied separately, only interacting via a common data structure, which passes results between detection modules. However, App-Ray also supports dependencies between detectors, thereby enabling a combination of static and dynamic analysis techniques which leads to increased detection rates and less false positives. As each detector has access to the analyzer's scheduling queue, it is possible for one detector to invoke another one and to continue working on its provided information. For instance, during the static analysis, a detector might collect all class names and methods present in the code. This can easily be done by inspecting the dex bytecode, however, as soon as classes are loaded via reflection, e.g., via `Class.forName("classname")`, static analysis comes to its end, especially when classes are loaded from a remote location. In this case, the static analysis detection module can schedule a dynamic analysis module providing all remote data, wait for it to finish its execution and then retrieve the loaded class from the dynamic detector's result data.

## 3.5 Evaluation of the analysis

One goal of App-Ray is to support automated tests for user-specified security requirements. Hence, it is not possible to hardcode the evaluation of the previously collected raw data from the detectors, but rather a flexible, rule-based approach is necessary. Rules are defined as boolean expressions over predicates of the attributes which are collected during the analysis. Attributes may thus contain information such as method names, strings, decompiled source code, data flows, etc. Predicates are functions which take attributes as input and return a boolean value. They can be registered in the framework to support various evaluations, as shown by the following exemplary rule, using a `trivialImpl` predicate to check for flawed TLS implementations, and a `implClasses` function which returns all used classes that use the given interface. So, a (simplified) rule may look as follows:

$$\begin{aligned} \mathit{TLSFlaw} = {} & \mathit{trivialImpl}(\mathit{implClasses}(\text{HostnameVerifier})) \\ & \vee \mathit{trivialImpl}(\mathit{implClasses}(\text{X509TrustManager})) \\ & \vee \mathit{in}(\text{AllowAllHostnameVerifier}, \mathit{usedClasses}) \end{aligned}$$

This rule checks all classes implementing the *HostnameVerifier* or *X509TrustManager* interface for a flawed TLS implementation, or if the *AllowAllHostnameVerifier* class is used for TLS hostname verification.

## 3.6 Integration into existing threat assessment processes

App-Ray configures detection modules according to user-specific security requirements and thus returns user-specific reports. It is suitable to be embedded in threat assessment processes executed by security officers or administrators responsible for distributing apps to a user group, e.g. via an enterprise market. In the following, we describe the process of deriving user-specific security requirements and using them to configure detection modules. Note that eliciting security requirement is a process that requires domain specific knowledge, usually at an expert level, which can be tool-supported but is hard to automate. However, the process of security requirement elicitation itself is not in focus of App-Ray.

In a typical first step of threat analysis, assets of the mobile platform are analyzed. They are defined by the user and may relate to the result of a certain usage context the device is intended to work in or to an user-specific app to be deployed on the device. To exemplify our approach, it is assumed that a user wants to install a banking app on her device to conduct financial transactions. User-specific assets, such as sensitive data contained in the banking app, are refined by an expert until a technical representation is achieved. In the context of the banking app, part of such a technical representation of assets is the implementation of methods needed to encrypted the transaction data, authenticate the remote banking server and ensure integrity of transaction data sent. After having analyzed a device's assets, threats are derived. At first, the motivation of an attacker has to be modeled by identifying high-level goals an attacker is persuading, such as financial gains. In our example, an attacker may want to intercept financial transactions and change the receiver's account to rewire the transaction and thus steal the funds to be transferred. From the motivation of an attacker, technical targets are derived. In the banking app's case, one technical target may be to manipulate the communication for the financial transaction from the mobile device to the remote banking server. Then, ways to reach these targets, i.e., attack vectors are described. At this point, the results of the asset analysis pay off by providing domain specific knowledge. In our banking app example, the attacker may try to exploit a flaw in the app's security model. One known vulnerability which could allow the attacker to launch a man-in-the-middle attack may exist if the banking app overwrites the `javax.net.ssl.TrustManager`, thereby removing proper certificate and host name validation (c.f. [17]).

Based on the threat analysis, security requirements to test for can now be derived from the attack vectors. By negating the attack vectors' entry points, corresponding security requirements are derived. Consider again the example of overwriting the TrustManager with a void implementation: the entry point of the respective attack vector is e.g., any class extending `javax.net.ssl.TrustManager`, removing proper certificate and host name validation. Hence, the corresponding security requirement reads "An app shall not overwrite the default TrustManager with non-implemented methods". Negating this requirement leads to a security requirement which can directly be translated to the above mentioned evaluation rule for App-Ray (see Section 3.5).

Figure **??** shows how the derived security requirements control the execution of App-Ray and its detectors.

# 4 Prototype implementation

The App-Ray prototype supports user-specific requirements and is able to check apps from various sources. Apps are analyzed statically and dynamically using different security scanning techniques, e.g., using TaintDroid [15], Androguard [13], and information directly available from the application file as explained in Section 3.

All dynamic tests are executed in a virtual environment where apps are started and observed for several seconds. As already mentioned in Section 3.3, App-Ray supports scripting of simulated user interactions with an app, for further implementation details please refer to [27]. Generally, apps need different user input ranging from simple clicks to more difficult tasks, such as solving a level in a game. Since our tests presented hereafter focus on the feasibility of automating user-specific security scanning, the record and replay capabilities are disabled for these tests. To generate comparable output from the prototype for our tests, all apps are started and operated in the same way. Starting an app and observing the behavior for several seconds can already provide meaningful insights into an application, e.g., about the network traffic during start up. However, we acknowledge that results of certain dynamic techniques, e.g., behavior-based analysis, may benefit from the deployment of record and replay techniques to generate and execute user input.

Our prototype combines different scanning techniques by orchestrating them in the analyzer component according to the requirement specific protection profile. Results returned by the detectors are converted into a common data format and passed on to the evaluation component where the results are matched against the protection profile. However, to show feasibility of App-Ray, it is sufficient to summarize the results of all scanning techniques and provide a brief explanation for each finding.

## 4.1 Practical Application

Evaluating the feasibility of App-Ray to configure detection techniques according to high-level security goals, we selected privacy as a goal to start from. On this basis, we derived security requirements of a fictional user as described in Section 3.6 and selected the following two user-specific protection profiles (non-technical description) as part of the test set:

- Does the app use TLS, and if so, is it implemented correctly?

- Does the app include tracking or advertisement libraries whilst having access to sensitive information of the user?

These profiles are exemplary and App-Ray is not limited to them but rather can be configured to cover many others. Each protection profile is mapped to security requirements whose results are categorized as either *OK* or *NOK (Not OK):*

*TLS usage:* If an app (or parts of an app) communicates over HTTP, this is categorized as *NOK*, if all monitored communication is over HTTPS as *OK*.

*TLS implementation flaws:* Apps (or included libraries) can implement their own certificate verification which can either be non-existent or accept all certificates without any validation. Both cases result in *NOK*. If no custom certificate verification is implemented or the certification validation is implemented correctly or the certificate validation is not removed, it is categorized as *OK*.

*Profiling:* Apps can contain libraries for profiling users or collecting crash reports sending out private information to third-party services. If the app can also read sensitive information, i.e., IMEI, IMSI, telephone number, or location, this can indicate a privacy violation (*NOK*). If the app does not have access to sensitive information or does not contain libraries for profiling, it is categorized as *OK.*

*Advertisement:* Similar to tracking, apps can use advertisement libraries and have access to sensitive information. Since this can be a problem if the ad library uses sensitive information (e.g., as shown in [26]), this finding is categorized as *NOK*. If the app does not have access to sensitive information or does not contain advertisement libraries, it is categorized as *OK.*

These four security requirements result in four different evaluation rules. The rule for the TLS implementation flaw can be found in Section 3.5. The evaluation rule for Advertisement looks as follows:

$$\begin{aligned} Adv &= \left(in\left(permissions, \mathrm{INET}\right) \wedge in(libs, adLib_x)\right) \\ &\quad \vee in\left(capturedTraffic, adHost_y\right) \end{aligned}$$

The remaining two evaluation rules are similar to the two depicted ones and are omitted for the sake of brevity.

## 4.2 Results

50 popular, free apps available at Google Play have been tested, selected from the top 10 of the categories *business*, *communication*, *productivity*, *social*, and *tools* (January 2013). The scans were executed on a current PC (Core i5@3.3GHz, 8GB RAM, Ubuntu 12.04 64 bit) and took between 50 s and 980s (depending on the complexity and size of the app), with a mean of 185 s ($\sigma = 171$) and a total runtime of 155 min. The results are summarized in Table 4.1.

| Requirement | *NOK* | *OK* |
|---|---|---|
| TLS usage | 31/50 | 19/50 |
| TLS implementation flaws | 26/50 | 24/50 |
| Profiling | 29/50 | 21/50 |
| Advertisement | 24/50 | 26/50 |

Table 4.1: Test results

These results were reviewed manually, e.g., for TLS implementation flaws, the application's implementation of the certificate verification (if such a verification exists) was manually inspected to verify App-Ray's findings. The results show that our prototype is capable of scanning selected apps according to previously elicited, user-specific security requirements. On the basis of these results, the user is thus able to assess apps' security without manual inspection. App-Ray therefore provides meaningful information about an app's security and can serve as an information basis for an administrator who can, in turn, recommend apps or further investigate results not compliant with her requirements.

The test's results show that the scanned apps often contain severe security issues, for instance, more than half of the scanned apps contain profiling libraries and have access to sensitive information of the device.

# 5 Conclusion

In this paper, we presented an user-driven and fully automated Android app security assessment framework, called App-Ray. Our framework combines static and dynamic scanning techniques and analyzes apps according to user-specific security requirements, thereby providing lightweight integration into existing threat and risk assessment processes. A user can scan apps according to specific protection profiles and receives detailed results tailored to these requirements. As the user does not have to configure the scanning techniques, our framework is well suited for users without profound knowledge in, e.g., reverse engineering or bytecode analysis.

A practical application of the developed prototype with 50 popular, free apps from Google Play showed that more than half of the apps do not comply with a selected set of user-specific requirements representing a user's privacy.

In our future work, we plan to extend the analysis capabilities of App-Ray by further techniques, based on bytecode instrumentation and symbolic execution. We also intend to extend the catalogue of predicates and attributes of our evaluation rules. Thereby, we envision to better address the challenge of automatically generating artificial, but meaningful simulated inputs to the application under test. Studies on specific types of vulnerabilities and their distribution across the different markets will follow.

# Bibliography

[1] droidbox – Android Application Sandbox. http://code.google.com/p/droidbox/, accessed 10th Apr. 2013. 6

[2] VirusTotal. http://www.virustotal.com/, accessed 10th Apr. 2013. 9

[3] WALA – T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/, accessed 10th Apr. 2013. 7

[4] android-apktool – A tool for reverse engineering Android apk files. http://code.google.com/p/android-apktool/, accessed 29th Jan. 2013. 6

[5] smali – An assembler disassembler for Androids dex format. http://code.google.com/p/smali/, accessed 29th Jan. 2013. 6

[6] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM. 7

[7] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE)*, pages 66 –72, oct. 2011. 7

[8] A. Bauer, J.-C. Kuster, and G. Vegliach. Runtime verification meets android security. In *NASA Formal Methods Symposium (NFM'12)*, pages 174–180, Norfolk, Virginia/USA, April 2012. Springer-Verlag. 7

[9] T. Blaesing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak. An Android Application Sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62, 2010. 6

[10] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile device (SPSM)*, pages 15–26. ACM, 2011. 7

[11] P. P. Chan, L. C. Hui, and S. M. Yiu. DroidChecker: analyzing android applications for capability leak. In *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, pages 125–136. ACM, 2012. 7

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011. 6

[13] A. Desnos and G. Gueguen. New "open source" step in android application analysis. In *10th annual PacSec conference*, Nov. 2012. 7, 13

[14] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of malicious applications on android os. In *Proceedings of the 4th international conference on Computational forensics*, IWCF'10, pages 138–149, Berlin, Heidelberg, 2011. Springer-Verlag. 7

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX conference on Operating systems design and implementation*, 2010. 6, 13

[16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM conference on Computer and communications security (CCS)*, pages 235–245. ACM, 2009. 7

[17] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM. 9, 12

[18] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *5th international conference on Trust and Trustworthy Computing (TRUST)*, 2012. 7

[19] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2012. 7

[20] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *10th international conference on Mobile systems, applications, and services (MobiSYS)*, 2012. 7

[21] J. Jeon, K. K. Micinski, and J. S. Foster. Symdroid: Symbolic execution for dalvik bytecode. Technical report, University of Maryland, 2012. 6

[22] A. Kirchner. Data leak detection in smartphone applications. Master thesis, Technical University Vienna, Chair for Computer Science, Nov. 2011. 6

[23] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS Users and Compiler Infrastructure Workshop*, Oct. 2011. 7

[24] A. Saracino, F. Martinelli, D. Sgandurra, and G. Dini. Madam: a multi-level anomaly detector for android malware. In *Int'l Conf. Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS)*, 2012. 6

[25] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2011. 7

[26] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. *IEEE Mobile Security Technologies (MoST)*, 2012. 14

[27] D. Titze, P. Stephanow, and J. Schütte. A configurable and extensible security service architecture for smartphones. *Int'l Symposium on Frontiers of Information Systems and Network Applications (FINA)*, 2013. 10, 13

[28] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *18th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2012. 7

[29] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, WiSec '10, pages 37–48, New York, NY, USA, 2010. ACM. 7

[30] M. Zhao, T. Zhang, F. Ge, and Z. Yuan. Robotdroid: A lightweight malware detection framework on smartphones. *Journal of Networks (NoW)*, 7(4):715–722, 2012. 7