FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Data Leak Detection in Smartphone Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Andreas Kirchner

Matrikelnummer 0600112

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. George Candea (EPFL)
Mitwirkung: Vitaly Chipounov (EPFL)

Wien, 19.11.2011

Andreas Kirchner
(Unterschrift Verfasser)

George H. Candea
(Unterschrift Betreuung)

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

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Data Leak Detection in Smartphone Applications

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Andreas Kirchner
Registration Number 0600112

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. George Candea (EPFL)
Assistance: Vitaly Chipounov (EPFL)

Vienna, 19.11.2011

Andreas Kirchner
(Signature of Author)

[illegible]
(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andreas Kirchner

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, 22.11.2011	Andreas Kirchner
(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank *Vitaly Chipounov* from the dependable systems lab at EPFL. He helped me to find my way through puzzling error messages during the implementation phase. He even was not afraid of answering mails that were decorated with assembly code, bullet points and log files. Finally, he gave me instructive feedback on my thesis and thesis poster.

In the same breath, I want to thank *Prof. George Candea*, head of the dependable systems lab at EPFL, an inspiring teacher and advisor. During my exchange semester at EPFL, he and his team at the dependable systems lab revived my fascination for software development; he drew my attention to the rich opportunities of smartphones and the art of software testing; he was willing to be my thesis advisor although I decided to go back to Austria to finish my master there. Finally, he helped me to refine my ambitious thesis idea and introduced me to people which do research in software analysis and data leak detection.

Then I thank my friends *Angelika Hofstetter and Melina Höhn* for their feedback on the poster. Moreover, Melina is the creator of the avatar shown below — it worries about data security in its smartphone.

A master thesis is a full time job, at least it was for me. My friends can confirm that. I want to thank all of them for their mental support and their distractions: *Agnieszka, Sophie, Goldi, Klaus, Maria, Babsl, Kathl, Florian*, and all I have not mentioned.

This thesis would not have been possible without the support of my parents *Peter and Wilma* during my entire time as a student. I thank them also for their faith in me and my abilities.

Finally, I want to thank my wise and charming penpal *Emma* who kept my mind awake for other topics than data leak analysis. Her emails are a synthesis of surprising interventions and instructive observations — amusing and profound. May Kafka be with you. . .

Abstract

When a user installs a third party application on her smartphone, she does not know in advance whether sensitive data — like her current location, the unique device ID or even data about her health condition — leaves the phone. This raises new concerns about privacy and data security, since more and more people are using smartphones in private and business life.

In this thesis, we design and implement a prototype of `Leakalizer`, a tool to automatically detect data leaks in smartphone applications in three steps. First, `Leakalizer` thoroughly explores execution paths of the target application. Then, for each path, it traces sensitive data. Finally, `Leakalizer` reports all sensitive data that leaves the smartphone. The prototype emulates a smartphone device on a desktop PC to run the Android software stack, applies symbolic execution to traverse execution paths of the target application, and uses dynamic taint tracking to detect transmissions of sensitive data.

`Leakalizer` demonstrates how a thorough exploration of execution paths in a smartphone application can be used to make data leaking behavior visible, which helps users and organizations to assess the trustworthiness of a given application.

Kurzfassung

Wenn eine Benutzerin eine Smartphoneanwendung von Drittanbietern installiert, weiß sie vorher nicht ob sensible, auf dem Smartphone gespeicherte, Daten — wie etwa Daten über ihren aktuellen Aufenthaltsort, die eindeutige Geräte ID oder sogar Daten über ihren Gesundheitszustand — ihr Telefon verlassen. Das wirft Bedenken bezüglich Privacy und Datensicherheit auf, da Smartphones im beruflichen und privaten Alltag immer häufiger eingesetzt werden.

Diese Diplomarbeit beschäftigt sich mit dem Design und der Implementierung eines Prototyps von `Leakalizer`, einem Tool zur automatischen Erkennung von Datenlecks in Smartphoneanwendungen. Das geschieht in drei Schritten: Zuerst führt `Leakalizer` eine gründliche Untersuchung der Ausführungspfade der Zielapplikation durch. Danach werden sensible Daten für jeden Ausführungspfad verfolgt. Schließlich meldet `Leakalizer` wenn sensible Daten das Smartphone verlassen. Der Prototyp erstellt ein virtuelles Smartphone auf einem Desktop PC um den Android Software Stack zu starten, er wendet symbolische Ausführung an um Ausführungspfade der Zielapplikation zu traversieren, und er benutzt Dynamic Taint Tracking um das Versenden von sensiblen Daten zu erkennen.

`Leakalizer` demonstriert wie die systematische Untersuchung von Ausführungspfaden in Smartphoneanwendungen verwendet werden kann um Datenlecks sichtbar zu machen. Dies hilft Benutzerinnen und Organisationen die Vertrauenswürdigkeit von Applikationen von Drittanbietern zu bewerten.

Contents

CHAPTER 1

Introduction

In the 21st century the technology revolution will move into the everyday, the small and the invisible. (Mark Weiser)

In recent years, smartphones have become ubiquitous. Within one year — between June 2010 and June 2011 — the average time an US-citizen spends with mobile applications almost doubled from 43 minutes to 81 minutes per day, which is more than the average time spent on Web consumption (74 minutes per day) [24]. In June 2011, 400,000 Android phones were activated worldwide every day. One month later, the number increased by 38% to 550,000 [11]. The Android market store exists since August 2008 to provide and download third party applications. Within two month (from May to July 2011) the number of downloads increased by 50% from 3.0 billion to 4.5 billion [51].

The ability to access information from everywhere is one reason why smartphones are widely used in daily life. There are third party applications to read mails, to post on social networks, to manage the daily schedule, to find the nearest restaurant, and many more. The applications can be easily downloaded from smartphone application repositories. Examples are Google's Android Market for Android smartphones, Apple's App Store for iPhone and iPad applications, or Microsoft's Windows Phone Marketplace. Most of the applications are for free or available for low price.

Moreover, smartphones change the organization of businesses [36]. For example, in mobile health care, smartphones can be used to update electronic health records, improve the doctor-patient relationship, approve drug prescriptions, and access current evidence-based clinical guidelines [8]. According to various studies, more and more doctors and nurses are willing to use smartphones at work because they already carry a smartphone for private use [31] [48]. The health informatics sector also realized that smartphones can be useful in health care. According to a global survey among mobile health application developers, 78% of the respondents think that smartphones offer the best business opportunities for mobile health care applications [23].

1.1 Problem

The highly increasing use of smartphones raises new concerns about privacy and data security. Since smartphones are our daily companions, they store data which represents sensitive information about our private and professional life, or data which helps to derive this information. For example, we usually do not tell our employee the phone number of our doctor or the contents of short messages we send to our friends, unless there is a reason to do so. And we usually do not want our smartphone to automatically transmit such data to strangers.

Discussions on privacy date back to the 19th century, a time where newspapers and photography found their way into the everyday life. For example, Warren and Brandeis published an article in the Harvard Law Review in which they argued for the right to privacy, the right to be let alone [49].

In the 21st century, Internet and mobile devices heavily expanded communication technologies and made people think about privacy under current conditions. Advocates of a transparent society state that we should arrange ourselves in a world where we do not (or should not) have anything to hide anymore [50] [9]. According to a mediating position, privacy and transparency are not mutually exclusive, because privacy is not identical with secrecy [42] [16]. For example, it is not contradictory to demand more transparency — e.g., a society where public and private realms are melting together — and also demand that an owner of a smartphone should be able to control how sensitive data is transmitted.

We believe that privacy in smartphones is not primarily about holding back secrets but about feeling comfortable when using the rich opportunities of smartphones. If a user has control over her data, she might be more willing to use her smartphone for payments or storing her electronic health record, because she can more reasonably trust the applications which provide these features.

Various studies and articles have shown that a significant amount of smartphone applications transmit sensitive data without the user's permission. The Wall Street Journal tested 101 iPhone and Android applications. They found that 56 applications transmit the unique device ID and 47 disclose the location of the phone [46] [53]. Another study presented at the Black Hat Conference 2011 revealed that in a sample of 10.000 Android applications 8% leak private information. 11 applications even transmit the content of short messages [19]. Other studies come to comparable results [21] [27] [20]. Almost weekly, new cases about suspicious smartphone applications appear [30] [43].

The studies above let us conclude that our privacy suffers when installing some third party applications on smartphones. On the one hand, installing third party smartphone applications make the smartphone interesting, smart and useful. On the other hand, installing third party applications is like giving an invisible agent access to your home. In the digital realm, transmission and duplication of data can be done within milliseconds. With built-in technology, smartphone users do not have a chance to detect and block ongoing behavior on their smartphone.

Applications are to some degree able to access personal data like location data or the unique device ID. Before one notices, the current location or even the whole address book could have been sent over network to an advertisement company or even malicious destinations. Advertisement companies could exploit this by aggregating informations associated with the same device

ID to track users.

Consider the following scenario: A smartphone user is installing a weather application and is shown a permissions dialog. She might ask herself: “What if I install this weather application? The permissions dialog tells me that the application needs Internet access, that it accesses location data and the unique device ID. But what kind of data will be actually sent and to which destination?”.

Not every transmission of sensitive data has of course malicious intents, but one cannot know in advance whether the receiver redistributes the data or just processes it in order to provide a useful weather application. A weather application may require location data to be transmitted in order to provide the user with a relevant weather report [53]. But the server who receives the data could send it to third parties without the user’s agreement nor knowledge. In general, every redistribution of sensitive data increases the risk that sensitive data is revealed to unintended agents, either on purpose or by accident.

As soon as privacy is taken serious as the users ability to control distribution of sensitive information, the user needs to be informed when her smartphone transmits sensitive data over network.

Many smartphone applications do not inform users about the transmission of sensitive data. If there exists a permission system in the mobile operation system (like in Android), it is not fine-grained enough to control the type of data which is sent outside. If the user had a comprehensive summary of the application’s behavior, it would be easier to assess the risk of installing an application. Providing this information would be the first step to support privacy for smartphones.

Data leak detection tools assist users and organizations in deciding if they can trust a given application. The decision to trust an application cannot be automatically determined, it has to be taken by the user. As long as smartphone applications do not notify users when sensitive data is transmitted, additional tools need to extract this information.

Data leak analysis can also contribute to improve privacy support for smartphone applications. It requires not only legal regulations to face the challenges of privacy but efforts in software development. Speaking about privacy is pointless when a user or organization does not have control over her data any more. By automatically uncovering data leaks, smartphone application developers can be motivated to better support privacy.

The thesis deals with the problem of automatically detecting data leaks in smartphone applications. The motivation to solve this problem is to provide users and organizations with information about the usage of sensitive data. It is especially important to provide that information before the data leak occurs, e.g., before the application is installed.

1.2 Leakalizer - A Data Leak Analyzer for Smartphone Applications

In this thesis, we design and implement a prototype of `Leakalizer`, a tool to automatically detect data leaks in smartphone applications. `Leakalizer` detects data leaks and lets the user decide if a data leak means a privacy restriction to her. A privacy restriction is a restriction on the user's ability to determine for herself when, how and to what extent sensitive data is communicated to others. Before she installs an application, she can use `Leakalizer` to find out which data will be sent to which destination. `Leakalizer` is designed to run on a host computer or to be embedded in a Web service. We envision `Leakalizer` to create data leak reports of smartphone applications. Given a smartphone application, it shows to the user which type of sensitive data is eventually transmitted over the network, e.g., to an advertisement server. `Leakalizer` systematically explores possible execution paths of the application and creates a report that points out leakage of different types of sensitive data.

`Leakalizer` complements existing approaches to detect data leaks in smartphone applications. The main contribution is a path exploration engine which builds the basis for systematic detection of data leaks in a running Android application. To our knowledge, such an exploration engine has not been implemented before to automatically detect data leaks in Android applications. To achieve this, `Leakalizer` uses dynamic taint tracking and symbolic execution for a virtual smartphone.

The prototype is based on S2E, a platform for building multi-path analysis tools [15], the Android emulator which ships with the Android Software Development Kit [5], and an extended Android operating system.

1.3 Overview

The thesis is divided into the following chapters: Background (2), Design (3), Implementation (4), Evaluation (5), and Discussion (6).

Chapter 2 defines privacy, data leak, and sensitive data and discusses the state of the art in data leak analysis for smartphones.

Chapter 3 gives an answer to the research question of this thesis: How is it possible to systematically find data leaks in smartphone applications? From the requirements of an ideal data leak detection tool we derive five components to detect data leaks in smartphone applications. Finally, we decide on the techniques to implement the components.

Chapter 4 gives implementation details of `Leakalizer`, a data leak detection tool for Android applications. The prototype emulates a virtual smartphone device and applies symbolic execution to analyze Android applications. We discuss implementation challenges of `Leakalizer`.

Chapter 5 examines the effectiveness of `Leakalizer`. Is the proposed design suitable to detect data leaks in a systematic way? We use microbenchmarks on our prototype, to answer this question.

Finally, chapter 6 discusses the results of the thesis and sketches ways to extend and improve the prototype.

CHAPTER 2

Background

This chapter provides a definition of concepts used throughout the thesis and an overview of state of the art approaches in data leak analysis for smartphone applications. We classify and discuss existing approaches.

2.1 Definitions

First we define central notions used in this thesis: privacy, sensitive data and data leak.

Privacy is “the ability to determine for ourselves when, how and to what extent information about us is communicated to others” [17]. Thus, we speak of a *privacy restriction*, if this ability is restricted.

Sensitive data is the material basis for ’information about us’. The term denotes for example data about health, salary, current location, social network, unique identifiers, or daily schedule of a person. For companies, data of current projects or personal data of customers are considered to be sensitive. There is no method to distinguish between sensitive and non-sensitive data, because what is sensitive varies over time and depends on the social, cultural and legal context.

A *data leak* in general is any transmission of data to an undesired receiver. An *undesired receiver* is any receiver in a network which is not explicitly accepted as receiver of data. Privacy focuses on the handling of *sensitive* data. Therefore, a more narrow definition is used: A data leak is any transmission of *sensitive* data to an undesired receiver. For example, the purpose of a weather application is to provide recent weather reports based on the current location of the user. When the application sends the content of short messages over the network, a user will almost certain consider the transmission of short messages as a data leak, but she will probably not consider anonymous usage statistics about the most heavily used features as data leak or as a privacy restriction.

A *data distributor* is any agent who is responsible for data management of a person or institution [40]. In smartphones, multiple data distributors are involved. The smartphone operating system is a data distributor that gives sensitive data to third party applications. Moreover, third party applications are data distributors that occasionally transmit sensitive data over the network or to other third party applications. The receivers are again data distributors.

Privacy Support: An application (or operating system) supports privacy if it allows users to control the transmission of sensitive data. We consider privacy support as a quality attribute of software [39], like performance and usability.

We do not use the term 'privacy leak' because an ability cannot leak out; data can leak out of software (like oil leaks out of a ship).

2.2 State Of The Art Data Leak Analysis

Existing approaches fall under three main categories: (1) data leak detection, (2) data leak prevention, and (3) security enforcement. Data leak detection tools aim to detect data leaks in applications. They do not necessarily block software behavior which causes the data leak in a running system, but some do. Data leak prevention tools help the user to protect sensitive data stored in her smartphone. The user can specify what the system should do in case of an access to sensitive data. Security enforcement tools augment smartphone operating systems with additional security mechanisms which sometimes include protection of sensitive data.

Data Leak Detection

PiOS is a tool which analyzes the flow of sensitive data to detect data leaks in iOS applications [20]. PiOS proceeds in three steps: (1) First, PiOS uses static analysis to create an approximation of the control flow graph (CFG) of application binaries. (2) Then, the tool performs a reachability analysis on the CFG to find paths between methods which access sensitive data and methods which transmit data over the network. (3) To ensure that paths reflect the information flow between source and sink, PiOS performs additional analysis. The resulting paths entail a potential data leak. With this approach the authors analyzed the distribution of sensitive data in 1607 applications.

PiOS does not cover all data leaks due to three reasons: (1) Given the static binary analysis and the characteristics of Objective C[1] the analysis could not detect the corresponding Objective C methods for 18% of method calls, because in Objective C methods are called by sending a message. A dispatch function then sends the message to its destination at runtime which makes it difficult to find the right method name statically. Thus, the approximation of the CFG could miss data leaks, i.e., introduce false negatives. (2) The CFG does not include data flows that are interrupted by user interface interactions. The reason is that user event handling is located in the sources of the iOS software stack (which is not statically analyzed) and not in the application's binary. Although it is useful to filter out cases where the application asks the user to permit or block the transmission of sensitive data, it can introduce false negatives. (3) The current implementation supports a maximal path length of 100 nodes in a CFG. The evaluation in the paper does not mention false positives, i.e., wrong classification of data transmissions as leaks of sensitive data.

[1] Objective C is the programming language used in iOS applications.

TaintDroid is an privacy-monitoring application which sends notifications to the user if a third party application sends sensitive data via network. It requires to modify the Android platform on the user's phone and allows to monitor the flow of sensitive data for installed applications. For this purpose, TaintDroid uses dynamic taint tracking, an approach to label data and propagate the label during execution, based on different propagation rules [38]. Instead of using tainting at machine instruction-level, TaintDroid integrates four levels of taint tracking to detect data flows: variable-level tracking, method-level tracking, message-level tracking, and file-level tracking. The authors of TaintDroid identified different taint sources (e.g., in the location manager which supplied third party applications with coordinates from the GPS sensor) to taint sensitive data which is propagated along the flow of data and maybe reach a taint sink (e.g., a method which sends data over network). Both taint sources and taint sinks have to be located in interpreted bytecode. The authors of TaintDroid formally defined the propagation rules for Dalvik bytecode instructions.

TaintDroid aims to give users and security companies real time notifications about data leaks. It is designed to run on the phone on a daily basis. Thus, unmodified Android has to be replaced by TaintDroid which has three major drawbacks: Firstly, replacing the firmware of a phone leads to warranty loss. Moreover, replacing the firmware is likely to be too much work for an average user if she only wants to know how a given application uses her personal information. Second, not all third party applications are supported by this particular implementation, because TaintDroid does not allow an application to execute its own native libraries. Only native system libraries (e.g., OpenGL) are allowed. This is the consequence of considering the whole Android software stack as trusted code and the only untrusted code comes from interpreted code, executed by instances of the Dalvik virtual machine. Third, TaintDroid does not offer to block the behavior or never install applications that disclose sensitive data. If the user is notified after her data is exposed, it is too late to uninstall a leaking application, because it is impossible to reverse a data leak.

Finally, TaintDroid is not designed for systematic analysis of smartphone applications to detect data leaks, because the exploration of execution paths is only triggered by the actions of the user. There is no mechanism which guarantees to cover more than one execution path.

AppInspector is a tool for detecting data leaks of Android applications [26]. It is a successor project of the TaintDroid research group. The system is not realized yet, but the design aims to utilize a virtualized smartphone on a host computer where a modified Android software stack is augmented with four components: (1) An execution engine included in Android's application runtime to explore relevant execution paths. (2) The information flow tracking component utilizes dynamic taint tracking to mark sensitive data and log their propagation during execution. (3) The privacy analyzer creates dependency graphs to perform various analysis techniques which help to better understand potential data leaks. (4) AppInspector also aims to take into account end user license agreements (EULA) to distinguish between legal and illegal access of sensitive data. Natural language processing and crowd sourcing should be utilized to detect whether the text inside the user notification mentions private data. While AppInspector is similar to the design of `Leakalizer`, AppInspector has two drawbacks: First, AppInspector is not implemented yet. Second, the aim to distinguish between proper and improper use of sensitive data is framed as the problem of natural language processing which by itself is an open research question [6].

Data Leak Prevention

TISSA implements a privacy mode for Android [53]. With a privacy mode, users can setup and enforce their own privacy rules. The rules specify which data is accessible from third party applications. If an application requests data which does not comply with the privacy rules, TISSA generates bogus data, returns an empty result or anonymized data. TISSA adds an additional permission specification and enforcement layer on top of the Android software stack. To achieve this, three components are added: (1) Policy Decision Point: A content provider to store privacy permissions for third party applications. (2) Policy Administration Point: An Android application which serves as settings manager, i.e., lets the user update privacy settings. (3) Policy Enforcement Points: This is the heart of the privacy mode. Functionality is embedded in Android components (e.g., contacts content provider, location manager, telephony manager). They regulate the access to sensitive data based on stored permissions in the policy decision point. In the current implementation, three types of data are supported: phone identities (IMEI, MEID), location data, and phone call log. TISSA is a straightforward and efficient implementation of a privacy mode. Benchmarks have shown no measurable performance overhead.

Security Enforcement

SCanDroid extracts security specifications from the Android manifest of an application and checks at runtime if the application behaves in accordance to the security specification [25]. The researchers define abstract semantics for Android applications which allows to statically analyze data flows. The approach is based on former research on language based security for Android [12]. The implementation is built on top of WALA – a tool chain for Java program analysis. SCanDroid was not applied to real-world applications.

Paranoid Android (PA) proposes security as a service for smartphones [41]. The user's smartphone is synchronized with a virtual smartphone running as a process on a server. PA applies virtual machine recording and replaying techniques to perform security checks. The smartphone operating system is augmented with a tracer component on the users phone. The tracer first records various types of events on the user's phone: system calls and signals. PA records system calls to capture for example network traffic, sensor data and user actions. PA traces signals to capture errors like segmentation faults or timer expirations. After recording, PA sends the traces to the virtual device on the server on a regular basis. The virtual device then replays all events but additionally performs security checks. PA implements an anti-virus scanner and dynamic taint analysis (similar to TaintDroid) to perform the security checks.

From a privacy perspective, there are two major issues of decoupling security from execution: Firstly, storage and transmission of the event traces are by themselves a threat to security and privacy. As recently shown, a debugging tool for HTC phones could be exploited to gather sensitive user data [43]. Secondly, the user has to trust the server that hosts an exact copy of the smartphone and processes the same amount of sensitive data as the user's phone.

In this section, we have defined central notions used in this thesis. Then, we described six data leak analysis tools along with their advantages and limitations. As we will see later, the most interesting for the purpose of our thesis are the three data leak detection tools TaintDroid, PiOS and AppInspector because they focus on data leak reports. While TaintDroid requires to change the firmware of the user's phone, PiOS and AppInspector are independent from real smartphones and perform the analysis on a host computer. AppInspector has not been implemented yet.

CHAPTER 3

Design

In this thesis, we build `Leakalizer`, a tool that detects data leaks in smartphone applications. The user provides a smartphone application and receives a data leak report which makes it easier to decide whether to install the application or not. Therefore, the tool should not require the user to install the application on her phone. As a side effect, such a data leak detection tool could also *prevent* data leaks because the generated report could keep users from installing data leaking applications or organizations from embedding them in their IT infrastructure.

The research question is: How is it possible to automatically detect data leaks for a given smartphone application with high accuracy and without requiring the user to install the application on her phone?

We split the question in three parts: (1) *Requirements:* What do we demand from a data leak detection tool for smartphone applications? (2) *Components:* Which components are necessary to automatically find data leaks? (3) *Choice of Techniques*: Which techniques can be used to implement these components?

3.1 Requirements

An ideal data leak detection tool has the following properties:

A **Work on Binaries**: Usually, smartphone applications come in packages which do not contain the source code. For example, application developers write Android applications in Java. Then, the source code is compiled to Dalvik bytecode. A tool that analyzes Android applications should work on Dalvik bytecode and should not depend on the availability of Java source code.

B **No false negatives**: The analysis should find all leaks of sensitive data. Users and organizations which use the tool must be sure that they can rely on the report. In other words, if the report states that the application does not introduce data leaks, then the application does not introduce data leaks, no matter what the user or system input is.

C **No false positives**: A false positive is when a data transmission is wrongly qualified as leak of sensitive data. In other words, given a list of data types which are considered to be sensitive, the analysis should only report data leaks that actually are leaks of sensitive data.

D **Independence from user phone**: The tool should neither require the user to do any modifications on her phone nor should it rely on the user's phone at all. The rationale is that a data leak detection tool is more valuable when it allows the user to also prevent data leaks. Only when the tool is independent from the user's phone the user can read the data leak report and decide whether to install the application on her phone. For example, the tool could run on a typical desktop computer or as a web service.

The properties B and C express that the tool should be accurate. In fact, a data leak detection tool is a binary classifier. Each data transmission is either a data leak or not. To assess the performance of binary classifiers, sensitivity and specificity are often used. For example, in medical statistics, sensitivity and specificity are used to assess the performance of diagnostic methods.

In the context of finding data leaks, sensitivity describes the probability that the data leak detection tool reports a data leak when the transmission is actually a data leak. Specificity describes the probability that the tool classifies a transmission as harmless when the transmission actually is harmless, i.e. not a data leak.

Do existing tools meet our requirements?

We already described existing approaches to detect data leaks in chapter 2: TaintDroid and PiOS.[1] We shall see now to what extend they meet the requirements.

PiOS performs static analysis on binaries, which meets requirement A. Moreover, the evaluation does not mention false positives. Static analysis is known for giving false positives [13]. Thus, we consider requirement C as not fulfilled. PiOS has false negatives. It cannot find data leaks in 27 out of 172 tested applications due to limitations of the static analysis approach. That means, sensitivity is 84% in the sample of PiOS. Finally, it does not require to install the application on the phone which meets requirement D. To summarize, PiOS meets requirement A and D.

TaintDroid is embedded in the OS of the user's phone and tracks the flow of sensitive data at runtime. This means that no source code is required and requirement A is met. The authors of TaintDroid state that false positives are possible due to different granularity levels of data tainting. Thus, requirement C is not met (we are not provided with numbers). Finally, false negatives are possible due to various reasons. The most crucial reason is that TaintDroid relies on user inputs to detect data leaks. This is sufficient to observe the behavior of already installed applications and when manual application usage is no problem. This approach is not suitable for systematic exploration, because it is impossible that a user (or any other agent) explores all possible combinations of user inputs. Finally, TaintDroid does not meet requirement D, because it requires the user to modify the operating system on her phone and requires to install the application she wants to analyze. In summary, TaintDroid only meets requirement A.

[1]We cannot judge on AppInspector because implementation is not finished yet.

Neither TaintDroid nor PiOS meet our requirements. One major reason is that they are not sufficiently accurate. Therefore, we aim to improve the accuracy of data leak detection with the constraint to be independent from the user's phone.

We claim that a systematic exploration of possible execution paths is the key to a thorough data leak analysis. We present systematic exploration techniques in the next sections.

3.2 Components

The design of a data leak detection tool consists of the following high level components that we describe in this section: (1) entry point invocator (2) exploration engine, (3) tracer, (4) transmission observer and (5) reporting component.

To explain the purpose of the components, we define the following terms: unit, environment, and entry point.

A *unit* is the smartphone application of interest. The *environment* is the code that surrounds the unit. For each run, we concentrate on exactly one application of interest, i.e., on the unit. The environment is the whole system without the unit, as defined in [15].

An *entry point* is a point in the application where execution jumps to after an event occurs. The invocation of an entry point leads to the execution of a sequence of instructions, depending on the input. For example, consider figure 3.1 that shows entry points of a simplified weather application. The arrows represent method calls that follow from the invocation of an entry point. When the user presses a button the application calls the method `onTouch()` of an event handler and the instructions of `onTouch()` are executed. If the user presses another button, the application calls a method of another event handler — probably with different instructions.

The interactions between a unit and its environment can be understood by the model of a biological cell. A cell has multiple receptors to interact with the surrounding environment awaiting particular events to happen. Events activate receptors and initiate a cascade of processes inside the cell. In software, receptors are called entry points.

We first give an overview of the components and then discuss each of them in detail. The *entry point invocator* determines and invokes all entry points that could contain a data leak, e.g., event handlers, or methods that start the life cycle of an application. For each entry point, the *exploration engine* explores possible execution paths of a smartphone application. The *tracer* is responsible to label sensitive data (e.g., unique device identifiers) in order to distinguish them from other data circulating in the system (e.g., anonymous weather report). The *transmission observer* raises an alert when the application transmits labeled data over network. Finally, the *reporting component* generates a report of all found data leaks including additional information.

Entry Point Invocator

Depending on the design of the smartphone operating system, smartphone applications can have multiple entry points which are usually activated by events. In the example of figure 3.1, after the application invokes `onTouchListener`, it calls a handler for the `onTouch` event. If the input indicates that the user has touched the "generate weather report" button, the application calls the corresponding method `genWeatherReport()` which calls another method `displayWeather()`.

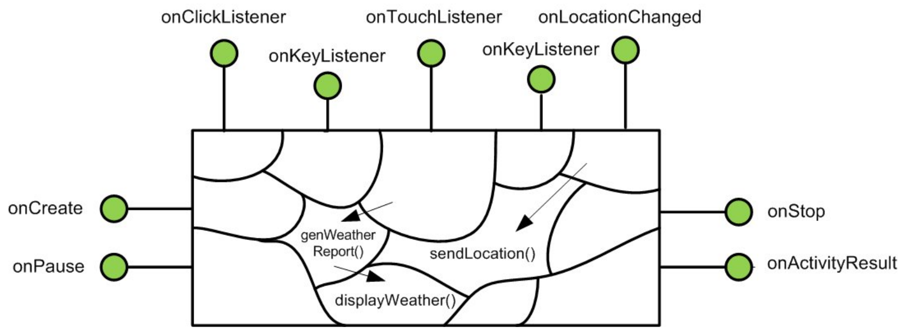


Figure 3.1: Multiple entry points in a simplified Android application. Event handlers trigger execution of different code fragments of the application.

To systematically detect data leaks, we need to find, invoke, and explore all entry points that contain third party code. Otherwise, we would miss potential data leaks. Let us take a second look at figure 3.1. From within the `onTouchListener`, not all methods that belong to the application are reachable. This is different from software that starts at the main method and where all code is reachable from within the main method. If we only explore the execution paths of the `onTouchListener` entry point, we will never find the data leak inside the `sendLocation()` method no matter what input (in particular, touch event) is generated.

The entry point invocator finds and invokes entry points. After that, systematic exploration of each entry point is done by the exploration engine.

Exploration Engine

The exploration engine is the heart of our tool because it determines the accuracy of the analysis.

To explain the task of the exploration engine, we define the following terms: program state, execution path and code coverage. A *program state* describes a snapshot of the system that includes all values used by the application and the system to function properly. An *execution path* consists of a sequence of program states.

Code coverage is a measure of systematic software testing. It describes the degree to which the source code of a program has been explored. Or in other terms: It describes the proportion between the number of actually explored execution paths and the number of all possible execution paths of the unit.

When executing a program with concrete input, not all possible execution paths are actually executed. Consider the following example:

```
public Report genWeatherReport(boolean useCache,
                       Location location, Date date) {
    if (useCache) {
       return getCachedReport(date);
    }
    String uid = TelephonyManager.getDeviceId();
    return retrieveReport(location, date, uid);
}
```

The method `genWeatherReport()` has two possible execution paths where one leads to a data leak. Depending on the parameter `useCache`, the method either returns a cached weather report or retrieves the report from the web by sending current location, current date and the unique device ID of the phone.

The higher the code coverage the higher the likelihood to find all data leaks. In the example above, if our exploration engine explored only the execution path that returns the weather report from the cache, it would not find the data leak, i.e. our exploration engine would result in a false negative.

It is not feasible to automatically check all possible execution paths for the whole system because the number of paths grows exponentially with the number of branches, which leads to the path explosion problem. In practice, an exploration engine that suffers from path explosion 'never' finishes the analysis.

We do not need to check all execution paths of the environment. In the example above, the telephony manager is part of the smartphone operation system and not part of the third party application. Since we only want to check if code from the third party application causes a data leak, we do not need to systematically explore code from the telephony manager. Thus, the exploration engine executes the method `getDeviceId()` of the telephony manager only to get a return value from the function.

To avoid the path explosion problem, we need a technique that explores possible execution paths of the unit and not of the environment. Since a typical smartphone application is much smaller than the surrounding environment, the exploration engine has significantly less execution paths to explore.

Depending on the way how smartphone applications are built, it may be challenging to distinguish between unit and environment from the perspective of the exploration engine. We will come back to this point in the implementation chapter at section 4.4.

Tracer

The tracer component focuses on the flow of sensitive data. It distinguishes between sensitive and non-sensitive data by labeling sensitive data at a relevant point, e.g., when the system generates the data or when the data enters the unit.

Moreover, the tracer distinguishes between types of sensitive data. Therefore, there are different labels for each relevant type of data. For example, to later generate a meaningful data leak report, labels of device identifiers are different from labels of location data.

For each execution path, the labeling mechanism needs to be consistent with the execution semantic. In particular, the tracer attaches a new label when a statement of the application copies an already labeled value. Additionally, the tracer preserves a label when a statement modifies sensitive data. In section 3.3 we discuss techniques to achieve this.

Transmission Observer

The transmission observer focuses on end points of sensitive data. It hooks calls to the network interface to find out, whether the application transmits sensitive data — i.e., data labeled by the tracer component — over network. If the application transmits labeled data, the transmission observer sends the type of data and the destination address to the reporting component.

We need to cover all interfaces that establish connections to a network. If one interface is missing, it can be used to leak undetected data leaks which introduces false negatives.

Reporting Component

The reporting component aggregates data that was collected during the analysis of the application and generates a human-readable data leak report. If the tool discovers a data leak it reports the following information: the entry point (e.g., an event handler), the type of sensitive data (e.g., location data), the IP address or the host name of the receiver (e.g., `198.78.202.118` or `fbi.gov`). Optionally, the execution trace can be replayed to reproduce the data leak and to enable the user to study under which conditions the data leak occurred.

3.3 Choice of Techniques

Which techniques allow to implement the five specified components of the last section? Until now, we have developed the high-level design of a data leak detection tool that aims to meet four requirements. The requirements reflect that our data leak detection tool has high accuracy, only needs the binary form of the application and is independent from the user's phone. We have specified five components that explain how we can meet these requirements in general.

Our data leak detection tool needs at least two techniques to implement all components, (1) a technique to systematically explore execution paths and (2) a technique to find data leaks for each execution path. A technique to systematically explore execution paths allows to implement the execution explorer and the entry point invocator. A technique to find data leaks allows to implement the tracer, the transmission observer and the reporting component.

For both techniques, we briefly describe possible strategies and assess their advantages and disadvantages. Based on the result of the assessment, we decide the techniques that we use for the prototype.

Exploration of execution paths

The task of our exploration engine is to explore execution paths of a smartphone application. In other words, the question is how to analyze the full spectrum of potential behavior, encoded in the unit? In the specification of the exploration engine (cf. section 3.2) we described that this question is crucial to automatically detect data leaks with low false negatives.

Data leak detection rate can be improved when the exploration of execution paths does not require the user to manually supply inputs by themselves. But which alternatives do we have, when we do not want to rely on manually providing user input, like TaintDroid does, as described in section 2.2 and 3.1?

Systematic exploration of execution paths can be done using static and dynamic approaches.

One approach is to statically obtain the control flow graph (CFG) of the unit to get an approximative CFG of all possible execution paths of the unit. As described earlier, the static approach was used by PiOS to detect data leaks (cf. section 2.2 and 3.1). One advantage of static analysis is that the application does not need to be executed. One disadvantage is that - due to limitations of static analysis [34] - it is impossible to obtain an exact control flow graph statically. This does not mean, that static analysis is insufficient for data leak analysis. PiOS has shown that static analysis is applicable for data leak detection. Would a dynamic approach yield better results?

A common practice in dynamic analysis is to randomly generate user input. By using the generated input in multiple executions, dynamic analysis explores different execution paths. The advantage is that it is straightforward to implement and there are already tools for common smartphone operating systems, like Monkey in Android [1]. Unfortunately, a study found out that this is not effective for smartphone applications, because they have multiple entry points that are coupled with user interface elements (like buttons and form elements). According to the study, randomly generated user events leads to code coverage of 40% or lower [26]. Since the disadvantages are predominant, we do not consider this option.

Another technique to systematically explore execution paths is *symbolic execution*. Instead of executing the program with concrete inputs (e.g., `zipcode = 2060`), symbolic execution assigns symbolic values (e.g., `zipcode = alpha`) to the inputs, which represent any possible concrete value that the variable could take. It then executes the program using these values.

Consider the following function that is called when the user edits the zip code field. The function updates the object's state, then retrieves the current day of week in order to send the zip code once a week to an advertisement server. Under symbolic execution, the zip code and the day of week are both symbolic. When symbolic execution reaches the if statement, since the condition variable is symbolic, the execution follows both paths. The transmission observer component automatically flags a leak when it notices that a symbolic user input (e.g., zipcode) is sent over the network along the true path. Without symbolic execution, thorough dynamic data leak detection would require blind guessing of inputs or exhaustive input enumeration, which is prohibitive.

```
public int onFieldEdit(int zipcode) {
   m_zipcode = zipcode;
   int day = Calendar.getInstance().get(Date.DAY_OF_WEEK);
   if (day == 1) {
      sendOverNetwork(zipcode);
   }
}
```

An early publication about symbolic execution compares symbolic execution with algebra [32]. The authors stated that symbolic execution is related to concrete execution as algebra to arithmetic. Like Algebra, symbolic execution *delays* the concrete computation by using symbols and collecting the effects that computations would have on the result.

The power of symbolic execution is its ability to cover whole sets of execution paths. In the function above, symbolic execution covers all execution paths within this function. One execution path assumes that the variable `day` is equal to 1 and another assumes that the variable `day` is not equal to 1. The first one calls the function `sendOverNetwork()`, the second one does not. In contrast, concrete execution follows only one path of execution because the variable day has one concrete value which forces the execution to either enter the if branch or not.

However, symbolic execution is affected by the state explosion problem. Like other multi-path analyses (e.g., model checking) symbolic execution allows to assert properties for a set of execution paths without testing each input value. But whenever symbolic execution reaches a branch, the exploration engine has to create a new execution state. The effect is, that the number of paths grows exponentially with the number of branches in the program.

A variant of symbolic execution is called *selective symbolic execution* (SSE). Selective symbolic execution alleviates state explosion by automatically reducing the amount of code that needs to be executed symbolically [15]. This is done by switching between concrete and symbolic execution depending on the requirements of the analysis. For our data leak detection tool, we require that only execution paths of the application are systematically explored.

An exploration engine that uses selective symbolic execution checks whether execution calls into a method that belongs to the unit — the application of interest — or to the environment. If the method belongs to the unit, the exploration engine systematically explores the method.

It uses symbolic values for each method parameter. Moreover, whenever execution reaches a method from inside a method of the unit, the exploration engine executes it concretely and replaces the concrete return value with a symbolic one. The effect is that execution forks if it reaches a branch, but only when execution is inside the unit.

If the method belongs to the environment (e.g., in Java: `System.out.println()`), this means that it is out of scope of the analysis and does not require exhaustive exploration, i.e. the exploration engine executes it concretely.

Is it feasible to systematically explore all paths of the smartphone application? A study counted the branches of 1100 smartphone applications and found out that 90% of the applications have about 4187 branches, a complexity which has already been managed by path exploration tools [26].

We consider symbolic execution as the most promising technique for our exploration engine. The question of this subsection was to find a way to systematically explore execution paths. We considered three alternatives: static code analysis, random generation of user input, and symbolic execution. We decide to use symbolic execution for our exploration engine. Although former work demonstrated that static analysis is applicable for detecting data leaks in smartphone applications, we believe that using symbolic execution leads to less false positives. In particular, we considered selective symbolic execution, a variant of symbolic execution that alleviates the path explosion problem of symbolic execution.

Data Leak Detection

The task of data leak detection can be framed as a problem of data flow analysis. Data flow analysis studies the propagation of values inside a program.

Similar to the exploration engine, there are two directions, static reachability and data flow analysis, and dynamic taint tracking.

Static reachability analysis answers the following question: Given a graph that represents possible execution paths of a program, is there a path that leads from a source of sensitive data (e.g., a method which retrieves the current location) to a data sink (e.g., a method that transmits data over the network)? A graph that represents possible execution paths is also called a control-flow-graph (CFG). Each node represents a *basic block*, i.e., a piece of code that contains no jumps or jump targets in the middle of the block. Each basic block starts with a jump target and ends with a jump.

Reachability analysis is not enough to accurately detect data flows for a given CFG, because a relation between the data source and the data sink does not necessarily imply that execution propagates the same data along the path. Thus, static reachability analysis is complemented by static data flow analysis.

Although applicable for data leak detection, static analysis suffers from false positives and false negatives. Static reachability analysis and static data flow analysis were used by PiOS to detect data leaks in iOS applicaiton [20]. We have discussed its limitations earlier in section 2.2.

Dynamic taint tracking marks input data and then monitors how the marked data propagates during execution. The process of marking input data is called tainting. Monitoring tainted data requires propagation rules to update the taint tag along an execution path.

Dynamic taint tracking can be used for various data flow analyses at runtime [47]. For example, the detection of software vulnerabilities like buffer overflow [38], the lifetime analysis of data inside an application [29], the generation of test data [54] and finally data leak analysis [21].

The principle of dynamic taint tracking is similar to scintigraphy in nuclear medicine [52]. In scintigraphy, radioisotopes which emit gamma radiation are introduced into the body of a patient. Outside, gamma cameras measure the emitted gamma radiation and track the movements of the 'tainted' isotopes inside the body. This method allows to identify the location of tumors, the location of bone fractures or diagnose the blockage of arteries.

We can combine the principle of dynamic taint tracking with symbolic execution. The symbolic execution engine can also be used for tainting data. Instead of tagging sensitive data separately, one can inject a symbolic value whenever the unit requests sensitive data. For example, when an application calls the application programming interface (API) to receive the current location, a symbolic value with a predefined name is returned. For example, one could give all symbolic values that represent location data the name 'location'. A data leak occurs whenever the transmission observer component detects a symbolic value with a known name in a method from the network API.

Dynamic taint tracking cannot detect data leaks that arise from implicit (or indirect) flow. We modified an earlier example to show a data leak that arise from implicit flow:

```
public int onFieldEdit(int zipcode) {
    m_zipcode = zipcode;
    int day = Calendar.getInstance().get(Date.DAY_OF_WEEK);
    if (day == 1) {
        switch(zipcode) {
            case 1:
                sendOverNetwork(1);
                break;
            case 2:
                sendOverNetwork(2);
                break;
            case 3:
                sendOverNetwork(3);
                break;
            // ...

            case 9999:
                sendOverNetwork(9999);
                break;
        }
    }
 }
```

Data leaks that follow from implicit flows require other methods such as static code analysis to be detected [18]. Such data leaks cannot be detected with dynamic taint tracking because `zipcode` is implicitly derived from the control flow of the code. No application programmer would write such code unless she explicitly wants to bypass data leak detection mechanisms. The authors of TaintDroid — which also use dynamic taint tracking — state that the use of implicit flows is already an indicator for malicious intentions, but cannot be detected by dynamic taint tracking [21].

In summary, dynamic taint tracking allows to detect data leaks that follow from explicit flows. One limitation of dynamic taint tracking is that it does not detect data leaks that follow from implicit flows. We decide to use dynamic taint tracking and combine it with symbolic execution. The tracer component inserts symbolic values into the system whenever the unit requests sensitive data. Afterwards, the transmission observer component reports a data leak whenever symbolic values are detected in the network API.

In this chapter, we developed a concrete picture of how to automatically detect data leaks in smartphone applications. First, we identified four requirements which drove the component specification of the tool. We discussed design choices and evaluated advantages and disadvantages of possible techniques. We decided to use symbolic execution to explore all execution paths of the unit. Additionally, we use symbolic values as tainting mechanism to find and report data leaks.

CHAPTER 4

Implementation

This chapter describes the current implementation state of `Leakalizer` and the challenges while building it. First, we give an overview of the implementation. Then we describe used tools and how we modified them. Finally, we sketch what we have built additionally in order to find data leaks with `Leakalizer`.

`Leakalizer` builds upon S2E and the Android emulator. As specified in chapter 3, we wanted a data leak detection prototype based on symbolic execution and dynamic taint analysis to converge to the properties of an ideal data leak detection tool. To achieve this for Android applications, we combine S2E and the Android emulator to diagnose data leaks.

S2E is a platform that uses symbolic execution to analyze whole software stacks at runtime. So far, S2E was only available for the instruction set architecture X86. First, we added support for ARM, an instruction set architecture used for mobile devices and other embedded systems. Then, we integrated the Android emulator into S2E. The intermediate result is a virtual smartphone device that can be analyzed by `Leakalizer`.

Figure 4.1 shows an overview of the implementation. Numbers in parentheses refer to components of `Leakalizer`. The core of `Leakalizer` (1) is built upon the modified S2E platform. The integrated Android emulator (2) boots the Android software stack (3) inside the virtual smartphone and starts the unit (4), i.e., the Android application of interest.

`Leakalizer` diagnoses data leaks in an Android application by using invasive and noninvasive methods. In medical diagnostics, noninvasive methods work from outside the living system, i.e. they do not invade the living body. In contrast, invasive methods require instruments inside the living body to provide diagnostic information.

We implemented the data leak detection mechanism from chapter 3 by splitting the components in a noninvasive host side and an invasive target side. Both sides communicate with each other in order to select execution paths of the unit and to diagnose data leaks.

We implemented the host side (5) in form of `Leakalizer` plugins. Plugins use the developer interface from S2E to exchange messages with the target side, to systematically explore execution paths of the unit, and to find data leaks along an execution path. We have built four `Leakalizer` plugins that cover all components of the design, except the entry point invocator.

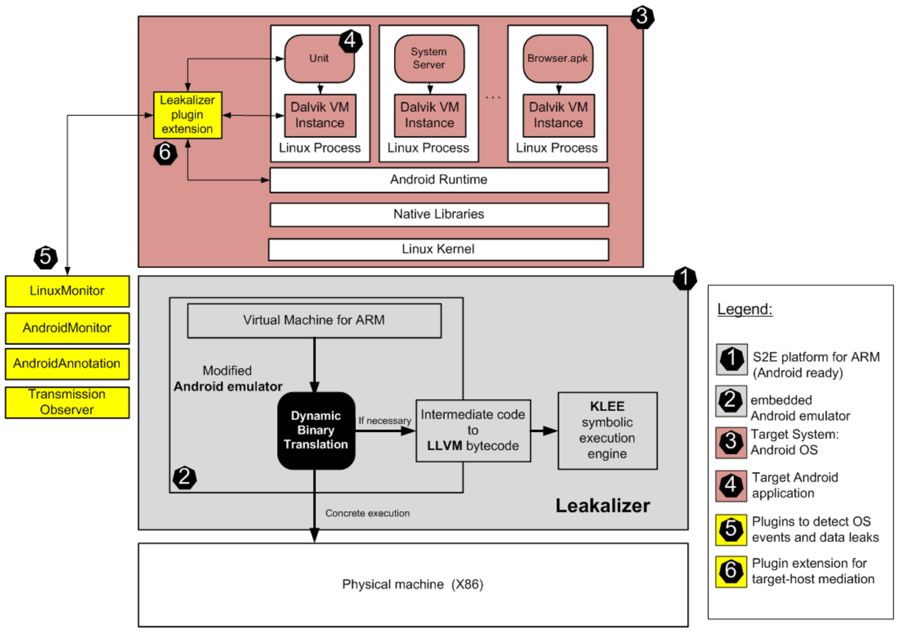

Unit
System Server
Browser.apk
Dalvik VM Instance
Linux Process
Leakalizer plugin extension
Android Runtime
Native Libraries
Linux Kernel
LinuxMonitor
AndroidMonitor
AndroidAnnotation
Transmission Observer
Virtual Machine for ARM
Modified Android emulator
Dynamic Binary Translation
If necessary
Intermediate code to LLVM bytecode
KLEE symbolic execution engine
Leakalizer
Concrete execution
Physical machine (X86)
Legend:
S2E platform for ARM (Android ready)
embedded Android emulator
Target System: Android OS
Target Android application
Plugins to detect OS events and data leaks
Plugin extension for target-host mediation

Figure 4.1: Components of `Leakalizer` — a prototype to detect data leaks in Android applications: (1) S2E, a platform for building analysis tools for big software stacks, (2) an embedded Android emulator to run (3) the Android software stack on a typical desktop computer. (4) One process inside Android runs a Dalvik virtual machine that executes the unit. (5) Plugins outside the target system detect OS events and observe sensitive data. (6) The Leakalizer plugin extension (LPE) inside the target system complements plugins on the host-side to explore possible execution paths of the unit and diagnose data leaks along each execution path.

The target side (6) consists of the Leakalizer plugin extension (LPE) — a mediator module that we have incorporated into the Android software stack. The LPE allows `Leakalizer` to focus on the unit, sources of sensitive data (e.g., the Android Location manager), and data sinks (e.g., the Android network API).

The rest of this chapter describes the main challenges of building `Leakalizer` as follows: Section 4.1 describes the inner working of the Android emulator. Section 4.2 explains S2E, a platform to build multi-path analysis tools by writing plugins. Section 4.3 describe how we modified S2E and the Android emulator in order to boot Android. The intermediate result is a custom virtual smartphone that boots Android and can be observed and controlled from `Leakalizer` plugins. From this ground, section 4.4 describes the challenge of analyzing Android applications and our approach to use a combination of noninvasive `Leakalizer` plugins and an invasive mediator module — the LPE — to detect data leaks caused by a single Android application. Finally, section 4.5 describes the four `Leakalizer` plugins.

4.1 The Android Emulator

The Android emulator allows Android application developers to test and debug their applications without the need to use their own phones (Figure 4.2).

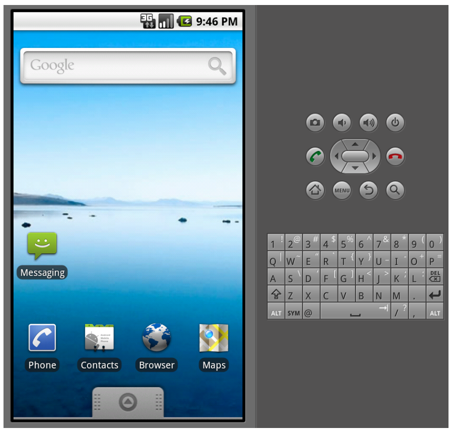


Figure 4.2: Screenshot of the Android emulator that emulates a virtual smartphone device to run the Android software stack on PC.

The Android emulator uses the full system emulator QEMU to run virtual machines in a process on the host machine. We first discuss QEMU and then the specifics of the Android emulator.

Dynamic Binary Translation with QEMU

QEMU is a full-system-emulator written in C. A *full system emulator* emulates target systems (CPU and periphery), in a process on the host system. In QEMU, target systems can have different instruction set architectures and use different hardware than the host system.

For example, in order to run a smartphone application on PC, QEMU needs to execute ARM instructions and to emulate the hardware of the smartphone. A typical PC has an X86 (or AMD64) processor. Smartphones often have a system-on-a-chip with an ARM processor and various sensors. QEMU translates ARM instructions into X86 instructions to emulate a smartphone environment.

QEMU uses *dynamic binary translation* (DBT) to translate ARM instructions into X86 instructions [7]. Dynamic binary translation translates a set of instructions into a translation block. A *translation block* consists of the intermediate representation of a sequence of instructions. The last instruction of the translation block changes the control flow. For example, consider the following four ARM instructions[1]:

```
0x0000810c:  mov        r0, r1
```

[1]Refer to the ARM instruction set references to get a list of all instructions along with their exact effects on CPU state, memory and periphery [2].

```
0x00008110:  mov        r2, #28672       ; 0x7000
0x00008114:  cmp        r0, r2
0x00008118:  bne        0x8000
```

The first two ARM instructions of the translation block modify register values and the third one modifies the program state register. All three instructions do not modify the control flow of the program. The fourth instruction is a branch instruction. Depending on the last comparison, it jumps either to `0x8000` or to `0x811c`. This means, that the first instruction of the next translation block is either `0x8000` or `0x811c`. In this example, the value of `r1` at runtime determines which translation block comes next.

Because QEMU uses a dynamic translation process, the value of `r1` does not need to be inferred with static methods. Whenever execution reaches the translation block from above, `r1` is already stored in the CPU state. Because QEMU emulates the behavior of a real ARM CPU, the CPU state is stored in a data structure and updated along execution.

Let us assume, that `r1` is `0x6000`. The comparison updates the condition flags of the program state register. Therefore, the condition (NE) is fulfilled and the branch instruction is executed. Finally, the program counter is set to `0x8000` and the next executed instruction is read from `0x8000`.

QEMU implements DBT in two steps. First, QEMU translates a sequence of instructions into an intermediate representation that consists of micro-operations. Then, QEMU translates the intermediate representation into host instructions.

In the first step, QEMU translates a sequence of guest instructions in an intermediate representation. This step is target-dependent, i.e., is different for ARM-targets and X86-targets. A disassembler reads one instruction after the other from the memory to detect the instruction type. When it detects the instruction, it calls helper functions that generate the corresponding micro operations. The disassembler continues until all instructions of the sequence are translated into micro-operations.

For the second translation step, QEMU uses the built-in-compiler TCG (Tiny Code Generator) that compiles micro-operations into host instructions. TCG translates the sequence of micro-operations of a translation block into instructions that can be executed by the host CPU. This step is host-dependent, i.e., depends on the instruction set architecture of the physical machine where QEMU is running.

When TCG is finished, QEMU stores the result in a cache. The cache avoids translating the same blocks again and again. Translation only starts when QEMU discovers a new translation block during execution. Finally, the host CPU executes the result.

Specifics of The Android Emulator

There are three major differences between vanilla QEMU and the Android emulator: (1) The front-end for Android SDK integration, (2) the virtual system-on-a-chip called *Goldfish* and (3) the multiplexing daemon *Qemud*.

The emulator adds a front-end to integrate into the SDK, especially to interact with the Android Virtual Device Manager (AVD). AVD is a graphical user interface to configure and start a virtual smartphone device [3]. AVD passes the user-defined configuration to the new front-end. The front-end then transforms the configuration into a sequence of parameters that are passed to QEMU. For example, AVD can command QEMU to start virtual device which has 256MB memory instead of 128MB.

Figure 4.3 shows the most important devices that make up goldfish. Goldfish includes an ARM CPU, which supports ARMv5TE instructions and the necessary periphery (e.g., the real time clock, NAND flash, battery, and multimedia card device). Most of the Goldfish devices are not included in vanilla QEMU.[2]

`Qemud` is a multiplexing daemon which enables communication between the emulator and the emulated Android system over a serial port. The main elements are: The `qemud` multiplexing daemon, `qemud` clients and `qemud` services (see Figure 4.4). The daemon and the clients run inside the emulated Android system. The daemon process named `qemud` is started during the boot process. A `qemud` client is any part inside the Android system that wants to communicate with the emulator.

The purpose of `qemud` is to control parts of the emulator from inside the Android system, e.g., change the intensity of the emulated LCD backlight, retrieve a list of available sensors, setup boot properties during the boot process or send AT commands to the emulated modem. `Qemud` services are listening for client messages. If a service receives a message that complies with the protocol, the service executes the command and the client receives an answer. For example, when `qemud` receives the command "temperature:25" from a `qemud` client, it sets the current temperature of the temperature sensor to 25°. In the documentation, the developers of the Android emulator state that `qemud` saves them from writing additional kernel drivers for the target and saves them from writing additional hardware emulation code. The documentation also includes the communication protocol between multiplexing daemon and services [4].

4.2 Selective Symbolic Execution with S2E

S2E is a platform that implements selective symbolic execution. In section 3.3, we already introduced selective symbolic execution as a variant of symbolic execution that mitigates the path explosion problem for a particular analysis task by automatically reducing the amount of code that is executed symbolically.

S2E is built on QEMU and KLEE, which is shown in figure 4.5. S2E instruments QEMU for dynamic program analysis. Dynamic program analysis allows to analyze software at runtime. S2E analyzes the software that runs inside the virtual machine. S2E incorporates KLEE, a symbolic execution engine which was originally built to automatically generate test cases [10]. If a translation block needs to be executed symbolically, S2E first translates it into the LLVM assembly language [35] and then passes the result to KLEE which performs symbolic execution. To switch between symbolic and concrete execution, KLEE and QEMU use the same states. Details can be found in the original papers [14] [15].

An analysis tool that is built on top of S2E consists of two types of plugins, analysis plugins and selector plugins. Selector plugins define the scope of the analysis, i.e. they split the set of possible execution paths into interesting paths — i.e., paths that belong to the unit — and other

[2]There are ongoing efforts to port parts of the Goldfish devices to vanilla QEMU. [28]

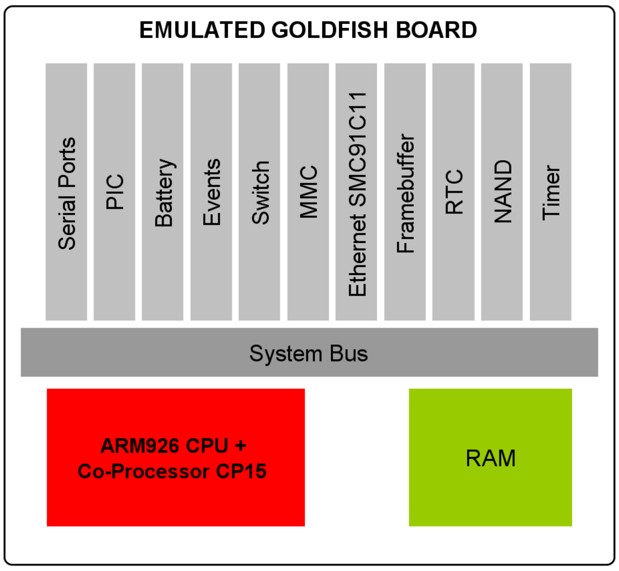


Figure 4.3: Overview of the Goldfish board (periphery and ARMv5 CPU)

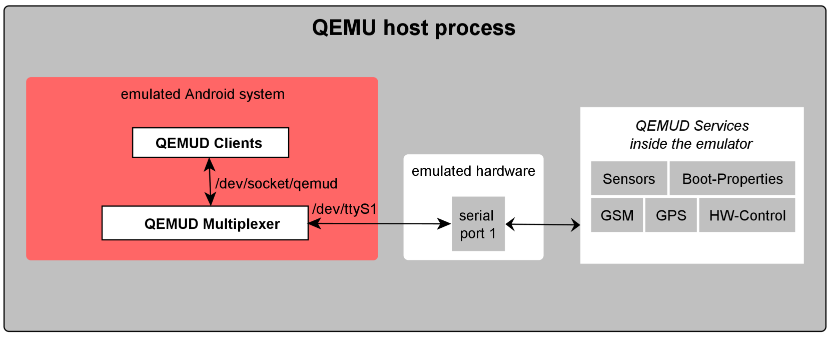


Figure 4.4: Qemud communication scheme between emulated Android system and emulator program used to control parts of the emulated hardware

paths — i.e., paths that belong to the environment. Thus, selector plugins implement *selective* symbolic execution. Analysis plugins check whether certain properties hold for explored paths, e.g., whether a path leaks sensitive data. Both types of plugins have access to the S2E application programming interface (API). The API provides low level access, i.e., access to memory, CPU registers, interrupts, system events, that execution reached an instruction at a particular program counter, etc.

Plugins can emit events to communicate with other plugins. This way, plugins can build up higher level events — e.g., the launch of an application — by consuming and processing events from other plugins.

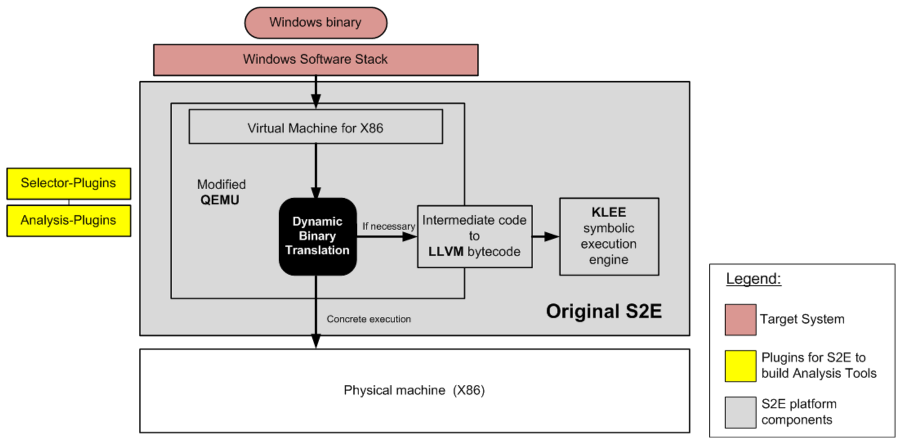


Figure 4.5: The architecture of S2E incorporates the emulator QEMU and the symbolic execution engine KLEE

4.3 Port S2E to ARM

The core of the exploration engine of `Leakalizer` is a modified S2E platform. We modified parts of S2E to boot an Android software stack inside S2E. Although there is a X86 port for Android, smartphones (and other embedded systems) are usually built around an ARM processor.

To boot Android with S2E, we ported S2E to ARM. Afterwards, we embedded the Android emulator into S2E. Overall, we conducted the following modifications which are further described in the next subsections: (1) S2E only adapted the X86-specific part of the dynamic binary translation process of QEMU. We adapted the ARM-specific part to communicate with the S2E core. (2) We modified the ARM-specific CPU state in QEMU. QEMU models the state of the CPU as a data structure in C. S2E only adapted the target-independent part and the X86-specific part of the CPU state and not the one for ARM CPU's. (3) We changed the core of S2E to support both X86 and ARM. In general, S2E supports other instruction set architectures, because the core of S2E does not require a particular instruction set. Some parts, however, *assume* X86. We changed these parts to support both X86 and ARM. (4) We adapted common CPU boards for ARM processors (Integrator- and Verbatim-boards) to give S2E full access to CPU and RAM. (5) We ported some of the already existing plugins for X86 to ARM. This enables ARM code inside the virtual smartphone to communicate with host-side plugins and the other way around. (6) We integrated the Android emulator into S2E to support Goldfish, boot Android and launch Android applications.

ARM support for S2E was tested by successfully booting an ARM Debian Linux and symbolically execute a small native ARM binary on it.[3] Since the V1.1 release of S2E, the ARM port of S2E is publicly available.

[3]The Symbolic Maze is an example to demonstrate the power of symbolic execution [37].

Control the state of a virtual ARM machine

We adapted the ARM-specific target of QEMU to give S2E control over the virtual machine at runtime. Although QEMU currently supports X86, ARM, MIPS, M68K and PowerPC as target systems out of the box, S2E only supports X86. S2E ships with vanilla code of the ARM-target of QEMU but does not instrument it to control the state of the virtual machine.

S2E gives plugins control over CPU and RAM of the emulated system. For example, they can read/write the value of a register or the value of a memory address. To achieve this, the emulated target CPU and target RAM need to be registered to S2E when the system is initialized by QEMU. When QEMU starts full-system emulation it allocates memory for the target CPU and the target memory and resets the state of the CPU to an initial state. At this point, S2E hooks the initialization routine to access target CPU and target RAM. We added hooks to two common ARM boards, the Integrator- and the Verbatim-board to control the state of the virtual ARM machine.

ARM CPU state

We analyzed the ARM CPU state structure and split it into symbolic and concrete regions, which is shown in figure 4.6. QEMU stores the state of the CPU (e.g., the value of registers) in a data structure. The data structure was rearranged to have a clear border between symbolic and concrete regions.

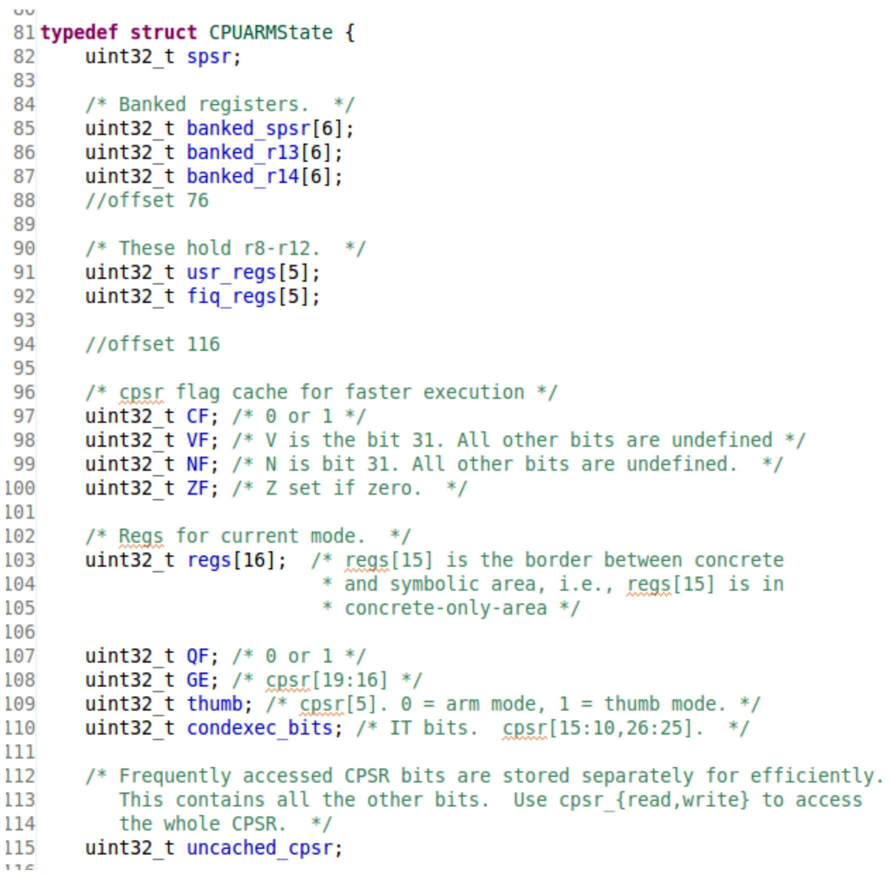
In S2E, registers can hold symbolic values, but some registers need to be always concrete. One example is the program counter (`r15`) that stores the address of the instruction which is executed next. If the address was symbolic, any address could contain the next instruction. As consequence, S2E needs to massively fork the execution after each instruction which is an unmanagable state explosion.

On the other hand, other fields of the CPU state need to be inside the symbolic area, e.g., the condition flags. The condition flags C (carry), V (overflow), N (negative) and Z (zero) influence the control flow for conditional instructions. In ARM, nearly every instruction can become a conditional instruction by using the conditional field. The condition needs to be satisfied in order to execute the instruction. A condition is satisfied if the condition flags are in a certain state. Consider the following sequence of ARM instructions:

```
0xab142460:  mov r7, #100  ; write 100 into r7
0xab142464:  mov r1, #200  ; write 200 into r1
0xab142468:  cmp r7, r1    ; set condition flags accordingly
0xab14246c:  bge 0xab1424b0 ; branch if r7 greater or equal r1
```

The ARM assembly code above is an example to show why the condition flags need to be inside the symbolic area. The instruction `cmp` (compare) updates the condition flags according to the result of subtracting `r1` from `r7`. Instruction `b` (branch) is only executed when the condition flags satisfy the condition `ge` (greater or equal).[4] Assuming that `r7` contains a symbolic value and that condition flags are in concrete-only area, S2E would concretize symbolic values before

[4] Consult the ARM Architecture Reference Manual for details [2].



```c
typedef struct CPUARMState {
    uint32_t spsr;

    /* Banked registers.  */
    uint32_t banked_spsr[6];
    uint32_t banked_r13[6];
    uint32_t banked_r14[6];
    //offset 76

    /* These hold r8-r12.  */
    uint32_t usr_regs[5];
    uint32_t fiq_regs[5];

    //offset 116

    /* cpsr flag cache for faster execution */
    uint32_t CF; /* 0 or 1 */
    uint32_t VF; /* V is the bit 31. All other bits are undefined */
    uint32_t NF; /* N is bit 31. All other bits are undefined.  */
    uint32_t ZF; /* Z set if zero.  */

    /* Regs for current mode.  */
    uint32_t regs[16];  /* regs[15] is the border between concrete
                         * and symbolic area, i.e., regs[15] is in
                         * concrete-only-area */

    uint32_t QF; /* 0 or 1 */
    uint32_t GE; /* cpsr[19:16] */
    uint32_t thumb; /* cpsr[5]. 0 = arm mode, 1 = thumb mode. */
    uint32_t condexec_bits; /* IT bits.  cpsr[15:10,26:25].  */

    /* Frequently accessed CPSR bits are stored separately for efficiently.
       This contains all the other bits.  Use cpsr_{read,write} to access
       the whole CPSR.  */
    uint32_t uncached_cpsr;
```

Figure 4.6: Code fragment from the data structure which store the ARM CPU state. The first fields until regs[14] are allowed to hold symbolic values. regs[15] is the first item inside concrete-only area.

it updates the flags. Therefore, execution state will not fork into two states at the branch instruction. But forking into two states is expected behavior when `r7` is symbolic. Thus, condition flags need to be in the symbolic area.

Inside the ARM CPU state, we formed two contiguous areas, one symbolic and one concrete area, and changed the order of the fields in the structure accordingly. In hardware, condition flags are part of CPSR (current program state register), but the structure in QEMU stores each condition flag separately as 32 bit unsigned integer for faster access and stores other contents of CPSR in the array `uncached_cpsr`. This makes it easy to form one contiguous block of fields which can hold symbolic values and another block of fields which can only hold concrete values. As figure 4.6 shows, it was not possible for the `regs` array, which represent the registers `r0-15`. `r15` is the program counter and needs to be always concrete to continue execution at a certain and concrete address. Thus, we moved the array to the end of the symbolic area and draw the border between `regs[14]` and `regs[15]`.

The border between symbolic and concrete area is defined in S2E's execution state. The symbolic area of the CPU state has a special treatment, because it does not only use allocated space from QEMU. Instead, S2E uses special memory objects which are capable of storing and handling symbolic values. The symbolic execution engine KLEE provides these memory objects. QEMU can access the symbolic regions through wrappers. All parts of QEMU which access fields inside the symbolic area need to use these wrappers in order to get the current value of these fields.

Conditional compilation for X86 and ARM support

We adapted S2E in order to support both X86 and ARM. For example, consider the following C++ method inside S2E's execution state:

```
 uint64_t S2EExecutionState::getPc() const
{
#ifdef TARGET_ARM
    return readCpuState(CPU_OFFSET(regs[15]),
           8*sizeof(target_ulong));
#elif defined(TARGET_I386)
    return readCpuState(CPU_OFFSET(eip),
           8*sizeof(target_ulong));
#endif
}
```

The method retrieves the program counter from the CPU structure of QEMU. The X86 CPU structure stores the program counter in the field `eip`, the ARM CPU structure in `regs[15]`. To access the right field depending for which target S2E is compiled, we do not want to have different source files. Therefore, we use precompiler macros (e.g., `TARGET_ARM`) and conditional compilation directives (e.g., `#ifdef`).

The build system of QEMU provides precompiler macros that indicate the target for the current compilation (e.g., `TARGET_ARM` and `TARGET_I386`). We globally use QEMU's precompiler macros for S2E to compile target-specific code without having different source files.

Port existing S2E plugins to ARM

We ported already existing S2E plugins to ARM whenever needed for building `Leakalizer` on top of it. So far, S2E was used for building various analysis tools for X86 targets [15]. Therefore, most of the existing plugins for S2E are written for X86. Moreover, many plugins analyze the Windows platform. We now describe how we ported the BasicInstruction plugin to ARM.

S2E allows target code to communicate with S2E plugins over custom opcodes. A custom opcode is an instructions of the guest system that is directly interpreted by S2E and is not translated into a host instruction. First, S2E calls `s2e_tcg_emit_custom_instruction()` in the target-dependent translation step, whenever it discovers an instruction with S2E's custom opcode. Then, S2E invokes a custom instruction handler. Finally, S2E notifies the BasicInstruction plugin about a custom opcode which can retrieve the operands and perform the corresponding action.

We ported communication with S2E plugins over custom opcodes to ARM. First, we chose an opcode that is not used by other ARM instructions according to the ARM instruction set reference [2]. Then we had to modify QEMU's dynamic binary translation engine accordingly to invoke the custom instruction handler whenever the new opcode appears in an instruction. Finally, we ported the BasicInstruction plugin to catch all operands.

The result is that the target ARM code can communicate with S2E to retrieve the S2E version, enable/disable forking of execution paths, retrieve the path id of current execution path, log messages, print a symbolic expression, insert a symbolic value, print memory contents, concretize a symbolic expression, etc.

Integrate Android Emulator Into S2E

`Leakalizer` exploits the fact that both the Android emulator and S2E are based on QEMU to analyze Android applications. Therefore, we can use the Android emulator instead of the vanilla QEMU to boot Android.

To integrate the Android emulator into S2E, we added hooks to the code base of the Android emulator. This enables `Leakalizer` and the plugins to control the state of the virtual smartphone at runtime. In most cases, the hooks for the Android emulator were similar to the hooks that S2E applied to control QEMU. We identified the changes and applied it to the Android emulator code base accordingly.

The result is that `Leakalizer` is able to boot Android, run Android applications and analyze the system. From outside, i.e. from the host system, `Leakalizer` can observe the current state of the target CPU and the target memory.

`Leakalizer` takes advantages of the snapshot mechanism of QEMU to speed up development cycles. Booting Android in `Leakalizer` takes longer compared to the original Android emulator. Thus, we first boot with the original Android emulator, store a snapshot of the current state at the local file system of the host system and load the snapshot in `Leakalizer` to continue a running Android system.

4.4 Analysis of Android Applications With Leakalizer

In the former sections of this chapter, we prepared the ground for detecting data leaks with `Leakalizer`. The result was a custom virtual smartphone that waits to be analyzed by plugins using the plugin infrastructure of the S2E platfrom. In other words, we have to write plugins on top of S2E that detect data leaks. We call those plugins *Leakalizer plugins*. As described earlier, the plugin infrastructure of S2E allows to write selector plugins — to select and explore execution paths of interests — and analysis plugins — to check for properties along execution paths (cf. 4.2).

This section first describes how to write `Leakalizer` plugins that analyze interpreted bytecode of a single Android application. This is challenging because the instruments of S2E are tailored to analyze native instructions, not interpreted code. Then, we describe how we approached this challenge by adding a module inside the Android system — the Leakalizer plugin extension

(LPE) — that gives `Leakalizer` plugins more control over the Android system, especially the Android runtime environment.

How to analyze interpreted bytecode?

Based on the design proposed in chapter 3, `Leakalizer` requires to distinguish between unit and environment. To be more precise, the execution engine needs to systematically explore execution paths that belong to the target Android application (the unit) and executes the rest of the Android software stack (the environment) concretely to mitigate the path explosion problem (cf. section 3.3).

Java applications consist of Java bytecode. Developers write their applications in the Java programming language. The code is compiled into Java bytecode. Finally, the *Java virtual machine* (JVM) interprets the byte code and executes the application. That means that Java applications are never compiled to native code.

Android applications consist of Dalvik bytecode. Application developers also write their applications in the Java programing language and again the Java source code is compiled into Java bytecode. But additionally, the Java bytecode is transformed into Dalvik byte code. The Dalvik bytecode is similar to the Java bytecode but optimized for mobile devices [45].

Interpreting Dalvik bytecode requires a virtual machine called the *Dalvik virtual machine* (DVM). Like a JVM it decouples applications from the rest of the system, which makes Android applications hardware-independent and to some extent more secure.[5]

The Android software stack houses multiple DVMs. Each Android application runs in a separate process with its own DVM. Since `Leakalizer` uses a virtual machine to analyze the Android software stack, we have the situation that multiple virtual machines are running inside a virtual machine. This situation is shown in figure 4.7. The Android emulator — that we have integrated into `Leakalizer` — emulates a virtual smartphone that is capable of booting Android. Android is based on a modified Linux kernel and runs multiple Linux processes. Some of the processes run a DVM to execute Dalvik bytecode from Android applications.

There is one DVM inside Android that interprets bytecode from the unit, i.e., the application to be analyzed. We call a DVM which interprets unit code *inner environment*, because it is tightly coupled with the unit. The rest of the environment is called *outer environment*. In summary, the Android system consists of a unit (the Android application of interest), inner environment (DVM) and outer environment (rest of the system). The outer environment includes other DVMs that interpret bytecode of other applications. They belong to the outer environment, because they are not within the scope of the analysis, i.e., we only want to find data leaks that are caused by one singular Android application.

A DVM is like an organelle in an eukaryotic cell. The tight coupling between the inner environment and the unit can be understood with the model of a biological cell in cell biology. We already used the model of a cell to explain entry points (cf. section 3.2). A cell organelle is a subunit of the cell that has a specific function. It is surrounded with a membrane to highly control interactions with the outer environment, i.e., the cytoplasm. Staying with the model, we are interested in the processes inside one organelle. We could take it out of the living system,

[5]According to various studies, the level of security is still not satisfying [33] [44] [22].

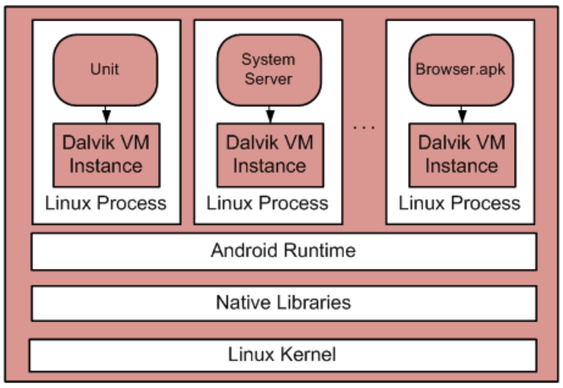


Figure 4.7: The target system of `Leakalizer` is the Android software stack which runs multiple instances of the Dalvik virtual machine of which one interprets the code of the unit.

but then we cannot study the interactions between the inner and the outer environment anymore, which is important for a thorough analysis.

The state and the processes inside the DVM are hard to focus from outside, especially when our instruments only measure low level processes. The execution of bytecode inside the DVM cannot be easily observed at native instruction level. From the point of view of the S2E platform, software is a sequence of native ARM instructions which modifies registers and memory. This approach suits well to analyze software like device drivers or kernel code. It also suits to analyze applications which are compiled into native ARM instructions. At the same time, it is hard to study the behavior inside a single DVM, because the abstraction level is different.

The diagnostic instruments that build the core of `Leakalizer` (i.e., the modified S2E platform) work at a different abstraction level than the unit code. Bytecode is not directly compiled into native ARM instructions but interpreted from a bytecode interpreter. The interpreter of Dalvik bytecode stands between the diagnostic instruments of S2E and the application logic of the unit, because the diagnostic instruments of S2E measure how sequences of ARM instructions affect the CPU state and memory and not how Dalvik bytecode of the unit affects the state of the DVM.

We tackle the problem by developing `Leakalizer` plugins at the host side that measure the overall state of the system. In addition to that, some of the plugins exchange information with the Android system by using a mediator module. The mediator module — incorporated into the Android software stack — communicates with plugins in order to detect data leaks. The mediator module is called Leakalizer Plugin Extension (LPE).

In summary, we implemented the components of chapter 3 by writing host-side plugins based on S2E's plugin infrastructure and the target-side mediator module LPE. The LPE communicates with the `Leakalizer` plugins via custom ARM instructions (cf. section 4.3). Figure 4.8 gives an overview of the interplay between plugins and LPE. At the host side, we implemented plugins that observe and manipulate the system from outside (1). The host-side plugins are extended by the `Leakalizer` plugin extension (LPE) inside the Android system (2,3).

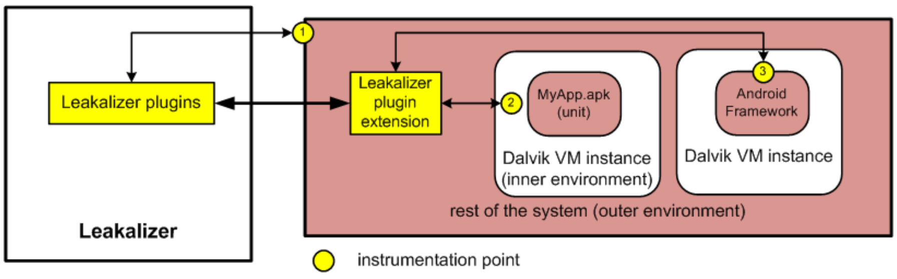

Leakalizer plugins
Leakalizer
1
Leakalizer plugin extension
2
MyApp.apk (unit)
Dalvik VM instance (inner environment)
3
Android Framework
Dalvik VM instance
rest of the system (outer environment)
instrumentation point

Figure 4.8: Interplay between `Leakalizer` plugins at the host side and the Leakalizer plugin extension (LPE) inside Android. While `Leakalizer` plugins alone focus the system as a whole (1), LPE allows to focus the inner environment (2), parts of the Android framework (3), or other aspects.

Leakalizer Plugin Extension (LPE)

LPE is a package of libraries that establishes communication with `Leakalizer` plugins at the host side. The purpose of the LPE is to measure and control the Android system, especially the inner environment which is hard to focus from outside. For example, LPE is used to trace Dalvik instructions of the unit and to filter out bytecode interpretation for all other DVMs.

Communication between the Android system and `Leakalizer` plugins is established via *instrumentation points*. An instrumentation point is a modification of the Android code base that uses LPE to help `Leakalizer` to focus on specific aspects of the Android system. While `Leakalizer` plugins alone focus on the whole Android system, LPE can focus on the inner environment or the network interface or the location manager inside the Android system. Instrumentation points can be added everywhere in the Android system and at different abstraction levels.

LPE allows to add instrumentation points at three levels of abstraction, which is shown in figure 4.9: (1) inside Java code, (2) inside C/C++ code, and (3) inside assembly code. Java methods use a class called `S2ECommands` to talk to host plugins, C/C++ code include a header file, and assembly code invokes custom S2E instructions directly (cf. section 4.3).

For example, we added an instrumentation point for the class `ActivityThread`. Whenever a new application is launched, an instance of `ActivityThread` calls a static method of `S2ECommands` to notify the `Leakalizer` plugin AndroidMonitor about the start of an Android application. AndroidMonitor compares the name of the application with the user-defined unit name in the configuration file. If they match, AndroidMonitor notifies LPE to focus on this application. Then, LPE sets the boolean `s2ehooks` — which is inside the global state of the DVM — to true. `Leakalizer` uses the new state information to trace the execution of unit code and to tag sensitive data only if the unit requests it. A lot of other usage scenarios are possible. In general, `s2ehooks` is a convenient way to distinguish between the unit and other Android applications inside the system which we count as environment.

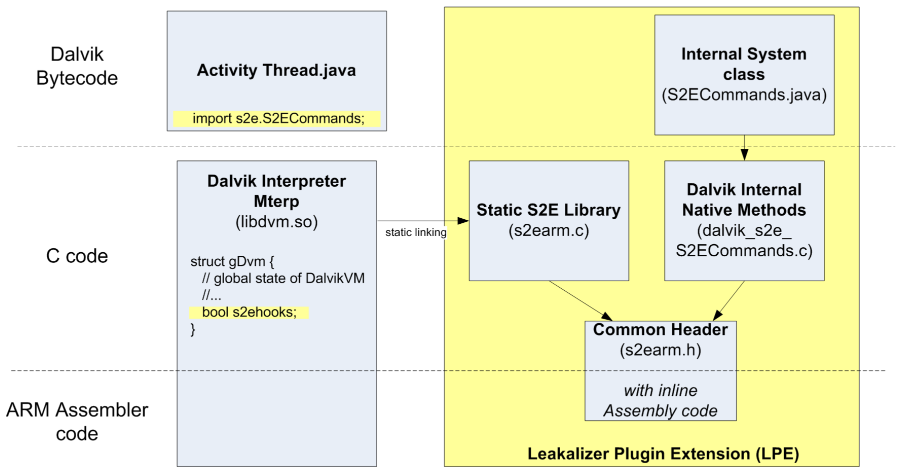

Dalvik Bytecode
Activity Thread.java
import s2e.S2ECommands;
Internal System class (S2ECommands.java)
C code
Dalvik Interpreter Mterp (libdvm.so)
struct gDvm {
// global state of DalvikVM
//...
bool s2ehooks;
}
static linking
Static S2E Library (s2earm.c)
Dalvik Internal Native Methods (dalvik_s2e_ S2ECommands.c)
Common Header (s2earm.h)
with inline Assembly code
ARM Assembler code
Leakalizer Plugin Extension (LPE)

Figure 4.9: The Leakalizer Plugin Extension (LPE) establishes communication between Android and `Leakalizer` plugins at three abstraction levels, Java code, C/C++ code and assembly code. The left side shows two modules of the Android software stack that use LPE.

4.5 Data Leak Detection Plugins of Leakalizer

We implemented the detection of data leaks with `Leakalizer` plugins and LPE. `Leakalizer` includes a basic implementation for all components except the entry point invocator.

Exploration Engine

The implementation of the exploration engine consists of three plugins: LinuxMonitor, AndroidMonitor and AndroidAnnotation. Moreover, some instrumentation points are added inside the DVM interpreter and the class `ActivityThread` which is involved in the launch of an Android application.

LinuxMonitor provides other plugins like AndroidMonitor with OS-level information, e.g., the current process including informations about loaded libraries, a list of running processes, system events, etc. LinuxMonitor retrieves this information in a non-invasive way, i.e. without LPE. But it requires knowledge about the data structures of the Android kernel.

AndroidMonitor detects the launch of the unit, i.e., a user-defined application of interest. Whenever AndroidMonitor detects the launch of a unit it does two things: (1) It notifies other plugins about it. (2) It enables `Leakalizer`-specific instrumentation of the DVM that launched the application. One effect is that `Leakalizer` supports tracing of Dalvik bytecode instructions of the unit.

AndroidAnnotation provides support for automatic exploration of code in a Java method. The user specifies a list of target methods. Target methods are methods that belong to the unit and that the user wants to be fully explored. Whenever the system calls a target method, AndroidAnnotation replaces the concrete parameters of the method with symbolic values. The consequence is that `Leakalizer` explores all branches that rely on method parameters. Currently the following Java data types are supported: int, float, long, char, double and byte.

Tracer

The implementation of the tracer component consists of instrumentation points inside the Android application framework. For example, we modified the telephony manager, a class that transmits the unique device ID to third party applications. Whenever a third party application requests the device ID, the telephony manager distinguishes between two cases: If the source of the request is the unit, it returns an unconstrained symbolic value with the name `device_id`. Otherwise, the telephony manager returns the concrete device ID. This strategy can be extended for other sources of sensitive data.

Transmission Observer

The implementation of the transmisssion observer component consists of a `Leakalizer` plugin named *TransmissionObserver*. In addition to that, we added instrumentation points inside the class `SocketImplementation` of Android. The instrumentation point notifies TransmissionObserver whenever data is written to a socket and that data contains a symbolic value. TransmissionObserver then checks if the symbolic value represents sensitive data. If it is the case, it sends the details (hostname, port, type of sensitive data) to the reporting component. This strategy can be applied to other network APIs in Android.

Reporting component

`Leakalizer` takes advantage of the logging facility of the S2E platform to report data leaks. Log files collect reported data leaks, information about the injection of symbolic values and the forking of execution states.

This chapter gave insights into our prototype `Leakalizer` — a tool to automatically detect data leaks in an Android application. We use the Android emulator to create a virtual smartphone device and integrate it into the multi-path analysis platform S2E to analyze an Android application inside a 'living' Android software stack. We use the plugin infrastructure of S2E to implement our data leak detection approach of chapter 3. Finally, we wrote the LPE that mediates between host-side plugins and interesting locations inside the virtual smartphone device. LPE helps `Leakalizer` to focus on the unit, sources of sensitive data (e.g., the Android Location manager), and data sinks (e.g., the Android network API).

CHAPTER 5

Evaluation

This chapter evaluates our data leak detection approach based on the prototype of `Leakalizer`. The goal of this thesis is to automatically detect data leaks without requiring the user to install the application on her phone and to be more accurate than existing tools. To achieve this, we designed a data leak detection tool consisting of five components (cf. chapter 3). We decided to use symbolic execution for the exploration engine to systematically explore execution paths of the unit and to combine it with dynamic taint tracking to detect data leaks along an execution path. Based on these decisions, we built a prototype of `Leakalizer` and described it in chapter 4. The following evaluates our approach based on the performance of `Leakalizer`.

In particular, the evaluation covers two questions. (1) Does `Leakalizer` accurately detect data leaks by using dynamic taint tracking along an execution path? (2) Which code coverage can be expected from an exploration engine that uses selective symbolic execution?

We answer each question with one microbenchmark. The benchmark of the first question evaluates whether `Leakalizer` detects leakage of the unique device ID. The focus is on the components tracer, transmission observer, and reporting. The benchmark of the second question targets the exploration engine and measures the code coverage of a method inside the unit.

The test setup consists of `Leakalizer` and an Android application named *Leakalizer Eval App* (LEA). LEA displays the microbenchmarks and performs them with a button click (figure 5.1). The host system, in which we compiled and run `Leakalizer` is based on Ubuntu 10.10.

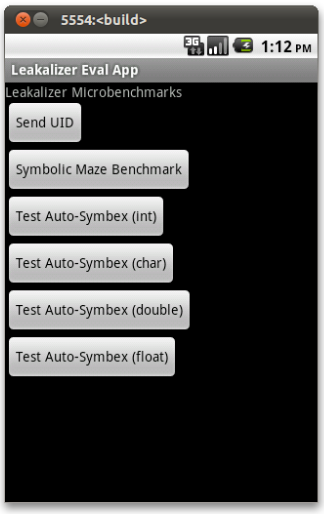


Figure 5.1: LEA, an Android application that performs our benchmarks.

5.1 Data Leak Detection With Leakalizer

LEA contains the following Java method that causes a data leak. LEA calls the method when we press the button `Send UID`.

```
public void onClickSendUid(View view) {
   TelephonyManager tManager = (TelephonyManager)
                    getSystemService(Context.TELEPHONY_SERVICE);
   String uid = tManager.getDeviceId();
   sendToServer(uid);
}
```

The method `onClickSendUid` retrieves the unique device ID and transmits it over network to a server. First, the method retrieves an instance of the telephony manager. The telephony manager is part of the Android application framework and provides access to telephony services and informations. Then, the method retrieves the device ID and sends it to `sendToServer` that creates a socket connection to the host 10.0.2.2 at port `6667` and transmits the device ID.[1]

We expect `Leakalizer` to detect the transmission of the device ID. To verify that, we invoke the entry point that eventually calls `onClickSendUid`. Since the entry point invocator is not implemented yet, we can either press the button manually or use Monkey, an application exerciser that generates random user events, e.g. a button click [1]. We press the button manually.

When LEA calls `getDeviceId()`, the telephony manager returns a “tainted string”, i.e. a modified object of `java.lang.String` that contains a symbolic value. The symbolic value is

[1]The IP `10.0.2.2` inside the virtual smartphone points to `localhost` of the host system, i.e., the physical machine that runs `Leakalizer`.

named deviceid to signify the type of sensitive data. The tracer component of Leakalizer hooks the telephony manager. Whenever the unit — and only the unit — requests the device ID, it returns a “tainted string“.

Before sendToServer sends the string to the destination, the transmission observer component notifies a data leak to the reporting component which then adds the following entry into the log file: ”Data leak of type ’deviceid’ detected. Data leak destination is 10.0.2.2 at port 6667.“ The log entry appeared three times during execution, because Leakalizer had to fork the execution path two times. This is because our tainting mechanism uses symbolic values to taint the string. Whenever a branch relies on a symbolic value, Leakalizer follows both paths (if former constraints on the path do not predetermine a decision).

Our benchmark showed that Leakalizer detects leakage of the unique device ID along one execution path. The Android application does not need to be annotated or modified in any way. The components of Leakalizer — tracer, transmission observer and reporting component — tag sensitive data, detect its transmission and report the data leak to the user with information about destination and type of sensitive data.

5.2 Systematic Exploration of Execution Paths With Leakalizer

Now we want to assess the exploration engine of Leakalizer. In particular, we expect the exploration engine to automatically explore all possible execution paths of the following Java method that is part of LEA:

```
private static void testAutoSymbexInts(boolean ok, int x, int y) {
    if (ok) {
        if (x == y) {
            S2ECommands.killState(0, "(int) ok: x == y");
        } else {
            S2ECommands.killState(1, "(int) ok: x != y");
        }
    } else {
            if (x == y) {
                S2ECommands.killState(2, "(int) !ok: x == y");
            } else {
                S2ECommands.killState(3, "(int) !ok: x != y");
            }
    }
}
```

The method testAutoSymbexInts has four possible execution paths. To be precise, the method has four possible execution paths at the abstraction level that focuses on Java source code. Java source code is compiled to Dalvik bytecode and interpreted at runtime. Only the bytecode interpreter generates machine instructions that the CPU executes. The sequence of machine instructions also contains code that do not belong to the application logic but arise from the interpretation itself. Thus, additional execution paths may be possible, depending on how the interpreter is implemented. For this evaluation, we are only interested in statements and execution paths that belong to the application logic at the Java abstraction level.

We expect the exploration engine to achieve 100% basic block coverage. Basic block coverage measures the proportion between the number of actually explored basic blocks and the number of all basic blocks in the target code. A basic block is any sequence of non-branching instructions.

In our benchmark, we have four basic blocks. Each basic block has only one statement that is a `Leakalizer` annotation provided by our mediator module LPE (cf. section 4.4). `S2ECommands.killState` tells `Leakalizer` to terminate the current execution path and to log this event with a particular message. We can verify that `Leakalizer` has explored all four basic blocks by checking the log file. If the log file contains all four termination messages, the exploration engine has achieved 100% basic block coverage.

Without the exploration engine, we could achieve 100% basic block coverage by manually calling the method four times, e.g., with the following input: $(true, 1, 1)$, $(true, 1, 0)$, $(false, 1, 1)$, $(false, 1, 0)$. Each input triplet covers one basic block.

To systematically explore `testAutoSymbexInts`, we add the unique description name of the method to the configuration file of `Leakalizer`. The specification in the configuration file is necessary, because the entry point invocator is not yet implemented yet for `Leakalizer`. Hence, the user has to inform the exploration engine about methods of interest.

According to the log file extract below, `Leakalizer` achieved 100% basic block coverage. First, the exploration engine injected three symbolic values — one for each parameter. Then, exploration engine forked execution three times to get in total four states. Finally, for each state, the annotation statement was executed, `Leakalizer` terminated the state and creates the corresponding message in the log file.

```
[State 1] Killing state 1
[State 1] Terminating state 1 with message 'State was terminated by opcode
            message: "(int) z: x == y"
            status: 0'
[State 1] Switching from state 1 to state 0

...

[State 0] Killing state 0
[State 0] Terminating state 0 with message 'State was terminated by opcode
            message: "(int) !z: x == y"
            status: 2'
[State 0] Switching from state 0 to state 3

...

[State 3] Killing state 3
[State 3] Terminating state 3 with message 'State was terminated by opcode
            message: "(int) !z: x != y"
            status: 3'
[State 3] Switching from state 3 to state 2

...

[State 2] Killing state 2
[State 2] Terminating state 2 with message 'State was terminated by opcode
            message: "(int) z: x != y"
            status: 1'
All states were terminated
```

Moreover, we modified `testAutoSymbexInts` to test whether exploration engine works for other datatypes and also achieved 100% branch coverage. We replaced the data type of the parameters x and y with other primitive datatypes, `char`, `float`, and `double`. For `char`, we achieved the same coverage with four states. For `float` and `double`, the exploration engine achieved the same coverage but forked 47 and 85 states, respectively, due to the way how floating point values are represented and interpreted in Dalvik bytecode.

In summary, our benchmarks demonstrate that `Leakalizer` detects a data leak and is able to achieve high branch coverage for a given method in an Android application. We further discuss our results in the next chapter.

CHAPTER 6

Discussion

We now discuss advantages and disadvantages of our solution `Leakalizer`.

Our approach leads to less false negatives compared to existing approaches. In our prototype, we used selective symbolic execution to achieve this. The benchmark results (cf. section 5.2) show that `Leakalizer` achieved basic code coverage of 100% by systematically explore possible execution paths of simple methods. Although `Leakalizer` has not yet been tested for bigger Android applications, our prototype suggests that a tool that implements the components of our design will lead to fewer false negatives than existing approaches.

`Leakalizer` is independent from the user's phone. `Leakalizer` emulates a virtual smartphone and detects data leaks inside that *living* system. Hence, the user can first read the data leak report and decide whether to install the application on her phone afterwards. In contrast, TaintDroid requires the user to modify the Android software stack on her physical phone, install the application and gets notified after the application leaked her personal data.

Our approach is also suitable for application developers that want to know whether some type of sensitive data can leak out of their application. `Leakalizer` does not require but allows to annotate Android applications to detect leaks of specific types of data by writing instrumentation points that use LPE (cf. section 4.5).

One limitation of our approach is that dynamic taint tracking focuses on data leaks that follow from explicit flow. Dynamic taint tracking does not capture data leaks that follow from implicit flow. We have discussed this in section 3.2.

Our prototype has the following limitations that arise from the fact that we have not fully implemented all five components specified in chapter 3. First, there is only a basic implementation of the tracer that only supports the detection of two types of sensitive data, device ID and current location. However, our mediator module LPE (cf. section 4.4) allows to add instrumentation points everywhere in the Android software stack, e.g., to add support for additional types of sensitive data. This way, the core of `Leakalizer` can also be used for other analyses of Android applications than data leak detection. Second, we have not implemented the entry point invocator that invokes all entry points of the unit. Hence, the user needs to specify the methods of interest. Third, the exploration engine works with simple data types and not yet with objects.

Therefore, our prototype is not ready to detect data leaks for all Android applications out of the box. Finally, our implementation does not take into account that Android applications can store sensitive data in an internal database and leak it later. In other words, our tainting mechanism currently does not consider data from the database as source of sensitive data.

In summary, benchmarks with our prototype `Leakalizer` suggest that combining symbolic execution and dynamic taint analysis is viable to automatically detect data leaks. This approach leads to less false positives compared to existing approaches. Moreover, our approach informs the user about transmissions of sensitive data before her own sensitive data is at the mercy of third party applications. However, our approach does not detect data leaks that follow from implicit flows and our prototype needs further improvements to detect more kinds of data leaks with less configuration work on the user side.

CHAPTER 7 Conclusion

In this thesis, we designed and built a prototype of `Leakalizer` — a tool that detects data leaks in smartphone applications without requiring the user to install the application on her phone. `Leakalizer` allows users and organizations to better decide whether the application can be trusted and should be installed on their phones.

The prototype of `Leakalizer` is based on symbolic execution, dynamic taint tracking and a virtual smartphone. The prototype thoroughly explores possible execution paths of an Android application inside the 'living' Android software stack and tracks sensitive data along each path to detect data leaks.

We evaluated `Leakalizer` by using microbenchmarks. `Leakalizer` yields 100% basic code coverage for simple Java methods and correctly diagnoses the transmission of the device ID of the virtual smartphone. The results suggest that a combination of symbolic execution and dynamic taint tracking yields higher code coverage and correctly diagnoses more data leaks — i.e., has fewer false negatives — than tools that rely on concrete user input or that perform static code analysis on only the application of interest.

CHAPTER 8
Appendix

Bibliography

[1] Application Exerciser Monkey. `https://developer.android.com/guide/developing/tools/monkey.html`. Last access: 2011/10/09.

[2] *ARMv5 Architecture Reference Manual*. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html`.

[3] Managing Virtual Devices. `https://developer.android.com/guide/developing/devices/index.html`. Last access: 2011/10/22.

[4] QEMUD for Android. `http://gitorious.org/0xdroid/external_qemu/blobs/3a0c4d9eed9ba76e2744258af212b8c1269a24a5/docs/ANDROID-QEMUD.TXT`. Last access: 2011/10/22.

[5] The Android Emulator. `https://developer.android.com/guide/developing/tools/emulator.html`. Last access: 2011/10/09.

[6] M. Bates and R. Weischedel. *Challenges in Natural Language Processing*. Cambridge University Press, Cambridge, UK, 2006.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Kent Bottles. Physician Executives Should Not Ignore How Smartphones Will Transform Healthcare. `http://empowerlms.wordpress.com/2011/02/04/physician-executives-should-not-ignore-how-smartphones-will-transform-healthcare/`, February 2011. Last access: 2011/10/09.

[9] David Brin. *The Transparent Society : Will Technology Force Us to Choose Between Privacy and Freedom?* Addison-Wesley, 1998.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[11] Jeffrey Van Camp. Google Activates 500.000 Android Devices A Day. `http://www.digitaltrends.com/mobile/google-activates-500000-android-devices-a-day-may-reach-1-million-by-october/`, June 2011. Last access: 2011/10/09.

[12] Avik Chaudhuri. Language-based security on Android. In Stephen Chong and David A. Naumann, editors, *PLAS*, pages 1–7. ACM, 2009.

[13] B. Chess and G. Mcgraw. Static analysis for security. *Security & Privacy Magazine, IEEE*, 2(6):76–79, 2004.

[14] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective Symbolic Execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. *SIGPLAN Not.*, 46:265–278, March 2011.

[16] Austin Cline. Privacy vs. Secrecy: Should We be More Open, Not Less? `http://atheism.about.com/b/2006/04/03/privacy-vs-secrecy-should-we-be-more-open-not-less.htm`, April 2006. Last access: 2011/10/09.

[17] Judith DeCew. Privacy. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, fall 2008 edition, 2008.

[18] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20:504–513, July 1977.

[19] Digitizor. Researcher Says That 8% of Android Apps Are Leaking Private Information. `http://digitizor.com/2011/07/21/android-malware/`, July 2011. Last access: 2011/10/09.

[20] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[21] 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 OSDI 2010*, October 2010.

[22] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[23] Joan Engebretson. Survey: Smartphones Will Be Dominant Mobile Health Device. `http://connectedplanetonline.com/business_services/news/Survey-smartphones-will-be-dominant-mobile-health-device-0112/`, January 2011. Last access: 2011/10/09.

[24] Flurry. Mobile Apps Put the Web in Their Rear-view Mirror. `http://blog.flurry.com/bid/63907/Mobile-Apps-Put-the-Web-in-Their-Rear-view-Mirror`, June 2011. Last access: 2011/10/09.

[25] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, IEEE S&P'10, 2010.

[26] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, pages 21–26, New York, NY, USA, 2011. ACM.

[27] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droid You're Looking for: Retrofitting Android to Protect Data from Imperious Applications. Technical report, UW-CSE-11-04-01, April 2011.

[28] Patrick Jackson. Upstreaming the Android Emulator. `http://gsoc11-qemu-android.blogspot.com/`. Last access: 2011/10/09.

[29] Ben Pfaff Tal Garfinkel Kevin Christopher Mendel Rosenblum Jim Chow. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.

[30] Dustin Karnes. Suspicious Android Apps May Be Stealing Millions of Users' Data. `http://www.talkandroid.com/8331-suspicious-android-apps-may-be-stealing-millions-of-users-data/`, July 2010. Last access: 2011/10/09.

[31] Gamble KH. Wireless Tech Trends 2010. Trend: Smartphones. *Healthcare Informatics*, 2010.

[32] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19:385–394, July 1976.

[33] Stefan Klement. Sicherheitsaspekte der Google Android Plattform. Master's thesis, Universität Bremen, April 2011.

[34] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, December 1992.

[35] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[36] M. H. Lyons, J. M. M. Potter, Holm, R. Venousiou, and R. Ellis. The Socio-Economic Impact of Pervasive Computing — Intelligent Spaces and the Organisation of Business. *BT Technology Journal*, 22:27–38, July 2004.

[37] Felipe Andres Manzano. The Symbolic Maze. `https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/`. Last access: 2011/10/09.

[38] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[39] Institute of Electrical and Electronics Engineers. IEEE Standard 1061-1992 Standard for a Software Quality Metrics Methodology, 1992.

[40] Panagiotis Papadimitriou and Hector Garcia-Molina. Data Leakage Detection. *IEEE Trans. Knowl. Data Eng.*, 23(1):51–63, 2011.

[41] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile Protection for Smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[42] Jeffrey Rosen. Why Privacy Matters. `https://www.wilsonquarterly.com/article.cfm?AID=90&AT=0`, October 2000. Last access: 2011/10/09.

[43] Artem Russakovskii. Massive Security Vulnerability In HTC Android Devices Exposes Phone Numbers, GPS, SMS, Emails Addresses, Much More. `http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more/`, October 2011. Last access: 2011/10/09.

[44] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security and Privacy*, 8:35–44, March 2010.

[45] Kahn Soheil. Analysis of Dalvik Virtual Machine And Class Path Library. Technical report, Security Engineering Research Group, Institute of Management Sciences Peshawar, 2009.

[46] S. Thrum and Y. Kane. Your Apps are Watching You. *Wall Street Journal*, 2010.

[47] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Blo Jason A., George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Neil Versel. Smartphone Boom Changes Physicians Relationship with Technology. `http://www.fiercemobilehealthcare.com/story/smartphone-boom-changes-physicians-relationship-technology/2010-08-31#ixzz0yKyd66hu`, August 2010. Last access: 2011/10/09.

[49] Samuel D. Warren and Louis D. Brandeis. The Right to Privacy. *Harvard Law Review*, IV(5), December 1890.

[50] Richard A. Wasserstrom. Privacy: Some Arguments and Assumptions. In Ferdinand David Schoeman, editor, *Philosophical Dimensions of Privacy: An Anthology*. Cambridge University Press, 1984.

[51] Wikipedia. Android Market — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Android_Market&oldid=442750223`, 2011. Last access: 2011/08/06.

[52] Wikipedia. Scintigraphy — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Scintigraphy&oldid=454451543`, 2011. Last access: 2011/10/09.

[53] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *TRUST*, pages 93–107, 2011.

[54] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT FSE*, pages 97–106, 2004.