PhD Thesis

A framework for automatic security checking of mobile applications

Luca Verderame

University of Genoa

PhD in Electronic and Computer Engineering, Robotics and Telecommunication

(XXVIII Cycle)

Supervisors: Prof. Alessandro Armando
Prof. Alessio Merlo

February 2016

Declaration

This dissertation was conducted under the supervision of Alessandro Armando and Alessio Merlo. The work submitted in this thesis is the result of original research carried out by myself, except where acknowledged. It has not been submitted for any other degree or award.

Luca Verderame

"A tutte le persone che mi hanno accompagnato e sostenuto in tutti questi anni.
Non sarei mai arrivato fin qui senza di voi."

Abstract

The swift and continuous evolution of mobile devices is encouraging both private and public organizations to adopt the Bring Your Own Device (BYOD) paradigm. As a matter of fact, the BYOD paradigm drastically reduces costs and increases productivity by allowing employees to carry out business tasks on their personal devices. However, it also increases the security concerns, since a compromised device could disruptively access the resources of the organization. The current mobile application distribution model based on application markets does not cope with this issue.

In this dissertation work I propose a novel mobile code distribution framework called Secure Meta-Market which supports the specification and enforcement of security policies and fully embrace the BYOD paradigm although preserving the employees experience. The Secure Meta-Market keeps track of the security state of devices and - through a functional combination of static analysis and code instrumentation techniques - supervises the installation of new applications thereby ensuring the enforcement of the corporate security policies.

Also, I present a prototype implementation of the Secure Meta-Market, called BYODroid, used for validating a wide range of popular Android applications against a security policy drawn from the US Government BYOD Security Guidelines. Experimental results obtained by running the prototype confirm the effectiveness and the efficacy of the approach.

Finally I discuss an implementation of BYODroid that has been prepared for the *NATO Communication and Information (NCI) Agency* and tested at the agency premises in The Hague, Netherlands. The contribution of this activity allowed the application of the Secure Meta-Market in a real world environment.

Acknowledgments

I would like to express my special thanks and gratitude to Alessandro Armando, Alessio Merlo and Gabriele Costa for assisting me throughout these years of research. Their guidance and advisoring deeply contributed to my professional and scientific growth.

I am also thankful to all the colleagues and friends of the Computer Security Laboratory at the University of Genoa for making the working environment a comfortable place to exchange ideas and suggestions.

I would also thank Konrad Wrona and his colleagues at the NCI Agency premise in the Hague, for their support and hospitality throughout the pilot project activities.

A special thanks goes to my extraordinary family that supports and encourages me so completely over the years, to my girlfriend Ilaria, for sustaining me in every single moment, and to my roommate Federico for being a great friend.

Last but not least, thanks to all the colleagues, friends and people who have accompanied me in this journey. I would not be here without them.

Luca Verderame

Publications and Recognitions Arising from this Thesis

Part of the work described in this thesis has been originally published as described in the following list:

1. A. Armando, G. Costa, A. Merlo, and L. Verderame, “Formal modeling and automatic enforcement of bring your own device policies,” *International Journal of Information Security*, vol. 14, no. 2, pp. 123–140, 2015.

2. A. Armando, G. Costa, A. Merlo, and L. Verderame, “Enabling byod through secure meta-market,” in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec ’14, (New York, NY, USA), pp. 219–230, ACM, 2014.

3. A. Armando, G. Costa, L. Verderame, and A. Merlo, “Securing the "bring your own device" paradigm,” *Computer, IEE Computer Society*, vol. 47, no. 6, pp. 48–56, 2014.

4. Alessandro Armando, Gabriele Costa, Alessio Merlo, Luca Verderame, "Market-Based Security for Mobile Devices", in *ERCIM News*, vol. 93, 2013.

5. A. Armando, G. Costa, A. Merlo, and L. Verderame, “Bring your own device, securely,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC ’13, (New York, NY, USA), pp. 1852–1858, ACM, 2013.

6. Alessandro Armando, Gabriele Costa, Alessio Merlo, Luca Verderame, "Securing the "Bring Your Own Device" Policy", *Journal of Internet Services and Information Security (JISIS)*, 2012.

Recognitions BYODroid won the **Star Cup 2013 - II Call**[*] of the University of Genoa and was one of the ten finalists in the Idea Challenge 2014 on Privacy, Security and Trust launched by the EIT ICT Labs of the European Institute of Innovation and Technology[†].

Moreover, BYODroid had been selected for the **NATO Communication and Information Agency Security Incubator Pilot Project 2015**[‡]. This project has allowed the development of an experimental setup tailored for NCI Agency and it has been validated by NCI Agency personnel. The final demonstration of this project took place at the NIAS Cyber Security Symposium in September 2015[§].

[*] https://unige.it/newsletter_uff/articoli/n41_startcup2013.shtml

[†] http://www.investintrentino.it/News/Nov.-13-14-Final-event-of-the-EIT-ICT-Labs-Idea-Challenge

[‡] https://www.ncia.nato.int/NewsRoom/Pages/150706-CyberSecuritySL-innovation-pilot.aspx

[§] http://nias2015.com/

Contents

Contents **viii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background and Preliminaries **4**
- 2.1 Android Environment 4
  - 2.1.1 Android Architecture 4
  - 2.1.2 Android Security Framework 5
  - 2.1.3 Android Applications 6
  - 2.1.4 Interactions in Android 7
- 2.2 Labeled Transition System 8
- 2.3 Büchi Automata 9
- 2.4 Security Automata 11

3 Secure Meta-Market **13**
- 3.1 Bring Your Own Device 13
  - 3.1.1 Motivating scenario 14
  - 3.1.2 Commercial solutions for BYOD 16
- 3.2 Mobile Application Market 17
  - 3.2.1 Security Issues 17
- 3.3 Secure Meta-Market 18
  - 3.3.1 Secure Meta-Market Workflow 20

4 Application Verification Framework **21**
- 4.1 Formalizing the Android Application Framework 23

4.1.1 Programming Framework . . . 23
4.1.2 Type and Effect . . . 27
4.1.3 Discussion . . . 31
4.2 Control flow graphs . . . 32
4.2.1 CFG to History Expression Conversion . . . 36
4.3 ConSpec Policy Language . . . 36
4.3.1 ConSpec Syntax . . . 36
4.3.2 ConSpec Semantics . . . 38
4.4 Application Verification . . . 39

5 Application Monitoring 45
5.1 Mobile Application Monitoring . . . 45
5.1.1 Application Instrumentation . . . 47
5.2 SMM Monitoring Environment . . . 48

6 BYODroid, A Secure Meta-Market for Android 51
6.1 BYODroid Architecture . . . 51
6.2 Security Manager . . . 52
6.3 Model Extractor . . . 53
6.4 Application Verifier . . . 54
6.4.1 Promela Encoding . . . 55
6.5 Application Manager . . . 60
6.6 Application Instrumenter . . . 60
6.7 Byodroid Client Application . . . 61
6.7.1 Application instrumentation and monitoring . . . 61

7 Use Case: BYODroid for NATO Agency 64
7.1 Case study . . . 65
7.2 Encoding the NCI Agency policy . . . 67
7.2.1 Policy compositionality and roles . . . 69
7.3 BYODroid policy enforcement . . . 70
7.3.1 Overall architecture . . . 70
7.3.2 Development . . . 71

8 Experimental Results 73
8.1 Experimental setup . . . 73
8.2 Experimental results . . . 76
8.3 Discussion . . . 79

9 Related Works 83

10 Conclusion and Future Works **87**

A Appendices **89**
A.1 Technical Proof 89
A.2 Promela Specifications 95
A.3 NCI Agency Policy 99

Bibliography **102**

List of Figures

2.1 The Android Stack 5
2.2 Example of Label Transition System. 9
2.3 Example of Büchi Automaton. 11
2.4 Example of Security Automaton. 12

3.1 A BYOD scenario. 15
3.2 The Secure Meta-Market approach. 19
3.3 The Secure Meta-Market workflow 20

4.1 A (simplified) example of the model extraction process (contd.). . . . 35
4.2 ConSpec automaton of the policy of Example 10 39
4.3 Büchi automaton B. 41
4.4 Label Transition System M'. 42
4.5 Product automaton $\mathcal{M}$. 43
4.6 Violating trace η in the product automaton $\mathcal{M}$. 44

5.1 Architecture of distributed PDP. 46
5.2 Architecture of centralized PDP. 46
5.3 PEP Direct Injection. 48
5.4 PEP Wrapping Injection. 48
5.5 SMM Instrumentation and monitoring. 50

6.1 BYODroid Architecture. 52
6.2 Meta-market client interface. 62
6.3 The Instrumentation procedure. 63

7.1 A schematic representation of the BYOD interoperability model. . . 65
7.2 BYODroid workflow for NCI Agency. 70
7.3 BYODroid Client interface for NCI Agency. 71

7.4 BYODroid interoperability model. 72

8.1 BYODroid Model Extraction. 77
8.2 Applications model size. 77
8.3 BYODroid Application Verification. 78
8.4 BYODroid Instrumentation. 79
8.5 BYODroid Total Verification Time. 80

9.1 The Security-by-Contract workflow including code producer and consumer. 85

List of Tables

4.1 Syntax of applications and components 23
4.2 Semantics of expressions (Part 1). 25
4.3 Semantics of expressions (Part 2). 26
4.4 Semantics of history expressions 28
4.5 Typing rules. 30

8.1 BYODroid Server computation times and outcomes (excerpt). 81
8.2 BYODroid Client computation times and outcomes (excerpt). 82

1

Introduction

The rapid and continuous evolution of mobile technologies is pushing forward the interest for novel, widespread paradigms. Among them, the **Bring Your Own Device** [1] (BYOD) is polarizing attention and investments [2]. The BYOD paradigm allows people to join an organization and access its resources by using their own device. Since it promises cost reduction and increased productivity, it is not surprising that the BYOD paradigm is receiving a growing attention from organizations worldwide. Indeed, current predictions[†] state that by 2017, nearly half of the employers will require their employees to supply their own device for work purposes, causing the corporate-liable programs an exception.

However, securing BYOD environments poses new challenges due to the twofold role of devices. On the one hand, personal use of the device must comply with the security policy of the organization. Since the behavior of a device depends on the installed software, some policy enforcement technology must verify or monitor its execution. On the other hand, the enforcement of the security policy should not overturn the user experience. It must be noted that current mobile applications deployment frameworks, mostly based on dedicated web services called *applications markets* (e.g. Apple Store, Google Play Store and Samsung Store), provide little or even no support to tackle these challenges. Some application markets implement a review process, e.g., see [3, 4], featuring some form of security analysis, but they are poorly documented and it is therefore unclear what kind of guarantees they actually provide. Interestingly, severe limitations have been also reported. For instance, malicious apps have been positively reviewed and published [5]. This lack of security support is often mitigated by asking to the device owner to take security-critical decisions, e.g. to decide whether to

[†] `http://www.gartner.com/newsroom/id/2466615`

install an application on the ground of the permissions requested and the reputation of the developer. Needless to say, the vast majority of users lack of the skills to take informed decisions and this approach therefore exposes organizations to dangerous threats.

In this dissertation work, in order to tackle the above issues, I describe the design of a new mobile code distribution framework, called **Secure Meta-Market** (SMM) [6, 7, 8], that supports the analysis of applications scheduled for installation in order to detect and prevent malicious behaviors against the device or not respecting the security policy of an organization.

A Secure Meta-Market masks the actual application markets by mediating all the installation requests and supports the enforcement of BYOD security policies through formal analysis and monitoring inlining. This is achieved by supporting (*i*) the definition of fine-grained security policies, (*ii*) the static verification of applications, and (*iii*) the security instrumentation necessary to monitor the applications that fail the verification step.

The verification of the compliance of applications is the central activity of the Secure Meta-Market. This is achieved through model checking, i.e. by verifying that a given model, i.e the model of an application, satisfies a specification, i.e. a security policy. If the application is compliant with the policy, the Secure Meta-Market validates the application, which is freely installed on the device.

If the response of the model checker is inconclusive or negative, the package is sent to a proper component which instruments the code with security checks. The resulting applications is then installed on the device and monitored by the Secure Meta-Market in order to prevent runtime policy violations.

This thesis also presents a prototype implementation of the Secure Meta-Market, called **BYODroid**, which supports Android-based devices and allows the security assessment of applications taken from the Google Play Store against corporate BYOD policies. BYODroid consists of two main components: the BYODroid Server and the BYODroid Client. BYODroid Client allows users to access the BYODroid Server and install only new applications that comply with the policy provided by the organization. The BYODroid Server is responsible for holding the security state of the devices registered to the organization and for mediating the access to application markets by only allowing a connected device to install applications compliant with the security policy. In particular, BYODroid Server implements the model extraction from applications, its verification against the policy and the application instrumentation procedure.

To assess the effectiveness of the Secure Meta-Market approach, I applied BYODroid to check the compliance of 860 Android applications taken from the Google Play Store. Applications are validated against a real world BYOD security policy (extracted from the US Government BYOD Security Guidelines [9]). The experimental results confirm the effectiveness of the proposed approach.

Finally, I present the development of an experimental setup of BYODroid specifically tailored for the **NATO Communication and Information (NCI) Agency**. This activity allows the evaluation and validation of BYODroid at the agency premises in The Hague, Netherlands, and the definition of a BYOD policy extracted from their security rules on mobile devices.

Structure of the Thesis This thesis is structured as follows. Chapter 2 sketches the background required throughout the discussion. The concept of Secure Meta-Market is described in Chapter 3. The application verification framework is discussed in Chapter 4, while the application monitoring is introduced in Chapter 5. The BYODroid prototype implementation is extensively discussed in Chapter 6, while the real-case implementation prepared for the NCI Agency is described in 7. The experimental results are reported and discussed in Section 8 while Chapter 9 puts the thesis in context by discussing the related work. Finally, Chapter 10 draws some concluding remarks.

2

Background and Preliminaries

In this chapter I introduce the mobile ecosystem, i.e. Android, used as a motivating example throughout the thesis. Finally, I sketch some formal concepts useful for the verification framework presented in Chapter 4.

2.1 Android Environment

Although the Secure Meta-Market framework is platform independent, its current prototype implementation BYODroid has been developed for the Android operating system. Being the most widespread, with over one billion of users in the late 2015[†], Android is a mobile OS developed by Google from 2008.

In the following of this section a brief overview on Android and its applications is provided.

2.1.1 Android Architecture

The Android stack, depicted in Figure 2.1, can be represented with 5 functional levels: Application, Application Framework, Application Runtime, Libraries and the underlying Linux kernel.

1. *Application Layer*. It includes both system (home, browser, email,. . .) and user-installed Java applications.

2. *Application Framework*. It provides the main OS services by means of a set of APIs. This layer also includes services for managing the device and

[†]http://www.theverge.com/2015/9/29/9409071/google-android-stats-users-downloads-sales

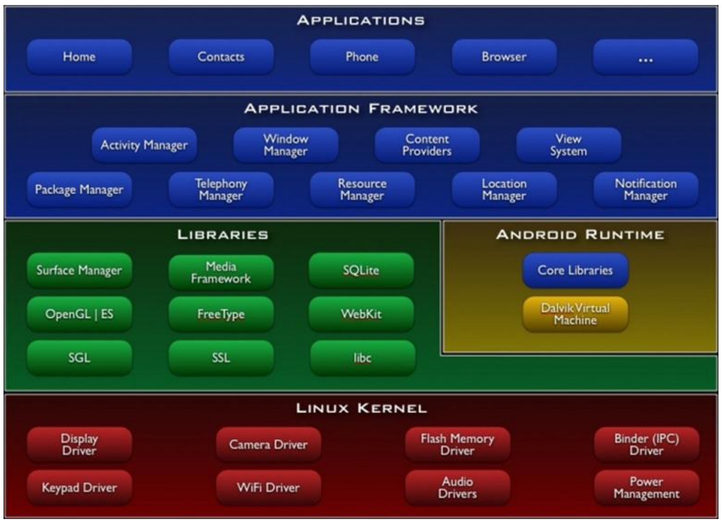


Figure 2.1: The Android Stack

interacting with the underlying Linux drivers (e.g. *Telephony Manager* and *Location Manager*).

3. *Android Runtime.* This layer comprises the Dalvik virtual machine, the Android's runtime's core component which executes applications.

4. *Libraries.* It contains a set of C/C++ libraries providing useful tools to the upper layers and for accessing data stored on the device. Libraries are widely used by the Application Framework services.

5. *Linux kernel.* Android relies on a Linux kernel for core system services. Such services include process management and drivers for accessing physical resources and Inter-Component Communication (ICC).

2.1.2 Android Security Framework

The Android Security Framework (ASF) provides a cross-layer security solution (i.e. sandboxing) built by combining native per-layer security mechanisms. Each layer in the Android stack (except the Libraries layer) comes with its own security mechanisms:

- **Application layer (Android Permissions)**. Each application comes with a file named `AndroidManifest.xml` that contains the permissions that the application may require during execution. During installation the user is asked to grant all the permissions specified in the manifest.

- **Application Framework (Permission Enforcement)**. Services at this layer enforce the permissions specified in the manifest and granted by the user during installation.
- **Runtime (VM Isolation)**. Each application is executed in a separate Dalvik VM machine. This ensures isolation among applications.
- **Linux (Access Control)**. As in any Linux kernel, resources are mapped into files (e.g. sockets, drivers). The Linux Discretionary Access Control (DAC) model associates each file with an owner and a group. Then, DAC model allows the owner to assign an access control list (i.e. read, write, and/or execute) on each file to the owner itself (UID), the owner's group (GID) and other users.

Sandboxing is a cross-layer solution adopted in Android to provide strong isolation among applications. In detail, Android achieves sandboxing of the applications by binding each Android application to a separate user at Kernel layer, thereby combining the native separation due to the execution of applications on different Dalvik VMs with the isolation provided by native Linux access control.

Once an application is installed on the device (i.e. the user accepts all required permissions in the `AndroidManifest.xml` file) a new user at the Linux layer is created and the corresponding user id (UID) is bound to the installed application. Once the application is launched, a new process, with such UID, and a novel Dalvik VM are created in order to execute the application.

This solution forces any non privileged UID to have at most one process running (i.e. the one containing the running application). Rarely, more than one active process for the same UID can be allowed if explicitly requested in the `AndroidManifest.xml`. However, the maximum number of active processes is upper-bounded by the number of components composing the application.

At runtime, sandboxing and other per-layer security mechanisms are expected to avoid illegal interplay. For instance, if an application tries to invoke a `kill` system call on the process hosting another application, the sandboxing is violated (i.e. a Linux user tries to kill a process belonging to another user) and the system call is blocked.

2.1.3 Android Applications

Applications are made of *components* corresponding to independent execution modules, that interact with each others. There exist four kinds of components:

- *Activity*, representing a single application screen with a user interface,
- *Service*, which is kept running in background without interaction with the user,
- *Content Provider*, that manages application data shared among components of (potentially) distinct applications,
- *Broadcast Receiver* which is able to respond to system-wide broadcast announcements coming both from other components and the system.

Components are defined in *namespaces* that map components to a specific name which allow to identify components in the system.

2.1.4 Interactions in Android

In Android, interactions can be *horizontal* (i.e. application to application) or *vertical* (i.e. application to underlying levels). Horizontal interactions are used to exploit functionalities provided by other applications, while vertical ones are used to access system services and resources. Component services are invoked by means of a message passing paradigm, while resources are referred by a special formatted URI. Android URIs can also be used to address a content provider database.

Horizontal interactions are based on a message abstraction called *intent*. Intent messaging is a facility for dynamic binding between components in the same or different applications. An intent is a passive data structure holding an abstract description of an operation to be performed (called *action*) and optional data in URI format. Intents can be *explicit* or *implicit*. In the former case, the destination of the action is explicitly expressed in the intents (through the *name* of the receiving application/component), while in the latter case the system has to determine which is the target component accordingly to the action to be performed, the optional data value and the applications currently installed in the system.

Intent-based communications are granted by a kernel driver called Binder which offers a lightweight capability-based remote procedure call mechanism. Although intent messaging passes through the Binder driver, it is convenient to maintain intent's level of abstraction for modeling purpose. In fact, every Android application defines its entry points using *intent filters* which are lists of intent's actions that can be dispatched by the application itself. Furthermore, an intent can be used to start activities, communicate with a service or send broadcast messages.

Vertical interactions are used by applications to access system resources and functionalities which are exposed through a set of APIs. Although system calls can cause a cascade of invocations in the lower layers, possibly reaching the kernel, all of them are mediated by the application framework APIs. Hence, APIs mask internal platform details to the invoking applications.

Internally, API calls are handled according to the following steps. When an application invokes a public API in the library, the invocation is redirected to a private interface, also in the library. The private interface is an RPC stub. Then, the RPC stub initiates an RPC request with the system process that asks a system service to perform the requested operation.

2.2 Labeled Transition System

Labeled Transition Systems (LTS) [10] is one of the major proposal for modeling non deterministic systems.

A LTS is a structure, graphically depicted as a direct graph, consisting of states with transitions, labeled with actions, between them. The states model the system states; the labeled transitions model the actions that a system can perform.

Definition 2. *Let A be the set of finite actions of a system $\mathcal{S}$. The LTS representing $\mathcal{S}$ is a tuple (S, i, Σ, T), where*

- *S is a finite set of states;*
- *$\Sigma = A \cup \{\tau\}$ is the set of actions of the system including the silent action τ;*
- *$T \subseteq S \times \Sigma \times S$ is a set of transaction relations;*
- *$i \in S$ is the initial state;*

Definition 3. *An* execution trace η *of a Labeled Transition System is a series of transitions labels (starting from the initial state $i \in S$) in the form $\sigma_1\sigma_2 \dots \sigma_n$ if and only if $\exists s_1, \dots, s_n \in S \mid s_0 \xrightarrow{a_1} s_1 \dots s_{n-1} \xrightarrow{a_n} s_n$.*

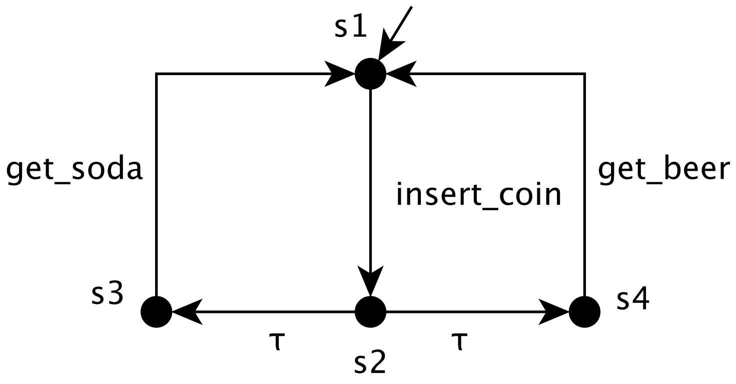
Example 2. *Let consider a vending machine that only delivers a drink (soda or beer) after the user inserts a coin.*

The corresponding LTS $\mathcal{L}$ is a tuple (S, i, Σ, T), where

- *$S = \{s_1, s_2, s_3, s_4\}$;*

- *$\Sigma = \{instert\_coin, get\_soda, get\_beer, \tau\}$;*
- *T is depicted in Figure 2.2;*
- *$i = s_1$.*

Figure 2.2: Example of Label Transition System.



$\mathcal{L}$ is non deterministic since the user can choose alternatively a soda or a beer. This results in the τ transition.

Possible execution traces *of $\mathcal{L}$ are $\eta' = insert\_coin, \tau, get\_soda$ and $\eta'' = insert\_coin, \tau, get\_beer$.*

□

2.3 Büchi Automata

Büchi automata are an extension of finite automata to infinite inputs. The formalism was invented by Büchi around 1960 to prove the decidability of the Monadic Second Order Theory of Natural Numbers [11]. Intuitively, a Büchi automaton defines an abstract controller which reads the elements of an execution trace. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the final states infinitely often.

Büchi automata recognize the omega-regular languages, the infinite word version of regular languages.

Definition 4. *A Büchi automaton is a tuple (Q, Λ, T, F, q_0) where*

- *Q is a finite set of states;*
- *Λ is a finite alphabet of actions;*
- *$T \subseteq Q \times \Lambda \times Q$ is a transaction relation;*
- *$F \subseteq Q$ is a set of final (or accepting) states and*
- *$q_0 \in S$ is a set of initial states;*

Büchi automata accept sequence of input symbols in the form $\beta = \alpha_0, ...\alpha_n \in \Lambda^*$. To process β, an automaton B moves from the initial state q_0 to a state $q' \in Q$ according to the symbols contained in β. If a sequence β causes the automaton to reach a final state, i.e. $T(T...(T(q_0, \alpha_0), \alpha_1), ..., \alpha_n) = q' \in F$, then β violates the security property represented by B.

Example 3. *Let consider a Büchi automaton B for a security policy that prohibits execution of $Send$ operations after a $FileRead$ has been executed. In this diagram, the automaton states are represented by the two nodes labeled q_nfr (for "no file read") and q_fr (for "file read"). B is the tuple (Q, Λ, T, F, q_0) where:*

- *$Q = \{q\_nfr, q\_fr, f\}$;*
- *$\Lambda = \{notFileRead, notSend, FileRead, Send\}$;*
- *T is a transaction relation depicted in Figure 2.3;*
- *$F = \{f\}$;*
- *$q_0 = q\_nfr$.*

The transition predicate $notFileRead$ is assumed to be satisfied by input symbols (system execution steps) that are not file read operations, and transition predicate $notSend$ is assumed to be satisfied by input symbols that are not message-send operations.

The message-send execution steps, defined from q_fr to f, cause the automaton to reach a final accepting state that corresponds to the policy violation.

Given two sequences β' and β'', where:

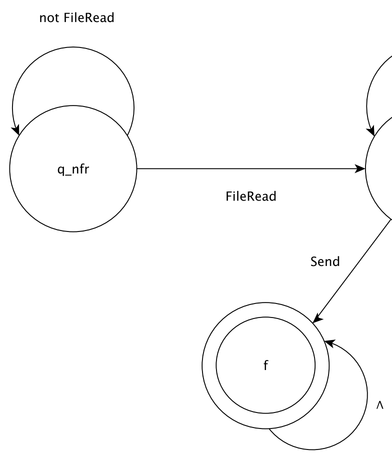
$\beta' = Send, Send, Send, FileRead, FileRead$

$\beta'' = Send, Send, FileRead, Send$

the Büchi automaton will accept β' and reject β'', since that β'' reaches the f state.

□

Figure 2.3: Example of Büchi Automaton.



2.4 Security Automata

Security Automata [12] (SA) are class of Büchi automata that accept safety properties. SA are similar to ordinary non-deterministic finite-state automata.

Definition 5. *A SA is a tuple $(Q, \Lambda, \delta, q_0)$ where*

- *Q is a set of states;*
- *Λ is a set of input symbols;*
- *$\delta : (Q \times \Lambda) \rightarrow Q$ is a transition function and*
- *$q_0 \in Q$ is the initial state.*

To process a sequence $\beta = \alpha_0, ...\alpha_n \in \Lambda^*$ of input symbols, the security automaton starts from the initial state q_0 and reads the sequence, one input symbol at a time. As each input symbol α_i is read, the security automaton changes its current state q' to $\bigcup_{q \in q'} \delta(q, \alpha_i)$.

If there exists a state $q' \in Q$ so that $\delta(\delta...(\delta(q_0, \alpha_0), \alpha_1), ..., \alpha_n) = q'$, then the input is accepted, meaning that β *satisfy* the security property represented by the SA.

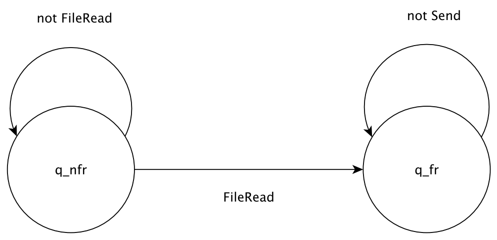
Example 4. *Consider again the security policy of Example 3. The security automaton of such a policy is a tuple $(Q, \Lambda, \delta, q_0)$ where*

- *$Q = \{q\_nfr, q\_fr\}$;*

- $\Lambda = \{notFileRead, notSend, FileRead\}$;
- δ *is a transaction relation depicted in Figure 2.4;*
- $q_0 = q\_nfr$.

Differently from the Büchi automaton of Figure 2.3, no explicit transition is defined from q_{fr} *for input symbols corresponding to message-send execution steps. This implies that the security automaton in Figure 2.4 rejects inputs in which a* $Send$ *follows a* $FileRead$*. Given the same two sequences* β' *and* β'' *of Example 3, the security automaton will accept* β' *and reject* β''*, meaning that* β'' *does not satisfy the security policy.*

Figure 2.4: Example of Security Automaton.



□

3

Secure Meta-Market

This chapter introduces the security issues related to the use of mobile devices and applications inside corporate infrastructures.

To solve such issues, I propose the Secure Meta-Market, a new mobile code distribution framework that supports the automated analysis of applications with respect to security policies.

3.1 Bring Your Own Device

Broadly speaking, *Bring Your Own Device* (BYOD) can be defined as a paradigm that allows people to use their own devices, whether occasionally, primarily or exclusively, for work. Many organizations allow people to supplement their corporate-owned machine(s) with additional devices/smartphones, tablets, laptops, home PCs-as needed for optimal flexibility, mobility and productivity.

Some go further and eliminate certain corporate-owned device types entirely for eligible employees who prefer to use their own devices instead. Contractors are increasingly required to use their own devices rather than being provided with corporate-owned equipment.

Ideally, organization's practices around BYOD should be detailed in a formal policy. The reality is that many people are already bringing their own devices to work, regardless of whether the organization has a BYOD policy in place.

The current average number of devices connecting to the corporate network is 5.18 per IT worker - 4.43 devices across all workers - and predicted to rise to almost six devices by 2020[†].

[†]Citrix, Workplace of the Future: a global market research report, September 2012.

In part, this reflects a shift in the nature of the endpoint environment, as the dominance of traditional desktop PCs gives way to a wider range of options that let people choose the right mix of device mobility, performance, size and weight for their purposes, whether it be a laptop, tablet or smartphone.

To date, BYOD has remained an informal practice for many organizations. In the meantime, the lack of a more coherent approach to BYOD can leave the organization exposed to risks from security and compliance gaps to escalating IT complexity.

As this trend continues its rapid rise, there is a growing demand for complete BYOD strategies, encompassing both policy and technology. From a technology perspective, the most obvious question is how people will be able to access enterprise applications and business information on their personal devices.

Simply installing apps directly on the device would raise serious security, privacy and compliance risks. Many people have also started using unmanaged third-party apps and online services for work - IT needs a way to control and manage this usage and prevent these apps from introducing security risks to the organization.

In the following of this section, a typical scenario for a company embracing a BYOD strategy is discussed.

3.1.1 Motivating scenario

In a BYOD scenario, each user joining an organization must register his owned device. The owner of a registering device can be either a new employee, a temporary collaborator or a customer.

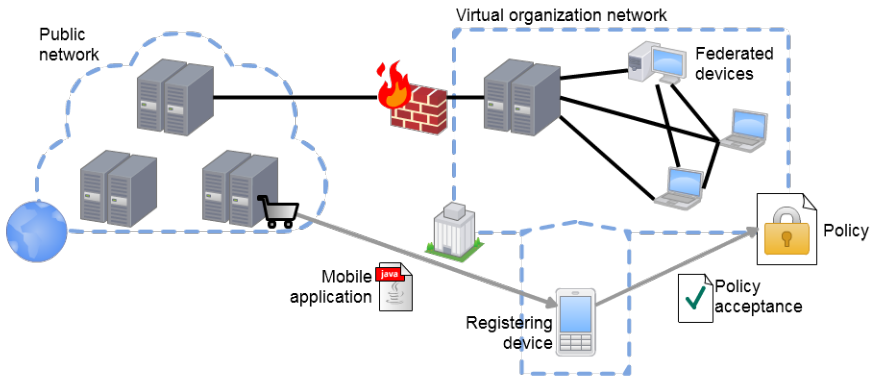
Many organizations have security policies that participants must accept and respect. Such policies are devoted to a two-fold aim, namely avoiding users to i) introduce malicious software inside the organization (outsider threats) and ii) take outside sensitive information for the organization (insider threats).

In general, security policies can cause a, possibly temporary, restriction on the usage of the devices. Still, devices can change their configuration by installing new software. Figure 3.1 depicts this scenario.

In detail, once a personal device is registered to the organization, it gets a policy from the organization which the device configuration is initially expected to fulfil. Installation/removal of external applications by the user are likewise expected to fulfil the policy.

A policy management system for this scenario must fulfil some important requirements. First of all, it must provide support for policy checking. This means that it must be always possible to verify the organization policy against a

Figure 3.1: A BYOD scenario.



registering device, independently of its specific configuration. Policy compliance must be formally granted, e.g., through model checking or automatic theorem proving.

Moreover, it ought to provide support for configuration changes. In particular, application installation and removal must be validated and, possibly, forbidden. Clearly, such validation cannot require a complete revaluation of the whole configuration of devices, which would result in a massive computation.

Finally, although the system will probably need on-device support, this must be implemented avoiding customizations of the devices. Indeed, device customizations, e.g., OS modules replacement, would drastically reduce the applicability of the system, while a solution built on top of the existing OS is general and non-invasive.

Example 5. *Consider a mobile device joining a virtual organization which applies a security policy saying "devices cannot access the network after using local file system in the same session". When an Android user, willing to respect this constraint, installs a new application, he checks the application description, namely its* `manifest`*, for behavioral details. He decides to avoid applications declaring both activities (although he cannot be sure they are unsafe). However, he cannot take a decision about programs which only require one of the two permissions. As a matter of fact, no information about possible interactions between the new application and already installed ones is available. As a consequence, the user can either decide to avoid suspicious applications (i.e., those using files or network) or take the risk to install some of them (which could lead to policy violation).* □

3.1.2 Commercial solutions for BYOD

To address the problem of managing mobile devices inside the corporate environment, as described in Section 3.1.1, companies can rely on Enterprise Mobility Management (EMM) solutions like Maas360[*] or Mobile Iron[†].

EMMs solutions provide a set of tools for managing devices, applications and contents inside a corporate environments. Indeed, a company can track all mobile devices of employees and enforce operations that can span from disabling wireless connection or remote update of software to a complete wipe of the phone.

Focusing of mobile applications management, EMM tools propose a combination of three different approaches:

1. **Blacklisting/Whitelisting**: the IT administrator controls permitted or denied applications by putting them in the corresponding list. The EMM system monitors the installation requests performed by employees and blocks the blacklisted applications. On the contrary, whitelisted applications are remotely installed on mobile devices.

2. **Application Containers**: the EMM system splits the mobile device into two or more partitions in which working applications are isolated from personal ones. This solution is able to provide strong security guarantees although it requires OS modifications and typically forces employees to use multiple copies of the same application (e.g. email and browser).

3. **Application Wrapping**: the EMM system instruments applications with proprietary libraries for monitoring and enforcing corporate policies. Although effective, applying instrumentation indiscriminately to all application introduces a non negligible overhead for monitoring. Moreover, applications modification and redistribution to employees' devices are not trivial tasks.

Although those approaches are able to mitigate security risk of mobile devices inside the organization infrastructure, none of the proposed solution is able to effectively and automatically enforce corporate policies on application or group of applications without requiring modification of the underlying operating system or of the application itself.

[*] `http://www.maas360.com/`
[†] `https://www.mobileiron.com/en`

3.2 Mobile Application Market

In recent years, the main way to distribute content to mobile devices, like applications, music and books is through the so-called **Mobile Application Market**.

Application markets such as Google Play, the Apple Store, and the Windows Store, allowing users browse apps and install them on their devices with "a single click" have unquestionable advantages. Finding and acquiring software is straightforward, and installation is standardized for all apps and requires little expertise.

From a technological point of view, application markets as mobile digital distribution platforms do not constitute a radical innovation, being quite similar to software libraries or marketplaces controlled by Web software distributors like Handster (acquired by Opera[*]), PocketGear[†] and GetJar[‡]. However, the innovation lies in translating this Computer-based and Web- oriented nature in the Mobile context: leveraging new smart phones features and capabilities, stores can be accessed from different networks, e.g. mobile or Wi-Fi, are mostly populated by software application and are characterized by improved quality of presentation - e.g. classification, search and discovery - directly impacting on user experience [13].

Moreover, the application creation and distribution paradigm's underlying idea is to grant higher openness and independence to third parties, following a "self-publishing model": not only traditional companies, but mainly single developers are provided with easy to use, widely interoperable and inexpensive - if not free - platforms and tools to create an application, publish it on the store.

3.2.1 Security Issues

Although application markets have non-negligible advantages in the widespreading of applications they also pose several security issues, as discussed in [14].

Indeed, application developers typically have no information about users' security requirements. Also, application markets' security concerns are usually limited to viruses and other common types of malware. Nevertheless, security violations involving non-malicious software have been reported - for example, confused deputy attacks [15] in which even trusted apps can become entry points for security breaches. Thus, usage control rules for private mobile devices in corporate environments should be evaluated against a device's specific applica-

[*]http://html5.oms.apps.opera.com/it_it/
[†]https://www.linkedin.com/company/pocketgear
[‡]http://www.getjar.com/

tion profile. (Hardware tampering and intentional system attacks by the device's owner or some other active adversary mostly involve intrusion detection and insider attack prevention, which I do not consider in this thesis.)

Being centralized repositories, application markets facilitate the deployment of security analysis tools and methodologies for the apps they host. For instance, the Apple Store checks all submitted apps before deciding whether or not to publish them, while Google Play employs Bouncer, an application scanning service, to mitigate malware diffusion [16].

Although helpful, these solutions suffer from numerous limitations. For instance, Tielei Wang et al. [17] showed how to forge a malicious application and publish it in the Apple Store, while Yajin Zhou et al. [18] likewise reported a method for easily circumventing Google Bouncer.

To effectively support the BYOD paradigm, application markets must specify security policies spanning multiple apps and check whether an app complies with a given security policy to ensure that the same app never violates that policy at runtime. Moreover, security solutions should be immediately applicable to existing mobile devices. Hence, deep OS customizations are out of scope, while application-level modifications are acceptable.

3.3 Secure Meta-Market

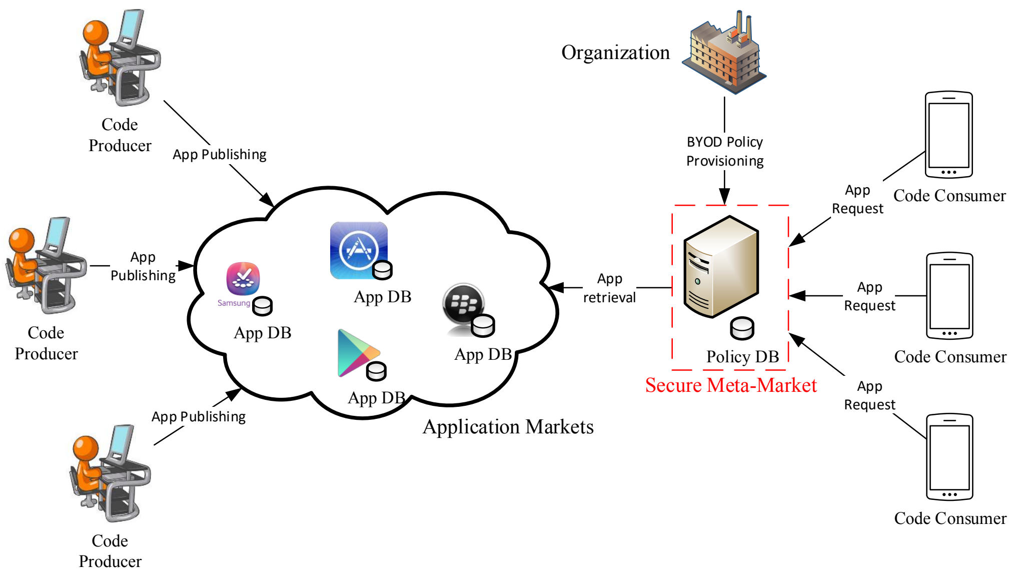
To mitigate security issues related to BYOD solutions for mobile devices and the security weaknesses posed by traditional application markets, I propose a new mobile code distribution framework called **Secure Meta-Market** (SMM).

The idea of Secure Meta-Market, firstly discussed in [7], is depicted in Fig. 3.2. The SMM acts as a security-enabled application market which enforces corporate BYOD policies on personal devices. Each corporation can install its own SMM and define the security policies suitable to its needs by using appropriate policy specification languages.

The Secure Meta-Market masks the actual application markets. For each application that the user wants to install, the SMM retrieves the application package directly from the official application market. Then, it verifies the compliance of the application against the current personal device configuration (i.e. the set of installed applications) and the BYOD policy. In case of non-compliance, the SMM proceeds by instrumenting the application through its client. Thus, the installation deploys a piece of software (original or instrumented) which keeps the device configuration compliant with the BYOD policy.

It must be emphasized that the Secure Meta-Market is transparent both to the

Figure 3.2: The Secure Meta-Market approach.



users and the application markets. In detail, the Secure Meta-Market has been designed to meet the following goals [19]:

1. *Transparency*. The user experience and usage patterns must not be affected. For instance, no (or minimal) customization of the OS interfaces and workflow must be implemented.

2. *Automatic verification.* The verification of the compliance of applications against the BYOD policy must be carried out in a fully automatic way, i.e., without user intervention.

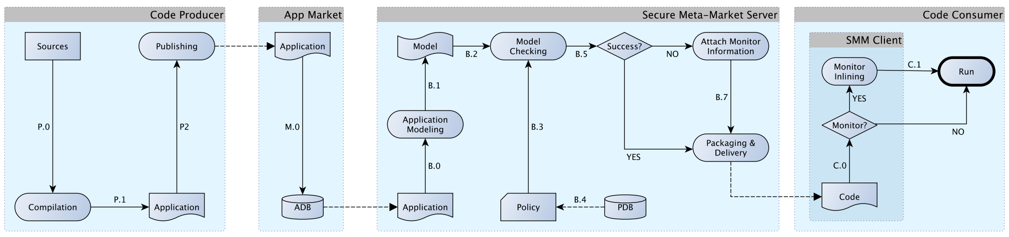
3. *Protection against colluding applications*. Before installation, an application is validated by keeping into account the whole configuration, i.e., all the installed applications, of the target device.

4. *Device configuration safety.* Changes to the configuration of a device which could lead to violations of the BYOD security policy should be prevented. Whenever the user's behavior causes irreparable conflicts with the policy, the device should be isolated from the BYOD environment and its resources.

3.3.1 Secure Meta-Market Workflow

The Secure Meta-Market features are supported through the workflow depicted in Fig. 7.2.

Figure 3.3: The Secure Meta-Market workflow



Initially, code producers compile (P.0) and generate mobile applications (P.1); then, they publish their applications (P.2) on a standard application market which stores (M.0) the software packages in a database (ADB). When a user requires an application from the Secure Meta-Market Server, the corresponding application is retrieved. Then, a model extraction procedure is applied to the application (B.0) to generate an application model (B.1). Since the model is extracted from the application code, no further validation is required. Hence, the model can be directly passed (B.2) to a verification process which checks its compliance against the security policy (B.3). Security policies are retrieved (B.4) from a policy database (PDB) handling policy instances customized over the devices configuration.

If the verification succeeds (B.5 → YES), the application is delivered to the user's device with no further action. Otherwise (B.5 → NO), the SMM attaches monitoring information to the application (B.7). Mainly, monitoring information consists of a digital signature which will be used by the SMM Client to obtain a correct instrumentation of the application. When the SMM Client receives a mobile application package, she checks whether it was marked for monitoring (C.0). If this is the case, before installation the mobile device instruments the application package with security checks by using information attached by the Secure Meta-Market Server (C.1). Otherwise, it is directly installed on the code producer.

In the following of this PhD thesis I will describe each of the component and technique that compose the workflow of the Secure Meta-Market and present an Android implementation of the SMM called **BYODroid**.

4 Application Verification Framework

The main activity of the Secure Meta-Market is the verification of applications behavior with respect to a security policy defined by an organization.
The verification consist in the following steps:

Model Extraction. The Secure Meta-Market needs to extract a mathematical structure formally describing the *behavior* of the application, i.e. a *model*.

Different models can be obtained from a piece of code depending on what information must/can be abstracted away. Usually, very abstract models are more compact and easier to handle/analyze while detailed ones lead to longer computations and, in certain cases, can also be intractable. Thus, the effectiveness of the proposed approach requires models to be as small as possible without neglecting any security-relevant aspect of the computation.

For instance, Android software packages (APKs) carry application code and non executable resources (e.g., pictures and multimedia files). Application code primarily consists of Android VM bytecode and, possibly, machine executable, a.k.a. *native*, modules. Every access to the valuable resources of the execution platform is performed by invoking a corresponding API or system call.

In [7] I proposed a formalization of the Android Application Framework and I extracted *history expressions* from applications using a *Type and Effect System* for a subset of the Java language, called Featherweight Java

[20]. This approach grants the soundness of the models that safely approximate the actual behavior of the applications

However, since the implementation of a framework based on the Type and Effect System for the Java/Android language poses technical difficulties (see Section 4.1.3), in [8] and [14] I opted for state-of-the-art algorithm for the extraction of *Control Flow Graphs*.

Policy Definition. A *security policy* is a definition of requirements that needs to be satisfied by a system.

Several policy specification languages have been presented for defining temporal properties over the execution traces. Among them, Automata-based formalisms are widely adopted. For instance, *Security automaton* defines an abstract controller which reads the elements of an execution trace. Whenever an execution trace leads the automaton to a final accepting state, it violates the policy.

Security automata have been also used for defining the formal semantics of some policy specification languages like PSLang [21] and ConSpec [22]. In particular, ConSpec has been specifically designed for high level languages, e.g., Java, which makes it suitable for the Android environment.

ConSpec is the current language used by the Secure Meta-Market, as discussed in [8] and [14].

Model Verification. Once the SMM extracts the model from a mobile application, it proceeds with the verification of its compliance with respect to a security policy using *model checking*.

Model checking [23] is a verification technique that systematically explores a computation model t by looking for an execution path, called *counterexample*, witnessing the violation of a given formula φ, expressing a desired property. If no counterexample exists, then t satisfies φ, in symbols $t \vDash \varphi$. Otherwise, the model checker returns the discovered counterexample.

The following of this chapter will detail all the verification steps involved in the Secure Meta-Market. In this chapter the Android ecosystem is used as a motivating example.

4.1 Formalizing the Android Application Framework

The research conducted in [24, 7] presented a model inference approach based on a type and effect system [25].

Such proposal relies on a formalization of the Android application framework based on the Featherweight Java. Although simplified, such framework included most of the relevant aspects of the Android IPC and components. The type and effect system infers a *history expression* [26] from each Android component. History expressions have been proved to *safely*, i.e., neglecting no security-relevant aspect, represent the actual behavior of the applications and to support the verification of a wide class of security policies.

The following of this section presents the Featherweight Java extension for Android and the extraction method of history expressions from an Android application.

4.1.1 Programming Framework

The syntax of the programming language is given in Table 4.1.

Table 4.1: Syntax of applications and components

$L ::= \texttt{class}\,\texttt{C}\,\texttt{extends}\,\texttt{C}'\,\{\bar{\texttt{D}}\,\bar{\texttt{f}};\,\texttt{K}\,\bar{\texttt{M}}\}$	Class
$K ::= \texttt{C}(\bar{\texttt{D}}\,\bar{\texttt{x}})\{\texttt{super}(\bar{\texttt{x}})\,;\,\texttt{this}.\bar{\texttt{f}} := \bar{\texttt{x}}\,;\}$	Constructor
$M ::= \texttt{C}\,\texttt{m}(\texttt{D}\,\texttt{x})\{\texttt{return}\,E;\}$	Method
$E ::= \texttt{null} \mid u \mid x \mid \texttt{new}\,\texttt{C}(\bar{E}) \mid E.\texttt{f} \mid$	Expressions
$E.\texttt{m}(E') \mid \texttt{system}_\sigma\,E \mid E;E' \mid (\texttt{C})E \mid$	
$\texttt{if}\,(E = E')\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} \mid \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \mid$	
$I_\alpha(E) \mid \texttt{icast}\,E \mid \texttt{ecast}\,\texttt{C}\,E \mid E.\texttt{data}$	

A class C, possibly extending C′, declaration contains a list of typed ($\bar{\texttt{D}}$) fields $\bar{\texttt{f}}$, a constructor K and a set of methods $\bar{\texttt{M}}$. A constructor K consists of its class C, a list of typed parameters $\bar{\texttt{x}}$ and a body. The body of the constructor contains an invocation to the constructor of the superclass `super` and a sequence of assignments of the parameters to the class fields. Methods declarations have a signature and a body. The signature identifies the method by means of its return type C (I write `void` when no object is returned), its name m and its typed parameters $\bar{\texttt{x}}$. Instead, the method body contains an expression E whose computation provides the return value. Finally, expressions can be either the void value `null`, a system resource u[*], a variable x, an object creation $\texttt{new}\,\texttt{C}(\bar{E})$, a field access $E.\texttt{f}$, a method invocation $E.\texttt{m}(E')$, a system call $\texttt{system}_\sigma\,E$, a sequence of two expressions $E;E'$, a type cast $(\texttt{C})E$, a conditional branch $\texttt{if}\;(E = E')\{E_{tt}\}\;\texttt{else}\;\{E_{ff}\}$ or a thread creation $\texttt{thread}\,\{E\}\;\texttt{in}\,\{E'\}$. Android-specific expressions are defined, namely the intent creation $I_\alpha(E)$, implicit or explicit intent propagation ($\texttt{icast}\,E$ and $\texttt{ecast}\,\texttt{C}\,E$, respectively), an intent content reading $E.\texttt{data}$. Intents correspond to messages that applications exchange in order to communicate.

Example 6. *Consider the following classes*

```
class Browser extends Receiver {
  Browser() { super(); }
  void receive(I_www i) { return system_connect i.data; }
}
class Game {
  Game() { super(); }
  void start() { return system_read ~/sav;
       if(UsrAct = TouchAD)
           then {icast I_www("http://ad.com") }
           else {/*...play...*/system_write ~/sav; } ;
  }
}
```

The class `Browser` *is a receiver for intents* `www`*. Method* `receive`*, when triggered, connects to the url carried by the incoming intent. Class* `Game` *implements a stand-alone application (assuming method* `start` *to act as entry point). Its first step consists in reading the content of a file* `~/sav`*. Then, its execution can take two different branches (for our purposes the terms in the guard are irrelevant). If users clicks on an in-game advertisement, an* `www` *intent containing a url is fired. Otherwise, the game begins and, eventually, the file* `~/sav` *is written.* □

Operational semantics

The behavior of programs follows a small step semantics. Some of the semantic rules are reported in Table 4.2 and Table 4.3.

Rules are of the type $\omega, E \to \omega', E'$ where ω, ω' are sequences of system calls performed by the execution, namely *execution histories*, and E, E' are ex-

[*]Resources belong to the class `Uri` and we can use specially formatted strings, e.g., "`http://site.com`" or "`file://dir/file.txt`", to uniquely identify them.

pressions. A rule $\omega, E \to \omega', E'$ says that a configuration ω, E can perform a computation step and reduce to ω', E'.

Table 4.2: Semantics of expressions (Part 1).

$$(\texttt{NEW})\ \frac{\omega, E_i \to \omega', E_i'}{\omega, \texttt{new}\,\texttt{C}(\bar{v}, E_i, \ldots) \to \omega', \texttt{new}\,\texttt{C}(\bar{v}, E_i', \ldots)}$$

$$(\texttt{FLD}_1)\ \frac{\omega, E \to \omega', E'}{\omega, E.\texttt{f} \to \omega', E'.\texttt{f}} \qquad (\texttt{FLD}_2)\ \frac{\mathit{fields}(C) = \bar{D}\,\bar{f}}{\omega, \texttt{new}\,\texttt{C}(\bar{v}).\texttt{f}_\texttt{i} \to \omega', v_i}$$

$$(\texttt{SYS}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{sys}_\sigma\, E \to \omega', \texttt{sys}_\sigma\, E'} \qquad (\texttt{SYS}_2)\ \omega, \texttt{sys}_\sigma\, u \to \omega \cdot \sigma(u), \texttt{null}$$

$$(\texttt{METH}_1)\ \frac{\omega, E \to \omega', E''}{\omega, \texttt{E.m}(E') \to \omega', \texttt{E}''.\texttt{m}(E')} \qquad (\texttt{METH}_2)\ \frac{\omega, E' \to \omega', E''}{\omega, \texttt{E.m}(E') \to \omega', \texttt{E.m}(E'')}$$

$$(\texttt{METH}_3)\ \frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = \texttt{x}, E}{\omega, (\texttt{new}\,\texttt{C}(\bar{v})).\texttt{m}(v') \to \omega, E[v'/x, (\texttt{new}\,\texttt{C}(\bar{v}))/\texttt{this}]}$$

$$(\texttt{INT})\ \frac{\omega, E \to \omega', E'}{\omega, I_\alpha(E) \to \omega', I_\alpha(E')} \qquad (\texttt{IMPC}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{icast}\, E \to \omega', \texttt{icast}\, E'}$$

$$(\texttt{IMPC}_2)\ \frac{\texttt{new}\, C(\bar{v}) \in \mathit{receiver}(\alpha)}{\omega, \texttt{icast}\, I_\alpha(u) \to \omega, \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

$$(\texttt{EXPC}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{ecast}\,\texttt{C}\, E \to \omega', \texttt{ecast}\,\texttt{C}\, E'}$$

$$(\texttt{EXPC}_2)\ \frac{\texttt{new}\, C(\bar{v}) \in \mathit{receiver}(\alpha)}{\omega, \texttt{ecast}\,\texttt{C}\, I_\alpha(u) \to \omega, \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

$$(\texttt{DATA}_1)\ \frac{\omega, E \to \omega', E'}{\omega, E.\texttt{data} \to \omega', E'.\texttt{data}} \qquad (\texttt{DATA}_2)\ \omega, I_\alpha(v).\texttt{data} \to \omega, v$$

Intuitively, rule $(\texttt{PAR}_1)$ says that the first of two threads E can make a step and reduce to E'' extending the shared history ω to ω'. Needless to say, rule $(\texttt{PAR}_2)$ is symmetric to $(\texttt{PAR}_1)$. Rule $(\texttt{PAR}_3)$ can be applied when both the threads have reduced to a value and says that only one of them is returned, i.e., v'. When a system call is performed (rule $(\texttt{SYS}_2)$), the current history ω is extended by appending the symbol $\sigma(u)$. A method invocation requires the target object and the actual parameters to be reduced to values, then rule $(\texttt{METH}_3)$ can be ap-

Table 4.3: Semantics of expressions (Part 2).

$$(\texttt{IF}_1)\ \frac{\omega, E \rightarrow \omega', E''}{\omega, \texttt{if}(E = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} \rightarrow \omega', \texttt{if}(E'' = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\}}$$

$$(\texttt{IF}_2)\ \frac{\omega, E' \rightarrow \omega', E''}{\omega, \texttt{if}(v = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} \rightarrow \omega', \texttt{if}(v = E'')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\}}$$

$$(\texttt{IF}_3)\ \omega, \texttt{if}(v = v)\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} \rightarrow \omega, E_{tt}$$

$$(\texttt{IF}_4)\ \frac{v \neq v'}{\omega, \texttt{if}(v = v')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} \rightarrow \omega, E_{ff}}$$

$$(\texttt{SEQ}_1)\ \frac{\omega, E \rightarrow \omega', E''}{\omega, E; E' \rightarrow \omega', E''; E'} \qquad (\texttt{SEQ}_2)\ \omega, v; E \rightarrow \omega, E$$

$$(\texttt{CAST}_1)\ \frac{\omega, E \rightarrow \omega', E'}{\omega, (\texttt{C})E \rightarrow \omega', (\texttt{C})E'} \qquad (\texttt{CAST}_2)\ \frac{D <: C}{\omega, (\texttt{C})\texttt{new}\,\texttt{D}(\bar{v}) \rightarrow \omega, \texttt{new}\,\texttt{D}(\bar{v})}$$

$$(\texttt{PAR}_1)\ \frac{\omega, E \rightarrow \omega', E''}{\omega, \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \rightarrow \omega', \texttt{thread}\,\{E''\}\,\texttt{in}\,\{E'\}}$$

$$(\texttt{PAR}_2)\ \frac{\omega, E' \rightarrow \omega', E''}{\omega, \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \rightarrow \omega', \texttt{thread}\,\{E\}\,\texttt{in}\,\{E''\}}$$

$$(\texttt{PAR}_3)\ \omega, \texttt{thread}\,\{v\}\,\texttt{in}\,\{v'\} \rightarrow \omega, v'$$

plied. Briefly, it reduces the method invocation to the evaluation of the method body E where the formal parameters have been replaced by the actual ones (also including the special variable `this`). Note that the function *mbody*, returning the body and parameters names of a method, also deals with methods lookup (see [20] for more details). Finally, the table reports rules for explicit ($\texttt{EXPC}_2$) and implicit ($\texttt{IMPC}_2$) intents propagation. Firing an explicit intent causes the system to check whether there exists a receiver $\texttt{new}\,C(\bar{v})$ of the specified class, then the configuration reduces to the execution of the special method `receive`[*]. Implicit intents are handled in a similar way. The only difference is that the receiver's class C is not fixed by the rule and the system is responsible for gen-

[*]We reserve the keyword $\texttt{I}_\alpha$ for the type of the parameter of method `receive`

erating it. In both cases, if no suitable receiver exists, the expressions reduce to `null`.

The behavior of the other expressions under the rules of the proposed operational semantics is informally described.

The expression $\texttt{new}\ \texttt{C}(\bar{E})$ accepts reductions for the expressions in $\bar{E}$ (in the specified order), until they all reduce to values. A field access $E.\texttt{f}$ reduces along with E and, when E is an object value, reduces to the corresponding field value. Intents $I_\alpha(E)$ can reduce until their data expression E is a value which can be accessed through $E.\texttt{data}$. A sequence $E; E'$ behaves like E as long as it is not a value (which is discharged) and then behaves like E'. Class cast $(\texttt{C})E$ can reduce along with E and, when E reduces to an object, the cast operator can be removed if it is an instance of a subclass of C. Finally, the conditional branch admits reductions for the expressions making its guard (E before E') and then reduces to one between E_{tt}, if the equality check succeeds, and E_{ff}, otherwise.

Example 7. *Consider again the classes of Example 6. Simulating the computation of*

$$E \doteq \texttt{icast}\ \texttt{I}_{\texttt{www}}(\texttt{"http://site.org"})$$

starting from the empty history and assuming the class `Browser` *to be the only receiver in the system.*

$$\begin{array}{ll} \cdot, \texttt{icast}\ \texttt{I}_{\texttt{www}}(\texttt{"http://site.org"}) & \rightarrow \\ \cdot, \texttt{system}_{\texttt{connect}}\ \texttt{I}_{\texttt{www}}(\texttt{"http://site.org"}).\texttt{data}; & \rightarrow \end{array} \tag{4.1}$$

$$\begin{array}{ll} \cdot, \texttt{system}_{\texttt{connect}}\ \texttt{"http://site.org"}; & \rightarrow \\ \texttt{system}_{\texttt{connect}}\ \texttt{"http://site.org"}, \texttt{null} & \end{array} \tag{4.2}$$

The initial reduction step (4.1) corresponds to an implicit intent casting operation. Since `Browser` *is the only valid receiver for* `www`*, the execution reduces to the body of its method* `receive` *(see Example 6) where the formal parameter has been replaced by the actual one. Finally, in (4.2) the data carried by the intent is extracted (left side) and used (right side) to fire a new system call, i.e.,* `connect`. □

4.1.2 Type and Effect

A type and effect system for Fareweight Java has been previously described in [27]. I extend it with rules for handling Android-specific instructions, e.g., for intents management.

Table 4.4: Semantics of history expressions

$$\sigma(u) \xrightarrow{\sigma(u)} \varepsilon \qquad \alpha_\chi(u) \xrightarrow{\alpha_\chi(u)} \varepsilon \qquad \frac{H \xrightarrow{\alpha_\chi(u)} H'' \quad \dot{H} = \sum H'\{\alpha_?(u)/h\} \text{ s.t. } \bar{\alpha}_C h.H' \in \rho(\alpha) \text{ and } \chi \succcurlyeq C}{H \dot{\rightarrow} \dot{H}}$$

$$\frac{H' \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H \parallel H''} \qquad \frac{H \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H'' \parallel H'} \qquad \frac{H \xrightarrow{a} H''}{H \cdot H' \xrightarrow{a} H'' \cdot H'}$$

$$\frac{H \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''} \qquad \frac{H' \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''} \qquad \frac{H\{\mu h.H/h\} \xrightarrow{a} H'}{\mu h.H \xrightarrow{a} H'}$$

History expressions

The type and effect system extracts *history expressions* from programs. History expressions model computational agents in a process algebraic fashion in terms of the traces of events they produce. The syntax of history expressions follows.

Definition 6. *(Syntax of history expressions)*

$$\begin{aligned} H, H' ::= \quad & \varepsilon \mid h \mid \alpha_\chi(u) \mid \bar{\alpha}_C h.H \mid \sigma(u) \mid \\ & H \cdot H' \mid H + H' \mid H \parallel H' \mid \mu h.H \end{aligned}$$

Briefly, they can be empty ε, variables h, h', intent emissions $\alpha_\chi(u)$ (with $\chi \in \{C, ?\}$), intent receptions $\bar{\alpha}_C h.H$, system accesses $\sigma(u)$, sequences $H \cdot H'$, choices $H + H'$, parallel executions $H \parallel H'$ or recursions $\mu h.H$.

The semantics of history expressions is defined through a *labelled transition system* (LTS) according to the rules in Table 4.4.

Roughly, a system access $\sigma(u)$ (rule *system*) fires a corresponding event and reduces to ε. Similarly, a intent generation $\alpha_\chi(u)$ (rule *intent*) causes a event and reduces to ε. Intent exchange requires a receiver $\bar{\alpha}_C h.H'$ to be compatible with the intent destination χ. Compatibility is checked through the relation $\cdot \succcurlyeq \cdot$ (s.t. for all C, $C \succcurlyeq C$ and $? \succcurlyeq C$). The function ρ is analogous to *receiver* and I will describe how it is structured below. Concurrent agents $H \parallel H'$ admit either a reduction for left or right component (rules *l-parallel*, *r-parallel*). A sequence $H \cdot H'$ behaves like H until it reduces to ε and then reduces to H' (rule *sequence*). Instead, the non deterministic choice $H + H'$ can behave like either H or H' (rule *choice*). Finally, recursion $\mu h.H$ has the same behavior as H where the free instances of h are replaced by $\mu h.H$ (rule *recursion*). When necessary, I also write $H \xrightarrow{\omega}{}^{*} H'$ as a shorthand for $H \xrightarrow{a_1} \ldots \xrightarrow{a_n} H'$ with $\omega = a_1 \cdots a_n$.

Moreover, a partial relation order over history expressions, denoted by $\sqsubseteq$, is introduced as: $H \sqsubseteq H'$ iff $H \xrightarrow{a_1} \ldots \xrightarrow{a_n} H''$ implies there exists $\tilde{H}$ s.t. $H' \xrightarrow{a_1}$

$\ldots \xrightarrow{a_n} \tilde{H}$. Also, I can write $H \equiv H'$ as a shorthand for $H \sqsubseteq H'$ and $H' \sqsubseteq H$ (see [28] for more details).

Type and effect system

In the extension proposed in [29], the type and effect system is extended with rules for handling Android-specific instructions, e.g., for intents management.

Before presenting the type and effect system, two preliminary definitions of *types* and *type environment* need to be introduced.

Definition 7. *(Types and type environment)*

$$\tau, \tau' ::= \mathbf{1} \mid \mathcal{U} \mid \mathcal{I}_\alpha(\mathcal{U}) \mid C \qquad \Gamma, \Gamma' ::= \varnothing \mid \Gamma\{\tau/x\}$$

Types can be the unit one $\mathbf{1}$, a set of resources $\mathcal{U}$, a intent type $\mathcal{I}_\alpha(\mathcal{U})$ or a class type C. Instead, a type environment is a mapping from variables to types.

In the following, the typing rules of the proposed type and effect system are introduced. A typing judgment has the form $\Gamma \vdash E : \tau \triangleright H$ meaning that "under environment Γ, the expression E has type τ and effect H".

Table 4.5 reports typing rules. Implicit intent casting has type unit and effect $H \cdot \sum_{u \in \mathcal{U}} \alpha_?(u)$ where H is obtained by typing the intent expression E and $\sum_{u \in \mathcal{U}} \alpha_?(u)$ is an abbreviation for $\alpha_?(u_1) + \ldots + \alpha_?(u_n)$ with $\mathcal{U} = \{u_1, \ldots, u_n\}$. The rule for explicit intents is similar, but it uses the receiver class C instead of the wild card $?$ to label the intents.

Method invocations, rule $(\texttt{T-METH})$ require more attention. In fact, the invocation $E.\texttt{m}(E')$ has type F' and effect $H \cdot H' \cdot H''$ where F' is the the type of the return expression E'', H is the effect generated by E, H' the effect for E' and H'' for E''. Also, E'' is typed under the environment $[C/\texttt{this}, D'/\texttt{x}]$ and the function $msign(\texttt{m}, C)$ returns the signature of a method, i.e., $C \to D$ for a method declaring input type C and return type D* (for the definition of the subtype relation <: see [20]).

System calls $(\texttt{T-SYS})$ have unit type $\mathbf{1}$ and their effect correspond to effect for their parameter E followed by the choice among all the valid instantiation of the access event σ. Instead, concurrent executions have the same type of the second expression, i.e., E', and effect equal to the parallel composition of the effects of the two sub-processes. The rule called *weakening* $(\texttt{T-WKN})$ allows for effect generalization. In words, the rule says that an expression with an effect H' can also be can represented with a more general one H, in symbols $H' \sqsubseteq H$. Intuitively, `null` has unit type and empty effect, a resource u has empty effect

*For a class C<:`Receiver` can be expressed as $msign(\texttt{receive}, C) = \mathcal{I}_\alpha \to \mathbf{1}$

Table 4.5: Typing rules.

$$(\texttt{T-NULL})\ \Gamma \vdash \texttt{null} : \mathbf{1} \rhd \varepsilon \qquad (\texttt{T-RES})\ \Gamma \vdash u : \{u\} \rhd \varepsilon \qquad (\texttt{T-VAR})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rhd \varepsilon}$$

$$(\texttt{T-INT})\ \frac{\Gamma \vdash E : \mathcal{U} \rhd H}{\Gamma \vdash I_\alpha(E) : \int \alpha(\mathcal{U}) \rhd H} \qquad (\texttt{T-DATA})\ \frac{\Gamma \vdash E : \int \alpha(\mathcal{U}) \rhd H}{\Gamma \vdash E.\texttt{data} : \mathcal{U} \rhd H}$$

$$(\texttt{T-FLD})\ \frac{\Gamma \vdash E : C \rhd H \quad \mathit{fields}(C) = \bar{D}\bar{f}}{\Gamma \vdash E.\texttt{f}_\texttt{i} : D_i \rhd H} \qquad (\texttt{T-IMPC})\ \frac{\Gamma \vdash E : \int \alpha(\mathcal{U}) \rhd H}{\Gamma \vdash \texttt{icast}\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \alpha_?(u)}$$

$$(\texttt{T-EXPC})\ \frac{\Gamma \vdash E : \int \alpha(\mathcal{U}) \rhd H}{\Gamma \vdash \texttt{ecast}\, \texttt{C}\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \alpha_C(u)} \qquad (\texttt{T-SYS})\ \frac{\Gamma \vdash E : \mathcal{U} \rhd H}{\Gamma \vdash \texttt{system}_\sigma\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \sigma(u)}$$

$$(\texttt{T-NEW})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash \texttt{new}\,\texttt{C}(\bar{E}) : C \rhd H_1 \cdots H_n} \qquad (\texttt{T-CAST})\ \frac{\Gamma \vdash E : D \rhd H \quad D <: C}{\Gamma \vdash (\texttt{C})E : C \rhd H}$$

$$(\texttt{T-IF})\ \frac{\Gamma \vdash E : \tau \rhd H \quad \Gamma \vdash E' : \tau' \rhd H' \quad \Gamma \vdash E_{tt} : \dot{\tau} \rhd \dot{H} \quad \Gamma \vdash E_{ff} : \dot{\tau} \rhd \dot{H}}{\Gamma \vdash \texttt{if}(E = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{ff}\} : \dot{\tau} \rhd H \cdot H' \cdot \dot{H}}$$

$$(\texttt{T-METH})\ \frac{\begin{array}{c}\Gamma \vdash E : C \rhd H \quad \Gamma \vdash E' : \tau' \rhd H' \quad \mathit{mbody}(\texttt{m}, C) = \texttt{x}, E'' \\ {[C/\texttt{this}, \tau'/\texttt{x}]} \vdash E'' : F' \rhd H'' \quad \mathit{msign}(\texttt{m}, C) = \tau' \to \tau'' \quad \dot{\tau}' \to \dot{\tau}'' <: \tau' \to \tau''\end{array}}{\Gamma \vdash E.\texttt{m}(E') : \tau'' \rhd H \cdot H' \cdot H''}$$

$$(\texttt{T-SEQ})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash E_1; E_2 : \tau_2 \rhd H_1 \cdot H_2} \qquad (\texttt{T-PAR})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash \texttt{thd}\,\{E_1\}\,\texttt{in}\,\{E_2\} : \tau_2 \rhd H_1 \| H_2}$$

$$(\texttt{T-WKN})\ \frac{\Gamma \vdash E : \tau \rhd H' \quad H' \sqsubseteq H}{\Gamma \vdash E : \tau \rhd H}$$

and type equal to the singleton $\{u\}$, the type of a variable x is assigned by the type environment Γ, an object $\texttt{new}\, C(\bar{E})$ has the type of its class C and effect equal to the sequence of the effects produced by the instantiation parameters $\bar{E}$. Field access $E.\texttt{f}$ has the type of $\texttt{f}$ and the effect for E while intent creation $I_\alpha(E)$ has type $\mathcal{I}_\alpha(\mathcal{U})$ (where $\mathcal{U}$ is the type of E) and the effect of E. Similarly to field access, intent content reading $E.\texttt{data}$ has the type of the intent (denoted by E) data and effect equal to that of E. Finally, the effect of sequences is the sequence of the two sub-effects for E and E' (while the type is that of E') and the effect of choices is the sequence of the two effects generated by the expressions in the guard and an effect generalizing those of the two branches, i.e., E_{tt} and E_{ff} (while the type is the same of E_{tt} and E_{ff}). More details about these and other similar rules for types and effects can be found in [27, 30, 28].

As expected, well-typed expressions do not produce erroneous computations, i.e., they always return a value or admit further reductions.

Lemma 1. *For each* closed *(i.e., without free variables) expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \triangleright H$ then either E is a value or $\omega, E \to \omega', E'$ (for some ω', E').*

Proof. (Sketch*) By induction over the structure of E. □

Also well-typed expressions generate history expressions which *safely* denote the runtime behavior of expressions. In particular, the following theorem states that typing a (closed) expression E I obtain an over-approximation of the set of all the possible executions of E.

Theorem 1. *For each* closed *expression E, history expression H, type τ and trace ω, if $\varnothing \vdash E : \tau \triangleright H$ and $\cdot, E \to^* \omega, E'$ then there exist H' and ω' such that $H \xrightarrow{\omega'}{}^* H'$, $\varnothing \vdash E' : \tau \triangleright H'$ and $\omega = \omega'$.*

Proof. (Sketch) The proof starts by proving that the property holds for single-step reductions. Then, it proceeds by induction on the derivations length. □

I extend the type and effect system to method declarations in the following way. Given a class C and a method $\mathtt{m}$ (s.t. $\mathtt{m} \neq \mathtt{receive}$) such that $mbody(\mathtt{m}, C) = \mathtt{x}, E$, $msign(\mathtt{m}, C) = D \to F$ and $[C/\mathtt{this}, D/\mathtt{x}] \vdash E : F \triangleright H$, I write $\vdash C.\mathtt{m} : D \xrightarrow{H} F$ and we say H to be the *latent effect* of $\mathtt{m}$. Instead, if C<: `Receiver` and $\mathtt{m} = \mathtt{receive}$ I write $\vdash C.\mathtt{m} : \mathcal{I}_\alpha \xrightarrow{\bar{\alpha}_C h.H} \mathbf{1}$.

Finally, I exploit the typing rules for the generation of the function ρ described above. For each existing receiver, we type the corresponding `receive` method and so to obtain

$$\rho(\alpha) = \left\{ \bar{\alpha}_C h.H \;\middle|\; \begin{array}{l} \mathtt{new}\, C(\bar{v}) \in receiver(\alpha) \quad \wedge \\ \vdash C.\mathtt{receive} : \mathcal{I}_\alpha \xrightarrow{\bar{\alpha}_C h.H} \mathbf{1} \end{array} \right\}$$

4.1.3 Discussion

Although simplified, the proposal of extending Featherweight Java for the Android Application framework includes most of the relevant aspects of the Android IPC and components.

*Technical proofs can be found in Appendix.

Although feasible, redefining the type system for the Java language (and also for its bytecode) can be a cumbersome process (due to its reach and redundant syntax) since defining such a type effect system would require defining rules for more than 200 operation codes*. Not to mention, the Java language is provided with no formal semantics which is necessary for defining a type and effect system. Nevertheless, some alternatives exist. As a matter of fact, several bytecode analysis frameworks apply to simplified languages. For instance, ConFlEx [31] translates bytecode programs into BIR, an intermediate, stack-less language. Similarly, Soot [32] relies on Jimple (Java sIMPLE), a stack-less, three address language. Clearly, defining a type and effect system for these simplified languages is easier than for the full Java/bytecode language.

4.2 Control flow graphs

In [33] the authors opted for generating history expressions out of *control flow graphs* (CFG). This solution voids the formal guarantee about the soundness of the generated models. Instead, it relies on the assumption that the Java runtime complies with its specifications. Following a similar reasoning, the Secure Meta-Market relies on Control Flow Graphs for the model extraction of Android applications.

A control flow graph is a data structure representing the possible execution flows of a piece of software. Intuitively, control flow instructions, i.e., conditional branches, unconditional jumps and loops, result in edges connecting the nodes of the CFG.

Nodes can be either basic blocks, i.e., linear, jump-free sequences of instructions, or conditional nodes, i.e., those representing the guard of a branching instruction. The level of abstraction of a CFG mostly depends on the instructions appearing in its nodes.

Regarding Android applications, two sets of instructions Δ and Γ, denoting the security-relevant relevant operations and the IPC invocations, respectively, are defined. Clearly, the content of Δ corresponds to the alphabet of the security policy under evaluation and typically changes when a new policy is considered. Instead, Γ is constantly defined as the set of the Android IPC APIs plus a generic silent action τ. To disambiguate, $\underline{\alpha}, \overline{\beta} \in \Gamma$ denotes that α is an incoming action and β an outgoing one.

The following simple example will provide the basic intuition.

*http://docs.oracle.com/javase/specs/jvms/se8/html/index.html

Example 8. *Consider the fragment of Java code of an Android component in Listing 4.1. It represents a minimal implementation of an Android* `BroadcastReceiver`*. It defines a method* `onReceive` *which can be triggered by sending an appropriate intent through the invocation* `sendBroadcast`*.*

Briefly, the fragment above invokes the URL class constructor, i.e., `<init>`*, and an* `InputStream` *is obtained from it. Then, the application creates a temporary file and opens an* `OutputStream` *on it. Finally, the code iterates over available bytes in the URL stream, reading them one by one, and writes each byte in the (stream pointing to the) temporary file.*

Upon compilation, a piece of bytecode, resembling that reported in Listing 4.2, is generated. Method invocations (instruction `ivk`*) contain a pointer to the invoked method (e.g.,* `.startService(Intent)`*). For the sake of brevity, verbose parameters were substituted with the ... placeholder. The other involved instructions are* `ldc` *(load a constant in a register),* `new` *(creates an instance of the given class type),* `goto` *(jumps to the given instruction), and* `ret` *(exit the current block).*

Listing 4.1: Code fragment of an Android component.

```
public class C extends BroadcastReceiver
{
 public void onReceive(Context c, Intent i) {

  URL url = new URL(i.getStringExtra("url"));
  InputStream uis = url.openConnection().getInputStream();
  File file = File.createTempFile(null,
    null,c.getCacheDir());
  FileOutputStream fos = new FileOutputStream(file);
  while(uis.available() > 0)
    fos.write(uis.read());

   Intent j = new Intent(c, ABC.class);
   c.startService(j);
  }
}
```

Listing 4.2: Simplified Bytecode of an Android component.

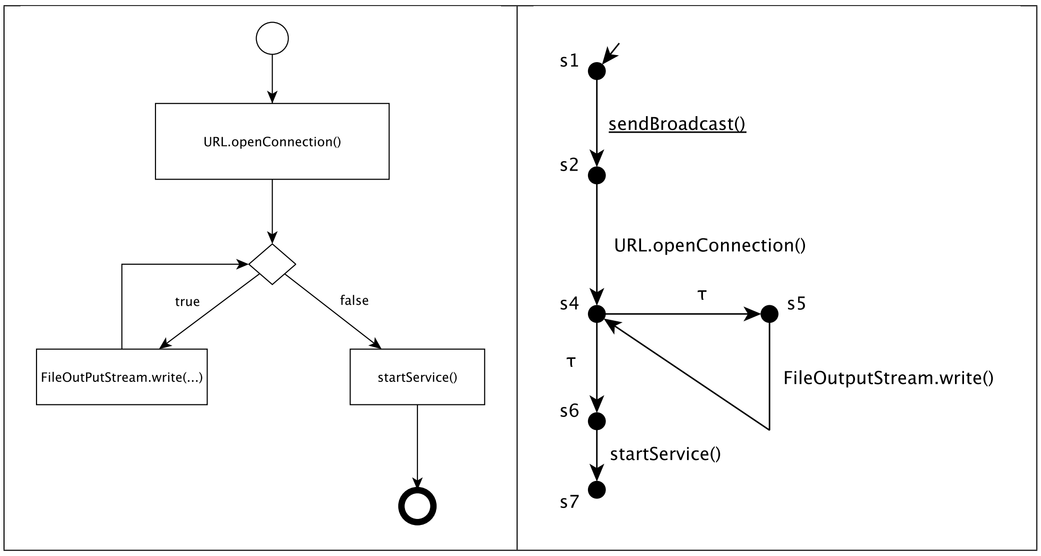
```
public onReceive(android.content.Context arg0,
  android.content.Intent arg1) {
  L1 {
    new java/net/URL
    ldc "url" (java.lang.String)
    ivk android/content/Intent getStringExtra((...)
      Ljava/lang/String;);
    ivk java/net/URL <init>(...);
    ivk java/net/URL openConnection(
                    ()Ljava/net/URLConnection;);
    ivk java/net/URLConnection getInputStream(...);
    new java/io/FileOutputStream
    ivk android/content/Context getCacheDir(...);
    ivk java/io/File createTempFile(...);
    ivk java/io/FileOutputStream <init>((Ljava/io/File;)V);
  }
  L2 {
    ivk java/io/InputStream available(()I);
    ivk java/io/InputStream read(()I);
    ivk java/io/FileOutputStream write((I)V);
  }
  L3 {
    goto L2
  }
  L4 {
    ldc Lcom/example/testapplication/ABC; (...)
    ivk android/content/Intent <init>(...);
    ivk android/content/Context startService(...);
    ret
   }
}
```

The CFG obtained from the given bytecode could resemble that on the left of Fig. 4.1, where $\Delta = \{$`URL.openConnection`, `FileOutputStream.write`$\}$[*]. *The nodes of the CFG correspond to the basic blocks of the bytecode.*

Moreover, a branching node $\diamond$ *is placed where the control flow can take two different paths. In this example, the instruction* `.available` *is interpreted as testing whether data can be written of the file. Also notice that the actions not*

[*]Notice that, for the sake of presentation, in this examples polyadic actions are used instead of monadic ones. The interested reader can find more details on this aspect in [34, 35].

Figure 4.1: A (simplified) example of the model extraction process (contd.).



appearing in $\Delta \cup \Gamma$ *(here the only actions in* Γ *which also appear in the CFG are* `startService` *and* `sendBroadcast`*) are dropped from the nodes.*[*]

□

CFGs are exploited in several analysis techniques and many authors proposed algorithms for the extraction of *safe*, i.e., correctly representing all the execution paths, graphs. As a consequence, several tools exist for extracting a CFG from binaries and bytecode. Among them, Androguard[†] and ConFlEx [31] are the most suitable for the SMM aim. Androguard consists of a suite of tools for the static analysis of Android applications and is specifically designed for processing the Android bytecode. Instead, ConFlex generates CFGs from Java bytecode even when some components are statically unspecified. Moreover, ConFlEx implements an *incremental* CFG building procedure which is both sound and rather precise. CFGs obtained in this way can be composed when a missing module becomes available so resulting in model refinement. Needless to say, this features is very appealing when extracting models from Android applications as they often use IPC for invoking statically unknown components. When a component of an application is dovetailed with others to form an execution context, their models compose in a single, refined one.

[*]Notice that these operations are removed through simplifications, rather than simply discharged. For instance, *constant propagation* [36, 37] can be applied before dropping references to variables.

[†]`https://github.com/androguard/androguard`

4.2.1 CFG to History Expression Conversion

After the extraction process, each CFG is converted into a history expression having the same semantic (following the same approach of [33]). In particular, the set of history expressions that represent the application components are composed by using the parallel operator ||, as discussed in Section 4.1.2. Example 8 clarifies the translation process.

Example 9. *Consider again the CFG of Figure 4.1 extracted through the steps sketched in Example 8. The CFG is converted into the LTS graphically depicted on the right of Fig. 4.1. Each transition is labeled with the action appearing in the corresponding node of the CFG. The conditional node generates two silent actions τ, representing the non-deterministic choice between the two branches of the while statement. Since the receiver begins its computation upon receiving an invocation, it is extended with the initial transition* `sendBroadcast`. *Such transition states that the computation of this LTS cannot start autonomously, but needs to be triggered by an intent.* □

4.3 ConSpec Policy Language

ConSpec has been specifically designed for high level languages, e.g., Java, which makes it suitable for the Android environment. The main purpose of ConSpec is to be a formal policy specification language suitable for both verification and monitoring of safety properties. In this respect, it inherits some of the features of other, commonly used policy specification languages, e.g., PSLang [38] and Polymer [39]. As a consequence, in the last years several authors proposed security frameworks using or being compatible with ConSpec, e.g., see [40, 41, 42, 43].

The following of this section presents ConSpec, the current language used by the Secure Meta Market, as discussed in [8] and [14].

4.3.1 ConSpec Syntax

A ConSpec policy consists of two blocks: (i) the security state and (ii) the policy rules. The security state contains a list of variable declarations. Each declaration consists of a scope, i.e., `SESSION`, `MULTISESSION` and `GLOBAL`, a type domain, e.g., `Bool` and `Nat[`n`]`, a variable name and an initial value. The scope defines how values are assigned to a variable according to the life cycle of the policy target, i.e., a running application. In practice, a `SESSION` variable can be modified only by the target application and it is reinitialized at each new execution. Similarly, `MULTISESSION` denotes a variable which is only modifiable by the application, but when the target terminates its last value is saved and reused when the application is launched again. Finally, `GLOBAL` variables are shared by all the applications referring to them and their value is persistent. As in [44], I use finite, lifted variable domains, that is variables can only assume a finite number of distinct values including the indefinite one $\perp$. However, I define domain size individually at each declaration, e.g., `Nat[3] x`, rather than using a single size declaration as in [44][*]. Intuitively, `Bool` denotes $\{\perp_{\texttt{Bool}}, \texttt{true}, \texttt{false}\}$, `Nat[`$n$`]` denotes $\{\perp_{\texttt{Nat}}, 0, \ldots, n\}$, `Str[`$n$`]` stands for $\{\perp_{\texttt{Str}}\} \cup \bigcup_{0 \leq i \leq n} \{[a \ldots Z]^i\}$ and `Obj` represents $\{\perp_{\texttt{Obj}}, \texttt{loc}_1, \ldots, \texttt{loc}_N\}$ (being $\texttt{loc}_i$ distinct memory locations).

Rules describe how to react to a security-relevant action, e.g., a system access invocation. Actions are parametric and parameters can be used in the body of a rule. Three kinds of rule exist. A `BEFORE` rule is evaluated immediately before the mentioned action takes place, `AFTER` is evaluated when the action returns a result (which can be used by the rule) and `EXCEPTIONAL` for actions resulting in an exception.

Each rule contains a list of clauses of the form *guard* `->` *statement*. Guards are boolean expressions which can refer to the policy variables and the action parameters. Instead, statements are finite sequences of assignments (being `skip` the empty one) used to update the policy state. Following the *default-deny* approach, a ConSpec policy fails when the next observed action does not activate any clause in any rule.

Example 10. *Consider a security policy that states that "After the connection to an url website, no local data can be saved on the device". The corresponding ConSpec policy is provide in Listing 4.3.*

The first rules is used to trace the outgoing connection through the `openConnection()` *API with a boolean variable* `connect`*. Then, the second rule prevents the creation of a* `FileOutStream` *to save local data when an internet connection has been previously opened.*

As a matter of fact, the real policy should include further statements (e.g., ruling all the available URL connection methods and the existing device-to-device interfaces).

$\square$

[*] Since we can always factorize domains size and use the maximum value, this does not change the expressive power of the language.

Listing 4.3: Example of ConSpec policy.

```
SECURITY STATE
SESSION Bool connected = false;

AFTER java.net.URL.openConnection()
PERFORM
(true) -> { connected := true; }

BEFORE java.io.FileOutputStream.write(int i) PERFORM
  (!connected) -> { skip; }
```

4.3.2 ConSpec Semantics

ConSpec semantics is given in terms of a particular class of security automata, called *ConSpec automata* [45]. In ConSpec automata, security relevant actions are method calls, represented by the class name and the method name of the method, along with a sequence of values that represent the actual argument list of the method. The set of security relevant actions is partitioned into a set of before actions Λ^b and a set of after actions Λ^a, corresponding to method invocations and returns.

Both refer to the program state prior the method invocation, while the latter also refers to the state after the method execution and the return value.

Definition 8. *A ConSpec automaton is a tuple $A = (Q, \Lambda, \delta, q_0)$ where:*

- *Q is a countable set of states,*
- *$\Lambda = \Lambda^b \cup \Lambda^a$ is a countable set of security relevant actions as described above,*
- *$\delta = \delta^b \cup \delta^a$ is a (partial) transition function where $\delta^b : Q \times \Lambda^b \to Q$ and $\delta^a : Q \times \Lambda^a \to Q$, and*
- *$q_0 \in Q$ is the initial state.*

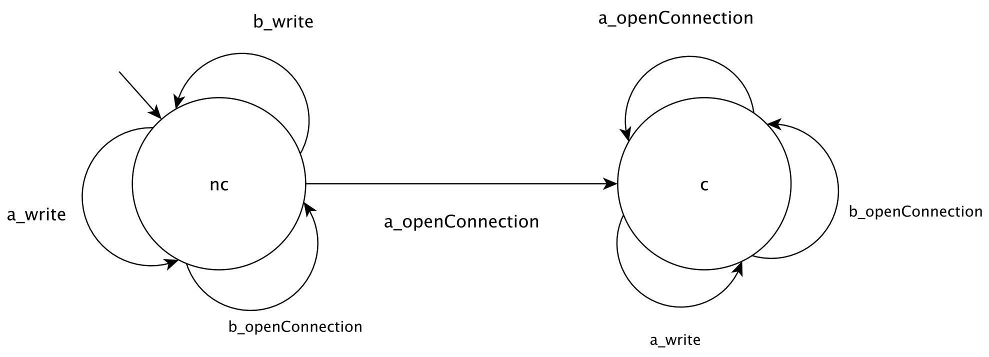
The translation from a ConSpec P to a ConSpec automaton A_P is given in [45]. Q is a set of labels $q_0, q_1, \ldots$ being the image of a function χ mapping each possible assignment of values to the n variables $x_1, \ldots, x_n$ in the `SECURITY STATE` to a unique name q_i. Intuitively, $q_0 = \chi(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the initial values of $x_1, \ldots, x_n$.

The actions A of the automaton are determined by the events mentioned in event clauses of the policy P. Each event clause of the policy induces a partial transition function. The transition functions δ^b and δ^a of the automaton are the union of the partial functions corresponding to event clauses with the BEFORE and AFTER modifier, respectively. For the sake of brevity, I label such transitions with $a\_$ o $b\_$ prefix for denoting Λ^a and Λ^b actions, respectively plus the name of the method. The following example clarifies such a mapping.

Example 11. *Consider the ConSpec policy of Example 10, the corresponding ConSpec automaton A_P is:*

- $Q = \{nc, c\}$,
- $q_0 = \{false\}$,
- $\Lambda = \{a\_openConnection, b\_openConnection, a\_write\, b\_write\}$,
- *δ as depicted in Figure 4.2.*

Figure 4.2: ConSpec automaton of the policy of Example 10



□

4.4 Application Verification

The following of this section presents the model checking technique, used by the Secure Meta-Market in [7] to evaluate the application model with respect to the security policy defined by the organization. Chapter 6 will detail the implementation choices.

In the last decades numerous tools and techniques for model checking were developed, like those surveyed in [46]. Among them, some successful techniques are based on some form of *reachability analysis* as described in [47] and [48].

The model checking can be mapped into a reachability problem, i.e. checking whether the model of the application cannot reach an undesirable state.

The reachability problem consists in identifying an execution trace (i.e. a path on a graph) from an initial state to a final accepting states that violates the policy. To this aim, such procedure needs a transition systems that contains both information about the state of system during a computation and the security policy.

Given a LTS model $M = (S, i, \Sigma, T)$ and a ConSpec automaton $A_p = (Q, \Lambda, \delta, q_o)$, the verification procedure consists of the following steps.

1. Build the Büchi automaton B that accepts all the traces that are not accepted by the the ConSpec automaton A_p, that is $B = (Q \cup \{f\}, \Lambda, \overline{\delta}, \{f\}, q_o)$ where $\overline{\delta} = \delta \cup \{(q, \lambda, f) \mid q \in Q \wedge \nexists q' \in Q \mid (q, \lambda, q') \in \delta\}. \cup \{(f, \lambda, f) \mid \lambda \in \Lambda\}$ and f is a final accepting state.

2. Extend the model M with the set of security relevant action of Λ, that is producing a LTS $M' = (S', i', \Sigma', T')$ where:

- $S' = S \cup \overline{S}$,
- $i' = i$,
- $\Sigma' = \Lambda$ and
- $T' = \{(s, b_\delta, \overline{s}) \mid (s, \delta, s') \in T \wedge \delta \in \Delta\} \cup \{(\overline{s}, a_\delta, s' \mid (s, \delta, s') \in T \wedge \delta \in \Delta\} \cup \{(s, \gamma, s') \mid (s, \gamma, s') \in T \wedge \gamma \in \Gamma\} \cup \{\tau\}$.

3. Compute the product automaton of the application model and the Büchi automaton in order to combine the transition of the model of the system with the evaluation of the policy. This step produces $\mathcal{M} = (\mathcal{S}, i'', \Sigma', \mathcal{T})$ where:

- $\mathcal{S} = \{(s, q) \mid s \in S' \wedge q \in Q \cup \{f\}$;
- $i'' = (i, q_0)$ and
- $\mathcal{T} = \{((s, q), \gamma, (s'q)) \mid (s, \gamma, s') \in T' \wedge \gamma \in \Gamma \cup \{\tau\}\}$
  $\cup \ \{((s, q), \delta, (s'q)) \mid (s, \delta, s') \in \overline{\delta}\}$.

4. Inspect the resulting product automaton, also referred as the *state space*, by using a search algorithm like depth-first search (DFS) [49]. According to [50], the product automaton has an acceptance execution (i.e. an execution trace) if and only if it has some state $f \in F$ that is reachable from the initial state. The path to the acceptance state constitutes the *counterexample* of the model checking.

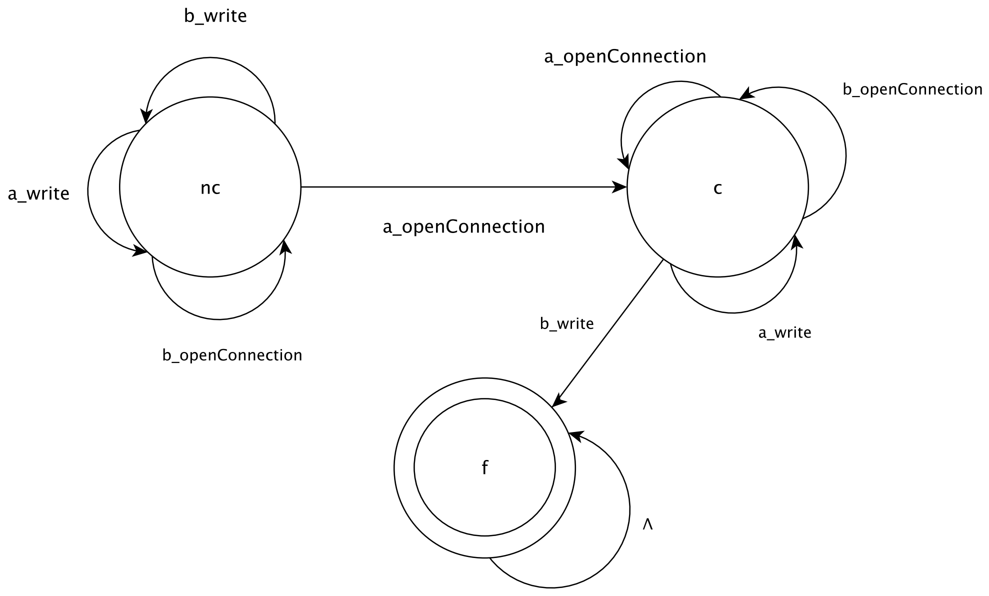
The following example will clarify the entire model checking procedure.

Example 12. *Consider the ConSpec automaton A_p of Figure 4.2 and the LTS of of Figure 4.1.*

1. *The Büchi automaton B that accepts the executing traces not accepted by A_p is:*

- $Q = \{nc, c, f\}$,
- $\Lambda = \{a\_openConnection, b\_openConnection, a\_write\, b\_write\}$,
- δ *as depicted in Figure 4.3,*
- $f = \{f\}$,
- $q_0 = \{not\_connected\}$.

Figure 4.3: Büchi automaton B.

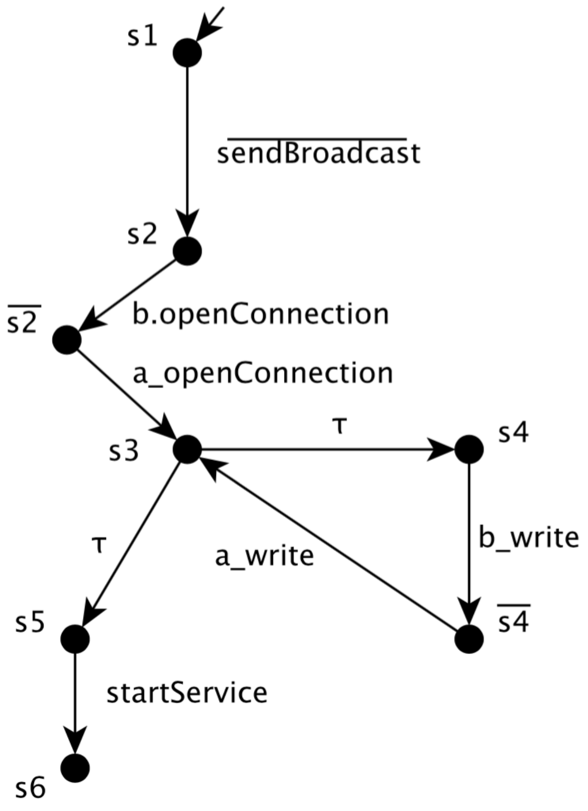


2. *The extension of the model M' is:*

- $S' = \{s_1, s_2, s_3, s_4, s_5, s_6, \overline{s_1}, \overline{s_2}, \overline{s_3}, \overline{s_4}, \overline{s_5}, \overline{s_6}\}$,
- $i = \{s_1\}$,

- $\Sigma' = \{a\_openConnection, b\_openConnection, a\_write\, b\_write\}$ *and*
- T' *as depicted in Figure 4.4. To improve readability, I omitted unreachable states, i.e.* $\overline{s_1}, \overline{s_3}, \overline{s_5}$ *and* $\overline{s_6}$.

Figure 4.4: Label Transition System M'.



3. *The result* product automaton $\mathcal{M}$ *obtained by composing* M' *and* B *is:*

- $\mathcal{S} = \{\{s_1, nc\}, \{s_1, c\}, \{s_1, f\}, \ldots, \{\overline{s_6}, nc\}, \{\overline{s_6}, c\}, \{\overline{s_6}, f\}\}$,
- $i'' = \{\{s_1, nc\}\}$ *and*
- $\mathcal{T}$ *as depicted in Figure 4.5. To improve the readability, I shorten the actions name to* $ss, \overline{sb}, b_w, a_w, b_o, a_o$. *Moreover, I remove unreachable states.*

4. *Given the product automaton* $\mathcal{M}$ *of Figure 4.5, the DFS algorithm is able to identify a violating trace* η, *represented with the red arrows in Figure 4.6.*

□

Figure 4.5: Product automaton $\mathcal{M}$.

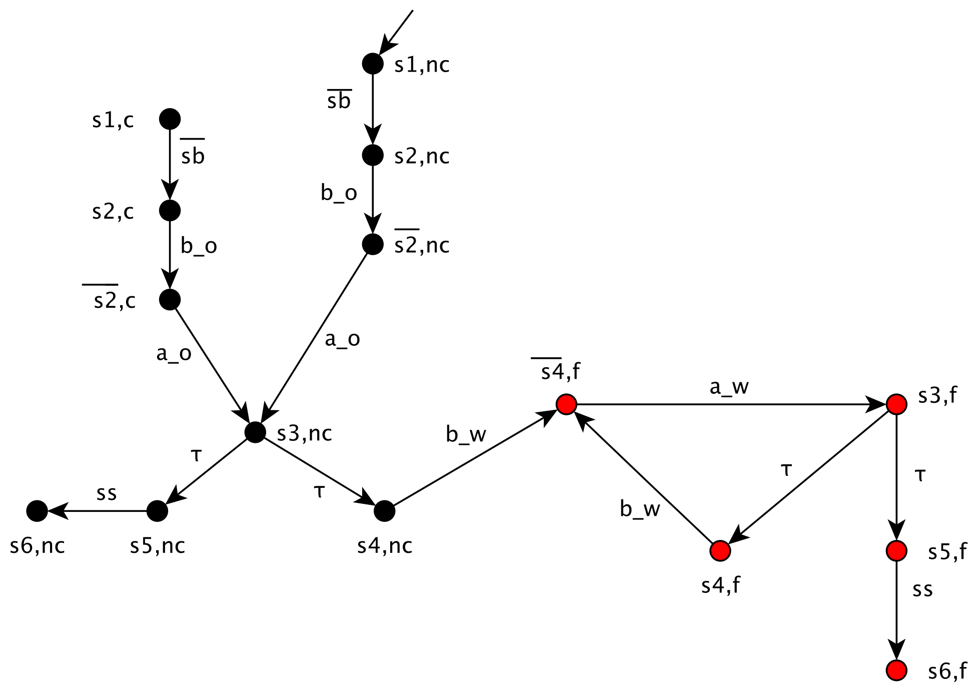
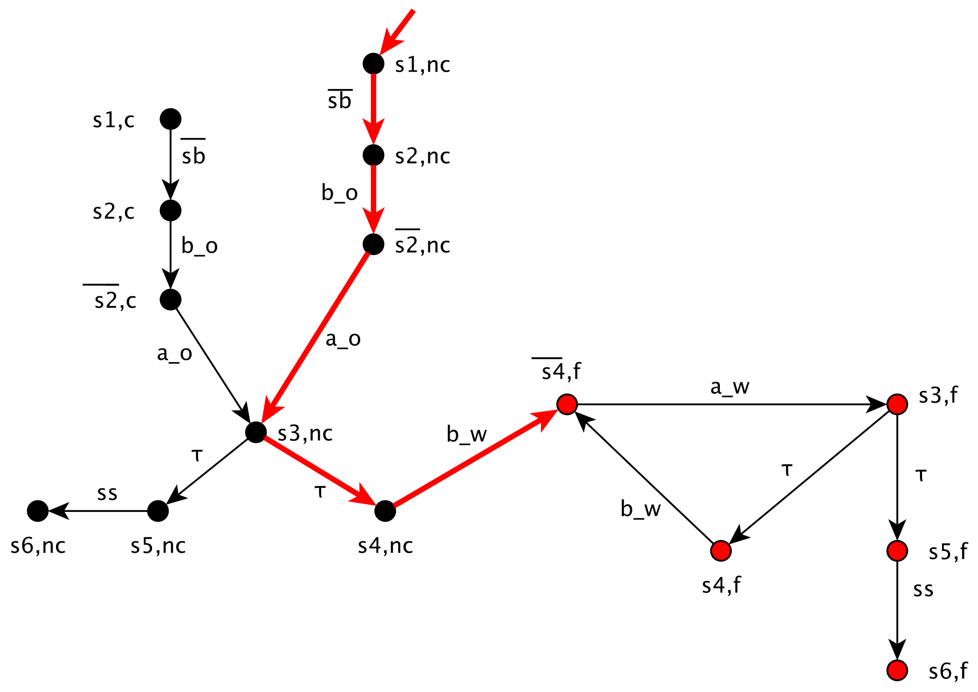


Figure 4.6: Violating trace η in the product automaton $\mathcal{M}$.



5

Application Monitoring

This Chapter presents the monitoring techniques for enforcing policy compliance at runtime. Finally, the Secure Meta-Market monitoring environment is discussed.

5.1 Mobile Application Monitoring

Application monitoring is a consolidated technique for enforcing policy compliance at runtime.

Several implementations appeared in the literature, also running on mobile, resource-constrained devices [51, 52]. Typically, a monitoring environment consists of a *policy decision point* (PDP) and one or more *policy enforcement points* (PEPs). The PDP holds the security policy and the current security state while the PEPs control the access to security critical operations. When an application attempts to perform a security-relevant access, a corresponding PEP is activated. The PEP triggers the PDP which checks the security state and policy and grants or rejects the permission. Finally, the PEP enforces the PDP's response by executing or blocking the requested access.

The PDP module can be implemented following two different approaches:

Distributed PDP. Each applications is included with the security policy and the enforcement code, as shown in Figure 5.1. Distributed PDPs grant less overhead and fastest verification times, since all the monitoring is made locally. However, the implementation of such architecture is more complex in case of application interactions and policy updates.

Figure 5.1: Architecture of distributed PDP.

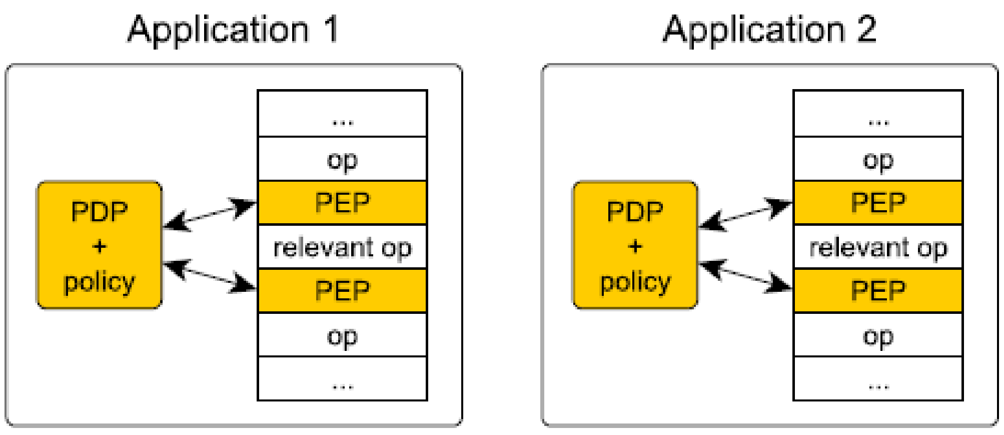
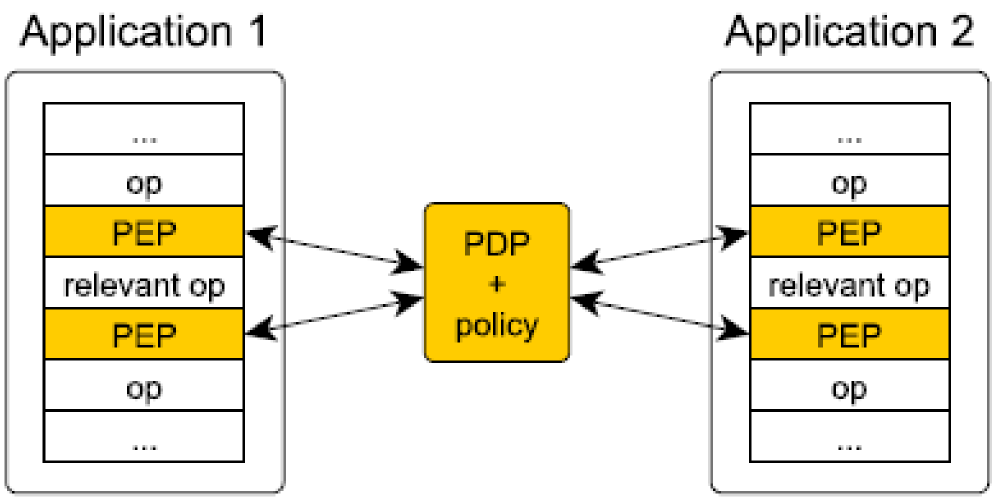


Figure 5.2: Architecture of centralized PDP.



Centralized PDP. This kind of PDP is implemented outside monitored applications, as depicted in Figure 5.2. This approach simplifies policy updates and data management.

The monitoring environment may differ for the PEPs deployment strategy. Traditionally, two approaches exist for enforcing policies on mobile applications.

Platform customization. PEPs are implemented as system components watching all the accesses to resources.

Application instrumentation. Pieces of code implementing the PEPs are injected in the instructions flow of each application.

Platform customization requires heavy modification of the mobile devices in order to replace/rewrite system components. For instance, for virtual machine-based systems, e.g., Java/Android, the customization consists in replacing the standard VM with a new one including the PEP functionality. Instead, under the same assumptions, the application instrumentation is much less invasive. Indeed, it only requires to slightly modify the intermediate code, i.e., the VM instructions, of the application. For a more detailed discussion about the PEP inlining for mobile applications see [52].

5.1.1 Application Instrumentation

PEP instrumentation consists in modifying the application by injecting the appropriate instructions directly inside the executable code. To this aim, two different approaches may be considered.

Direct Injection. PEPs are injected in the application bytecode, as show in Figure 5.3, directly wrapping security relevant invocations. Although simple, this approach jeopardize PEP enforcements and requires an heavy modification of the application executable.

Wrapping Injection. The security relevant invocations are wrapped into *API Wrapping Methods* and linked to the application executable as a Security Library. With this approach, the bytecode only needs to be modified with a link to the appropriate stub when required, as sketched in Figure 5.4.

Methods `checkBefore`, `checkException` and `checkAfter` implement the PEP logic by requesting authorizations to the PDP. Briefly, they interrupt the code execution and alert the PDP about the ongoing operation and its parameters. The PDP evaluates the action according to the current security state. The evaluation changes the security state and produces a return value for the PEP, i.e., *allow* or *deny*. If the PEP receive an *allow* signal, the execution is resumed, otherwise it throws a security exception. Listing 5.1 shows a code fragment obtained by instrumenting the instruction `Url url = new URL(addr)`. The interested reader can refer to [53] for further details.

Figure 5.3: PEP Direct Injection.

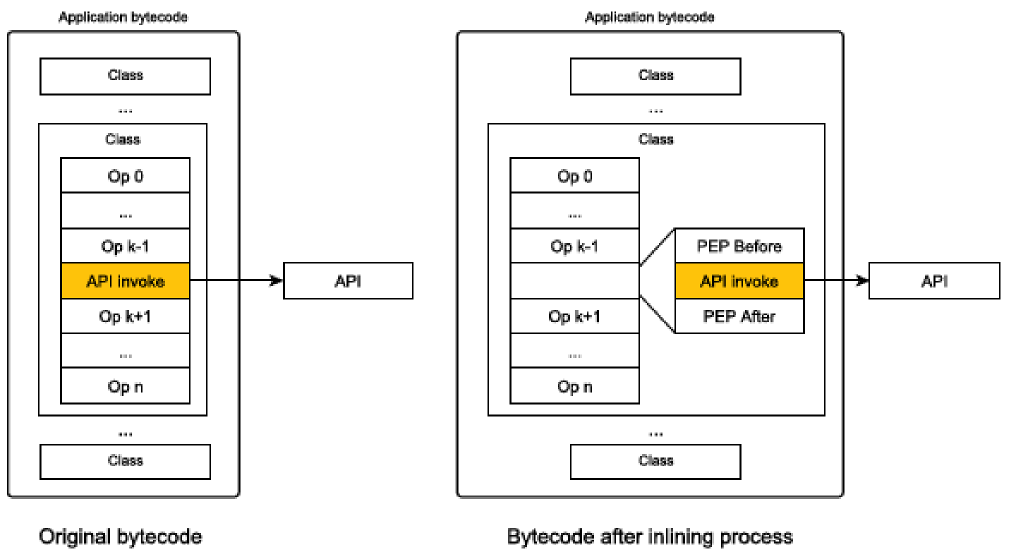
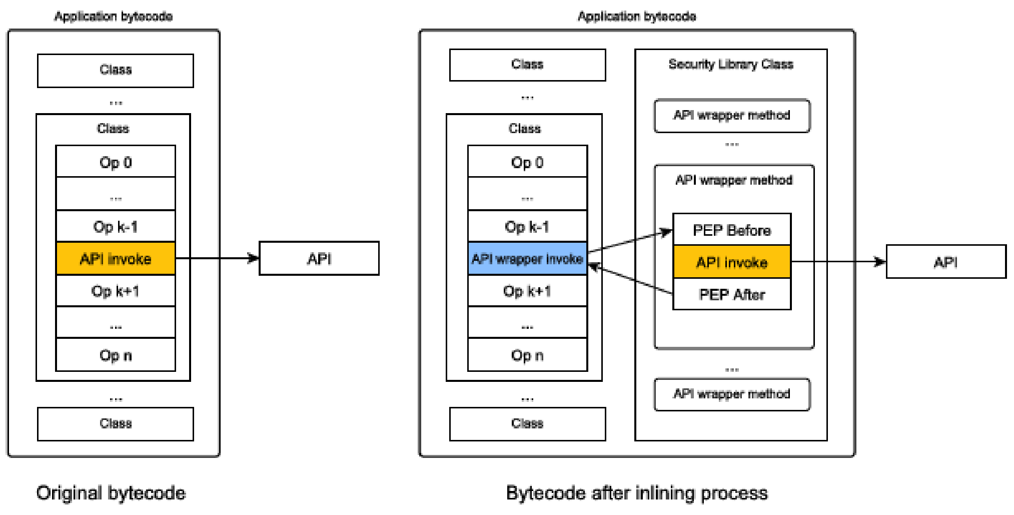


Figure 5.4: PEP Wrapping Injection.



5.2 SMM Monitoring Environment

The Secure Meta-Market implements PEP instrumentation directly on the mobile device and a centralized PDP for the policy evaluation. Intuitively, the SMM Client receives either verified applications or applications for instrumentation. In the first case, the application has been validated by the Secure Meta-

Listing 5.1: A fragment of instrumented code.

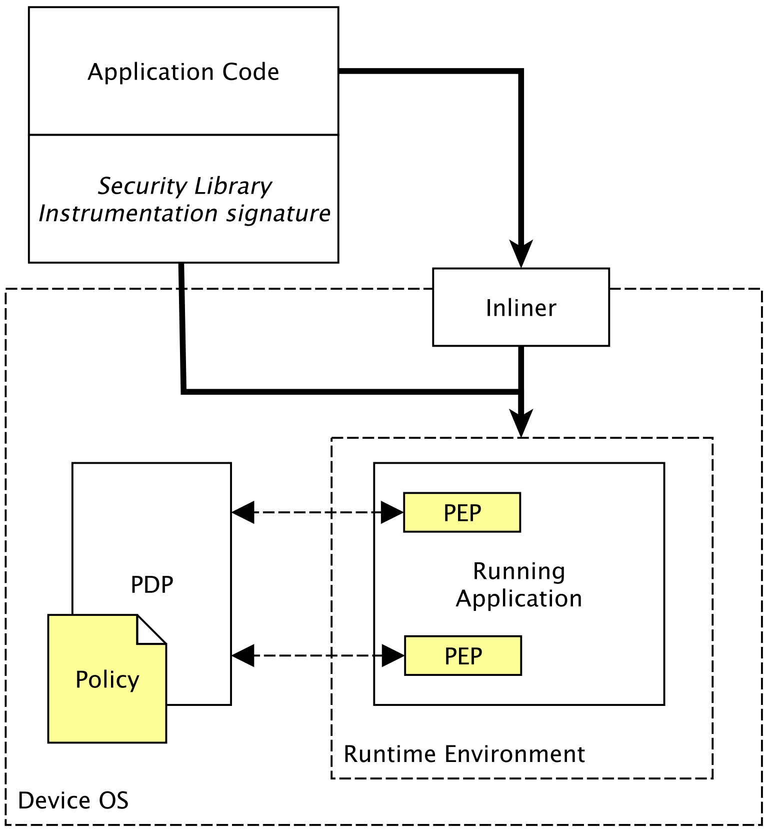
```
URL url;
PDP.checkBefore("java.net.URL.<init>",addr);
try {
  url = new URL(addr);
} catch(Exception e) {
  PDP.checkException("java.net.URL.<init>",e,addr);
  throw e;
}
PDP.checkAfter("java.net.URL.<init>",url,addr);
```

Market Server and, apart from installation, no further action is required. Otherwise, the instrumentation procedure attaches a Security Library and injects API Wrapping Methods of security-relevant operations in the application bytecode.

Figure 5.5 sketches the structure of the monitoring environment, its core components and their interactions. Roughly, the mobile device receives an application for instrumentation. The package contains application code together with an *instrumentation signature* and a *Security Library*. Both the signature and the library are computed by the Secure Meta-Market Server based on the application to instrument and the security policy of the company.

Instrumentation is carried out on the mobile device by an *inliner* component inside the Secure Meta-Market Client which injects the security checks and attaches the Security Library as discussed above. Eventually, the signature is attached and the application installed (solid arrows). Clearly, application instrumentation invalidates the software signature and, since most mobile OSes, e.g., Android and iOS, do not allow for the installation of unsigned contents, a new, valid one must be computed. Hence, the Secure Meta-Market generates a signature which only applies to the properly instrumented application. The signature guarantees the authenticity of the meta-market and prevents from illegal instrumentation. When the application is executed, it is loaded in a runtime environment which executes its instructions. Among them, PEP snippets cause communications with the PDP and, in case of policy violation, raise security exceptions (dashed arrows).

Figure 5.5: SMM Instrumentation and monitoring.



6

BYODroid, A Secure Meta-Market for Android

This chapter presents **BYODroid**, an implementation of the Secure Meta-Market that is meant to interact with the Android software development and distribution framework and adopts the techniques previously discussed. Such version allows the verification of applications against the security policy of the organization.

In the following I assume the code receiver to be an Android device and the application market to be Google Play.

6.1 BYODroid Architecture

BYODroid[†] is the first prototype implementation of a Secure Meta-Market for Android devices and supports the specification, verification and enforcement of fine-grained security policies over a network of federated devices. BYODroid has been presented in [8] and [7].

BYODroid consists of a server, namely the *BYODroid Server*, and an application client, i.e. the *BYODroid Client*, to be installed on Android-based mobile devices.

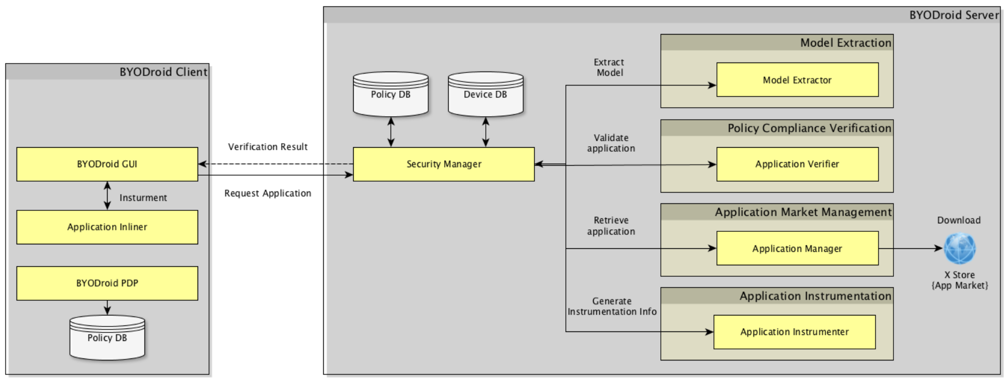
The BYODroid Server acts as a Secure Meta-Market: it retrieves original app packages from different app markets and verifies their compliance against a (formally specified) BYOD security policy provided by the organization. A user can browse applications from markets using the BYODroid Client. When the user asks to install an application, BYODroid Server retrieves the application

[†] `http://csec.it/software/byodroid/index.html`

from an app store and then analyzes its compliance w.r.t. the BYOD policy. If the application complies with the BYOD policy, it is directly installed on the device through the BYODroid Client. Otherwise, the user may decide to cancel the the installation or to ask BYODroid Server to instrument the application. The instrumentation step modifies the bytecode of the application so to make it compliant with the BYOD policy. The instrumented app is then safely installed on the mobile device. In this way all applications installed on the personal device by BYODroid are guaranteed to comply with the given BYOD policy.

The Architecture of BYODroid is depicted in Figure 6.1. In the following of this chapter, each of the implemented components is detailed.

Figure 6.1: BYODroid Architecture.



6.2 Security Manager

The Security Manager is the core management component. It supervises the interaction between the device (i.e. BYODroid Client) and the BYODroid Server. It gets from the device information useful for interacting with app-markets (i.e. userID, passwords, device id, carrier) that will be stored in the *Device DB*. Then, the Security Manager authenticates the subscriber to the app-markets and manages the verification flow, relying on the appropriate components in the BYODroid Server, as described in Chapter 3.

In the current deployment, the Security Manager allows to retrieve lists of available applications from app-markets as well as installing and removing applications from the device.

6.3 Model Extractor

Models are extracted directly from the application bytecode. The (Dalvik) bytecode is the intermediate language interpreted by the standard Android virtual machine, namely the Dalvik Virtual Machine (DVM). The application code is typically written in Java and then compiled in Dalvik bytecode. Actually, application packages consist of compressed Dalvik bytecode and resources, like audio files and pictures. Occasionally, application packages may also carry native code, i.e., libraries of machine-executable procedures. Although I do not present it below, the same approach can be applied for that kind of code (e.g., see [54]).

Basically, the model extraction process consists in building a CFG representing the behavior of the application. The CFG generation is a widely adopted technique [55] and several tools support the CFG extraction for bytecode. To this aim, BYODroid relies on the Androguard[*] framework, a state-of-the-art Android package analyzer written in Python.

Briefly, I implemented ad-hoc procedures for extracting and optimizing the CFG construction in terms of generated nodes and transitions.

The extraction process produces a distinct CFG for each method of the target code. These graphs are simplified, as detailed in Section 4.2, in order to reduce their size by removing irrelevant paths. Part of this activity consists in pruning those parts of a CFG which do not refer to any security-relevant method. Moreover, information about internal application invocations, i.e., how methods of the application invoke each other, is maintained. Such information is then used for the CFGs of the application components. The list of existing components is obtained by inspecting the implemented interfaces. For each of them, a general CFG is created by starting from interface methods, e.g., `startActivity` for activities. If the CFG of such methods contains invocations to other parts of the application, the corresponding sub-graphs are attached.

Each of these CFG is then converted to a corresponding LTS as explained in Section 4.2.

Naively, BYODroid could consider the parallel composition of all the models obtained in this way. Although such composition would safely represent the actual behavior of the applications, it would include many traces which never arise in actual executions. Indeed, in several cases the Android components do not execute concurrently. More commonly, when a component invokes another one, it is suspended or even terminated. According to the Android intent mechanism specification [56], component invocations usually transfer the execution

[*]https://code.google.com/p/androguard/

flow from the caller to the callee. This causes a significant reduction of the actual number of states obtained from the composition of two or more agents. Notice that, when required, parallel composition is still used, e.g., for Services and asynchronous tasks.

Further reductions concern the return flows. In many cases, upon termination components return the execution flow to their caller. For instance, this happens when the caller invokes the callee through `startActivityForResult(.)`. In this case, when the callee terminates, it returns some data and the caller resumes its execution. By applying this reasoning, several flows can be removed, i.e., those representing wrongly delivered responses. Although this step cannot be proved to preserve the soundness of the models, there are strong evidences, e.g., the platform documentation, supporting it.

Summing up, for each Android application the presented approach generates an inter-procedural model representing its security-relevant behavior. Application models carry transitions labeled with synchronization actions. These transitions represent the fact that two applications can interact through IPC, possibly transmitting data. Two or more models can be composed similarly to what has been done for the components of the same application. Intuitively, a composition of models represents the execution context in which the applications (the models have been extracted from) will operate at runtime. Instead, the composed model is context-insensitive w.r.t. the executing platform (i.e., the Android OS).

6.4 Application Verifier

Application validation against the security policy of the company is carried out through model checking, as presented in Chapter 4. Although several model checkers exist, I opted for Spin [57], a generic verification system for the Promela specification language that supports the design and verification of asynchronous process systems. Spin has its roots in the protocol verification systems based on on-the-fly reachability analysis as described in Section 4.4.

Spin has been successfully applied to the verification of a plethora of practical case studies in several different contexts (see [58] for a survey). In terms of reliability, Spin has been included in industrial verification activities as reported, for instance, in [59]. Moreover, Spin includes features for optimizing the use of CPU and memory. For instance, since version 5 Spin supports multicore computation [60].

The verification using Spin consists of three steps. First, Spin generates a C source file, called *pan.c*, implementing the verifier for the given agents and specification. Then, a C compiler is used to obtain the verification program (i.e. PAN) which is finally executed. The adopted compiler is gcc version 4.6.3.

6.4.1 Promela Encoding

The Promela specification language, offers a C-like syntax and allows to define processes to be run concurrently. Those processes can communicate via shared global variables or by message-passing via channels. Inside a process, constructs exist for non-deterministic choice, starting other processes and enforcing atomicity.

The mapping from LTSs denoting Android applications and ConSpec policies to Promela agents follows the procedure defined in [61] to append the agent to the original Promela specification obtained from the application model (see Section 4.2). The entire procedure is sketched below.

Briefly, the Promela agent for a given policy behaves like a message consumer. In practice, it actively waits for messages on a *system* channel. These messages represent the system accesses performed by the application agents. When a message is received, the policy agent reads the channel content, including invocation parameters, and process it against a sequence of clauses representing the ConSpec rules. If one is triggered, the policy state changes accordingly. Otherwise, the agent moves to a final, offending state.

Example 13. *Let consider a set of rules extracted from the White House security guidelines [9]. In Listing 6.1 I provide a ConSpec policy encoding them.*

The policy consists of a three-variables state and six rules. The first four rules define a policy saying that the user must not store data taken from the agency domain. In practice, when an application creates a URL pointing to the agency domain (rules 1 and 2), the URL object is saved in a variable (`agency_url`). Then, if the URL is used to open a stream connection (rule 3) a boolean flag (i.e., `connected`) is used to denote it. Finally, if the application tries to write on some local file after connecting (rule 4), the policy fails (since the only clause of the rule does not apply). The fifth rule enforces temporary files deletion upon application termination. Then, the last rule blocks outgoing bluetooth connections when an agency resource has been accessed (see above).

For the sake of presentation, here I minimized the number of rules. As a matter of fact, the actual policy includes further statements (e.g., ruling all the available URL connection methods and the existing device-to-device interfaces). These rules are analogous to those presented above and I omit them to improve the readability. For the experimental activity (see Chapter 8) I considered the full version of the policy.

Listing 6.1: ConSpec encoding of US government BYOD rules.

```
SECURITY STATE
  SESSION Str[11] agency_host = "agency.gov/";
  SESSION Obj agency_url = null;
  SESSION Bool connected = false;

/* (R1) User will not download business
        data on personal device         */
AFTER Obj url = java.net.URL.<init>(Str[64] spec)
  PERFORM
  (spec.contains(agency_host)) -> { agency_url := url; }
  ELSE -> { skip; }

AFTER Obj url = java.net.URL.<init>
    (Str[0] protocol, Str[64] host,
     Nat[0] port,     Str[0] file)
  PERFORM
  (host.contains(agency_host)) -> { agency_url := url; }
  ELSE -> { skip; }

BEFORE java.net.URL.openConnection()
  PERFORM
  (this.equals(agency_url)) -> { connected := true; }
  ELSE -> { skip; }

BEFORE java.io.FileOutputStream.write(int i) PERFORM
  (!connected) -> { skip; }

/* (R2) "When in Doubt, Delete it Out" */
AFTER Obj file = java.io.File.createTempFile()
  PERFORM
  (true) -> { file.deleteOnExit(); }

/* (R3) User will not transfer sensitive
        data to non-agency devices       */
BEFORE android.bluetooth.BluetoothSocket.getOutputStream()
  PERFORM
  (!connected) -> { skip; }
```

A fragment of the corresponding Promela encoding is given in Listing 6.2[*].

Listing 6.2: Promela encoding of US government BYOD rules.

```
proctype policy() {
/* Policy variables */
String agency_host = "agency.gov/";
Object agency_url = NULL;
bool connected = false;
/* Action parameters */
Object url, this;
String spec;
...
/* Channels */
mtype act;
...

/* Main loop */
endwait:
system?act ->
/* Rules implementation */
if
:: (act == java_net_URL__init__s_) ->
  /* Read parameters */
  ...
  if /* Guarded clauses implementation */
  :: (string_prefix(agency_host, spec)) ->
    { agency_url = url; goto endwait }
  :: else -> { goto endwait }
  fi;
:: (act == java_net_URL_openStream__) ->
  /* Read parameters */
  ...
  if /* Guarded clauses implementation */
  :: (this == agency_url) -> { connected = true;
                               goto endwait }
  :: else -> { goto endwait }
  fi;
:: /* Other rules */
```

[*]Full specifications can be found in Appendix A.2

```
  ...
else goto fail /* Default-deny */
fi;
/* Forces model checking failure */
fail: assert (false);
}
```

The policy agent has internal variables representing the ConSpec policy state. A second group of variables are used to store action parameters. Then, variable `act` *is used for receiving values over the* `system` *channel. The* `mtype` *domain is defined globally in the Promela specification and consists of an enumeration of possible (system or intent) actions labels. Briefly, the agent waits for actions over the* `system` *channel. When an action* `act` *arrives, the policy compares it with the expected ones in a conditional statement (*`if :: ... fi;`*). Then, it reads the action parameters (through a dedicated channel) and evaluates the existing clauses. If an illegal clause is true, the policy agent moves to label* `fail`*. If reached, the* `false` *assertion causes to model checker to report an error, i.e., a policy violation.* □

For the applications, BYODroid firstly extracts LTSs representing its components, i.e., activities, services, broadcast receivers and content providers. Then, it creates a Promela agent for each of them. Each Promela agent implements a finite state machine corresponding to the given LTS. Basically, the states of a Promela agent consist of finite sequences of three operations: system accesses, component invocations and conditional state jumps. System accesses consist of instructions sending a corresponding message on the *system* channel (see above), while the component invocations, i.e., intents sending, generate a message on the *broadcast* channel. Finally, the Promela instructions generated from a LTS are extended with an activation statement representing the specific kind of component it derives from. For instance, broadcast receivers wait for a certain message on the *broadcast* channel, while services start autonomously.

Example 14. *Assume the piece of code introduced in Example 8 is the main body, e.g., method* `onStart()`*, of an Activity component . The template of the corresponding Promela agent is given in Listing 6.3*[*].

Basically, it waits for an appropriate `intent` *signal before starting its computation. Then, it fires a sequence of actions (over channel* `system`*) equivalent to the trace described in Example 8 (original methods are reported in comments). According to the application code structure, certain parameters can be inferred*

[*]Full specifications can be found in Appendix A.2

Listing 6.3: Promela encoding of the CFG of Example 8.

```
proctype activity() {

/* Members */
Object url;
...

/* Channels */
mtype intent;
...

/* Activity is paused */
start:
broadcast?intent ->
  /* First block */
  ...
  /* new URL(s) */
  system!java_net_URL__init__s_;
  /* Send string s */
  ...

  /* url.openConnection() */
  system!java_net_URL_openConnection__;
  /* Send object url */
  ...

  /* file = File.createTempFile() */
  system!java_io_File_createTempFile__;
  /* Send object file */
  ...

  do /* while loop */
  :: /* fos.write(b) */
     system!java_io_FileOutputStream_write__b_;
     /* Send byte b */
     ...

  :: break /* loop exit */
  od;
  goto start; /* Activity can be re-executed */
}
```

(e.g., constants) while others are generated according to their type. Finally, note that here we omitted `atomic` *statements guaranteeing that the communications of system actions (and their parameters) cannot be interleaved by other actions fired by concurrent agents. The last line jumps back to the initial pause state so allowing the activity to run more than once.* □

All the agents denoting the existing components are combined with the policy agent (plus support agents, e.g., simulating the user activity) in a single Promela specification. The Promela specification obtained in this way serves as an input for the Spin model checker which verifies the reachability of the final state of the policy.

Notice that Byodroid implementation significantly simplifies the general version of [61]. As a matter of fact, BYODroid does not need to implement a generic framework for checking relations between agents. Under its assumptions, the verification procedure only needs to check a precise relation, i.e., strong simulation. Moreover, such procedure does not need to be parametric with respect to the observable channels/actions as they are statically defined as the elements of Δ (see Section 4.2).

6.5 Application Manager

The Application Manager is responsible for retrieving applications from the application markets. In the current implementation, BYODroid communicates with the Google Play service and the Samsung Store to get applications. Nevertheless, other application markets and sources exist and all of them can be integrated in our service by adding appropriate libraries to the Application Manager.

Moreover, the BYODroid Server uses these libraries to authenticate the subscriber to app-markets instead of the mobile device. By pretending to be the mobile device, the meta-market authenticates and obtains tokens from the app-market. Such tokens are then used to search and download available applications. App-market libraries are directly used by different components in the secure meta-market as discussed in the following.

6.6 Application Instrumenter

The Application Instrumenter is responsible for generating the Security Library and the instrumentation signature required for the BYODroid Client for the inlining procedure, as described in Section 5.2.

The *Security Library* is automatically compiled including the API Wrapping Methods of all security-relevant operations of the organization policy. The instrumentation signature instead is used to guarantee the authenticity of the BYODroid Server and to prevent illegal instrumentation by the client. Since the instrumentation procedure invalidates the application signature, the Application Instrumenter generates a signature which only applies to the properly instrumented application.

6.7 Byodroid Client Application

The BYODroid Client application provides access to the application installation/removal operations through a user interface which mostly resembles those of standard market client applications. Figure 7.3 depicts the client GUI in two different activities: applications browsing and application description. Briefly, the client is responsible for providing the user with details about the security of the available applications, e.g., green, verified labels identify secure applications. Also, it mediates the operations usually carried out by the application market clients, e.g., contents purchasing and refunding procedures[*].
When the user selects an application for installation, the client triggers the request and, consequently, the secure meta-market workflow implemented by the BYODroid Server. Eventually, the server returns a software package which, depending on the verification outcome, can require security instrumentation and monitoring.

6.7.1 Application instrumentation and monitoring

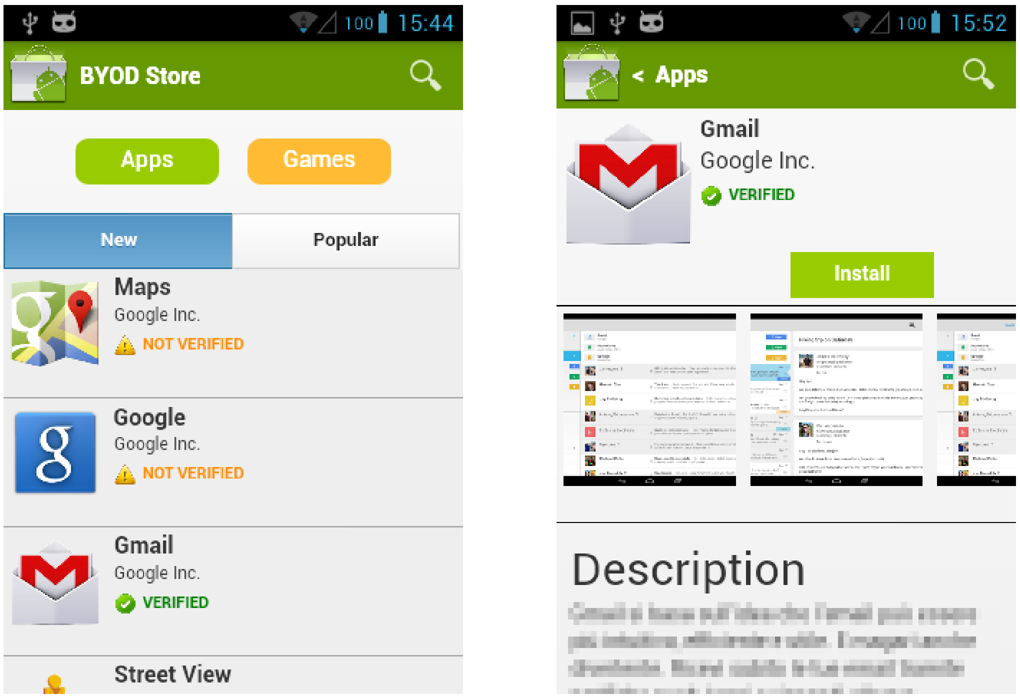
As discussed in Chapter 5, PEP inlining is a well known approach for enabling the security monitoring of mobile applications.

The current implementation relies on a modified version of *Redexer* [53] for instrumentation. Redexer is an OCaML implemented suite including several features for the assessment of some specific aspects of the Android application security, e.g., privilege exploitation analysis. Also, Redexer includes Dalvik bytecode parsing and rewriting facilities which I isolated and imported in the BYODroid prototype. Although not designed for running on Android platforms, it can be ported by means of the Java native interface[†]. Briefly, the instrumenta-

[*]Clearly, some of these tasks could need to be slightly reconsidered before integration in the meta-market. For the time being, the prototype does not support purchase refunding.

[†]See `https://sites.google.com/site/keigoattic/ocaml-on-android` for a technical explanation.

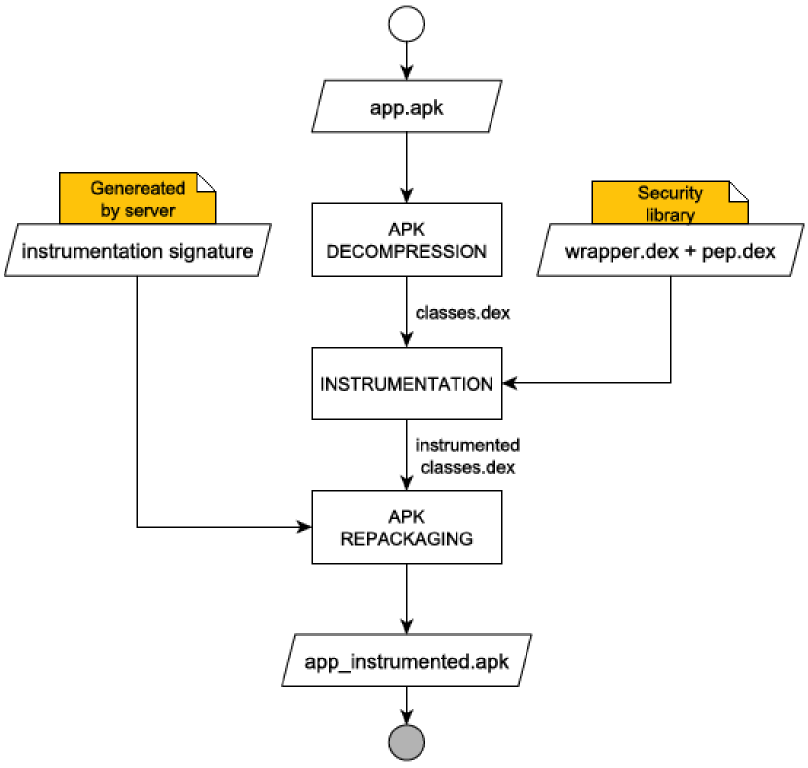
Figure 6.2: Meta-market client interface.



tion is carried out by defining a *visitor* which inspects the bytecode and injects PEP wrapping instructions of the *Security Library* provided by the Application Instrumenter. Eventually, the instrumented application is signed and repackaged for installation. The detailed instrumentation procedure carried out by the mobile device is depicted in Figure 6.3.

The PDP component, called B*YODroid Monitor*, is implemented as a background Android service running a policy interpreter. The service receives messages from the PEP instrumented in running applications. These messages are converted into events feeding the policy interpreter. Then, the interpreter simulates a corresponding step in the policy and returns a value representing the policy state, i.e., either *allow* or *deny*. An implementation of a ConSpec policy interpreter for mobile devices can be found in [52].

Figure 6.3: The Instrumentation procedure.



7

Use Case: BYODroid for NATO Agency

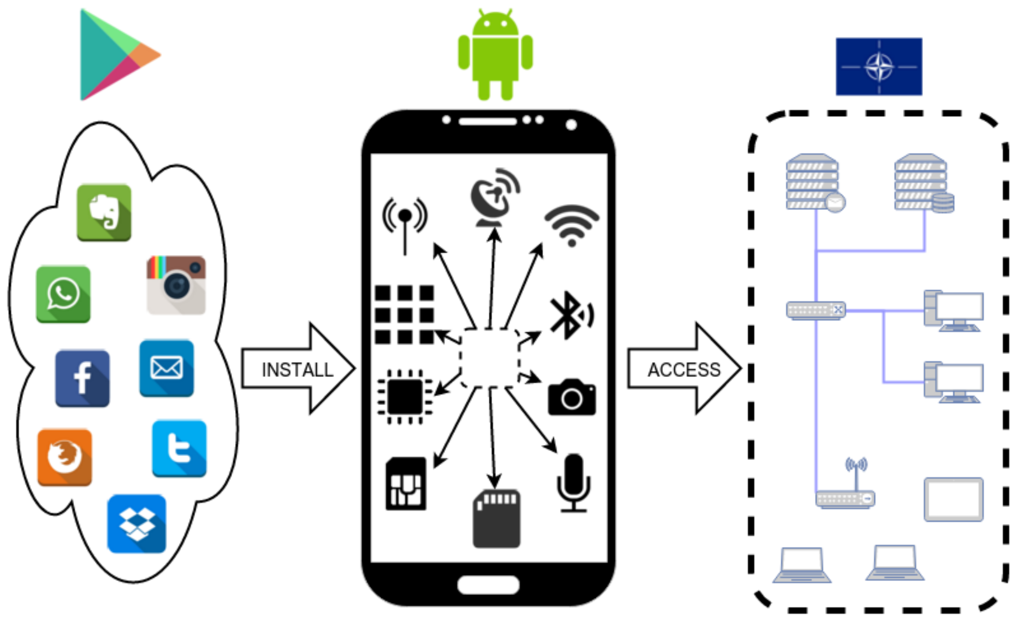
In this Chapter I present the implementation of BYODroid that has been prepared for the *NATO Communication and Information (NCI) Agency* and tested at the agency premises in The Hague, Netherlands. The contribution of this project, included in the *2015 Cyber Security Incubator pilot* project[†], was twofold.

The first activity was to evaluate the applicability of the BYODroid policy framework to the NCI Agency security rules for mobile devices. I started by processing the existing documentation and security guidelines in order to extract a BYOD policy. This process also involved active sessions with the NCI Agency security experts where the security rules were refined and precisely formalized. As a result, I could specify an unclassified test policy covering several aspects of the security-relevant operations that a mobile device can carry out within the NCI Agency infrastructure.

The second operation was to deploy the BYODroid support inside the NCI Agency ICT perimeter. This activity required us to extend BYODroid with extra components for the authentication and authorization of the users. More interestingly, I also added a role-based policy management system. In this way, the security administrator can specify different policies for different roles. Users are assigned to one or more roles and roles can inherit existing security policies from a parent, super-role. The actual policy that a user is subject to is obtained as the conjunction of the involved policies.

[†]https://www.ncia.nato.int/NewsRoom/Pages/150706-CyberSecuritySL-innovation-pilot.aspx

Figure 7.1: A schematic representation of the BYOD interoperability model.



7.1 Case study

The NATO Communication and Information Agency (NCI Agency) is a complex organization with more than 30 bases in the NATO countries and around 3100 employees. Due to the sensitivity the managed resources, the NCI Agency infrastructure is subject to strict rules and behavioral policies that the employees as well as the visitors must accept and respect. As a consequence, the usage of mobile devices, e.g., smartphones, is restricted and, in some cases, they are even prohibited.

Figure 7.1 schematically depicts the typical BYOD interoperability model (assuming the NCI Agency to be the organization adopting BYOD). From left to right, mobile applications reside on public app stores, e.g., Google Play. Applications developed by untrusted third parties are submitted to the store and made available to the users for installation. Once installed, the application executes on the host device and can access its resources and hardware, e.g., memory card, Bluetooth interface and other applications. Permitting to mobile device to enter the perimeter of the ICT infrastructure can allow the application to access and use sensitive data and resources. Since no security enforcement mechanism is in place, there is no actual guarantee that the application will not misuse these resources.

The access to the critical ICT infrastructure and resources of the NCI Agency is regulated by a rich documentation. Such documentation defines access and usage rules through natural language statements. Usually, rules are categorized according to the type of resources they involve. As a consequence, the BYOD policy tends to be distributed over multiple sources that need to be compared and composed. Below, I report an excerpt of the rules of interest and discuss them in more details.

Personal Area Network (PAN) security policy This class of policies is peculiar of mobile devices. As a matter of fact, smart objects, e.g., smart glasses and smart watches, comply with our definition of mobile device. Typically, they mount an operating system analogous to that running on a smartphone. Some applications exploit the capabilities of more than one of these devices to carry out specific tasks. For instance, a smart watch might collect hart-beat and pressure data and save it on a database stored by the user's phone.

Smart devices connect through via radio frequencies by means of the Bluetooth interface. The Bluetooth communications can cause information leakage. In fact, if such communications are not protected, nearby[*] receivers might intercept sensitive information.

Nevertheless, prohibiting the usage of the Bluetooth channel is possibly too restrictive. Indeed, several work related tasks may depend on it, e.g., Bluetooth printers and headphones. All in all, Bluetooth is accepted as far as it is appropriately secured.

The NCI Agency rule for the usage of Bluetooth connections can be summarized as follows.

PAN. Bluetooth devices shall establish encrypted communications.

Data management security policy Data is one of the most valuable assets to be protected within the NATO ICT infrastructure and several policies specifically target it.

Often documents are classified depending on their source and content. Access control mechanisms applied on data sources can prevent unauthorized persons form obtaining them. However, when an authorized user accesses a piece of data, the responsibility of protecting it passes to the user.

The usage obligations for classified documents are listed below.

DM1. Information stored on the device must respect policies depending on classification level.

[*]Notice that proximity is not an hard constraint as Bluetooth antennas can reach up to 100 meters and more.

DM2. Restricted, secret and confidential documents must never be stored on end devices.

DM3. Other documents must be eventually deleted from the device.

DM4. Mobile devices handling information belonging to the company must employ encryption to protect stored data.

DM5. Data encryption algorithms must meet company specifications.

DM6. Corporate data must not be stored in persistent memory, but rather in memory card.

Network access policy An application can attempt to create an FTP connection to pull and push documents from a remote host. Such behavior is dangerous as FTP is considered to provide an insufficient level of security. Having a running FTP client might permit to transfer sensitive documents in an insecure way and should be prevented.

The corresponding rule follows.

NET. File sharing services must be disabled.

7.2 Encoding the NCI Agency policy

This section presents a partial ConSpec encoding of the security requirements informally described in Section 7.1. For the sake of presentation some of the relevant Android/Java APIs are omitted. Nevertheless, the followed approach is general and can be replicated for them.

The basic operation for instantiating an unprotected Bluetooth connection is `createInsecureRfcommSocketToServiceRecord(UUID u)` being a member of class `android.bluetooth.BluetoothDevice`. Hence, the rule for preventing such operation is

```
BEFORE a.b.BD.createInsecureRSTR(Obj u)
PERFORM
 (false) -> { skip; }
```

where we abbreviated the full method name. The keyword `BEFORE` states that the rule must be evaluated before the actual execution of the operation, while `PERFORM` indicates the begin of the guarded actions list. Since the rule wants to permit the operation under no circumstances, I use the guard `(false)` which is trivially never satisfied.

Classified documents are retrieved from known web sources. The URL of the resource identifies its classification level. For instance, assume there exist four classification levels, i.e., unclassified, restricted, confidential and secret. The documents under this classification could be stored in `https://agency/unclass`, `~/restr`, `~/confid` and `~/secret`, respectively. A policy can track the access operations by declaring four Boolean variable indicating whether the corresponding URL has been accessed. Thus, the state of the variables can change when a new *java.net.URL* is created. Moreover, another Boolean variable can change when a connection is actually started, i.e., through `java.net.URLConnection.connect(URL url)` The ConSpec fragment for that is

```
SESSION Str[22] purl =
        "https://agency/unclass/";
SESSION Str[22] iurl =
        "https://agency/restr/";
SESSION Str[22] curl =
        "https://agency/confid/";
SESSION Str[22] surl =
        "https://agency/secret/";
SESSION Bool unclass = false;
SESSION Bool restr = false;
SESSION Bool confid = false;
SESSION Bool secret = false;
SESSION Bool connected = false;

BEFORE java.net.URL.<init>(Str[22] s)
PERFORM
 (s.startWith(purl)) -> {unclass = true;}
 (s.startWith(iurl)) -> {restr = true;}
 (s.startWith(curl)) -> {confid = true;}
 (s.startWith(surl)) -> {secret = true;}

BEFORE java.net.URLConnection.connect(Obj url)
PERFORM
 (true) -> {connected = true;}
```

where the meaning of the expressions is straightforward (only notice that `<init>` stands for the class constructor). Other rules can be added to implement the access control policy to be applied to the classification level. These rules must include in their guards the expressions for checking whether the corresponding url has been accessed.

For instance, rule DM2 excludes file saving for restricted, secret and confidential data. Moreover, (DM6) such files must be stored on the external memory, i.e., the SD card. The corresponding rule is

```
BEFORE java.io.File.<init>(Str[6] path)
PERFORM
 (path.startWith("sdcard") &&
   (!connected || !(restr || confid || secret))) -> {skip;}
```

which says that a file can be created only if they reside on the SD card and the application is not connected or, otherwise, has not accessed the restricted, confidential or secret data. For brevity I do not show the encoding of DM3, DM4 and DM5. Just notice that (DM3) deletion is guaranteed if the file is created through the `createTempFile` method, (DM4) data encryption requires the usage of the class `javax.crypto.Cipher` and (DM5) undesired encryption algorithms can be detected by checking the parameter of method `javax.crypto.Cipher.getInstance(String alg)`. The complete encoding of the policy can be found in Appendix A.3.

7.2.1 Policy compositionality and roles

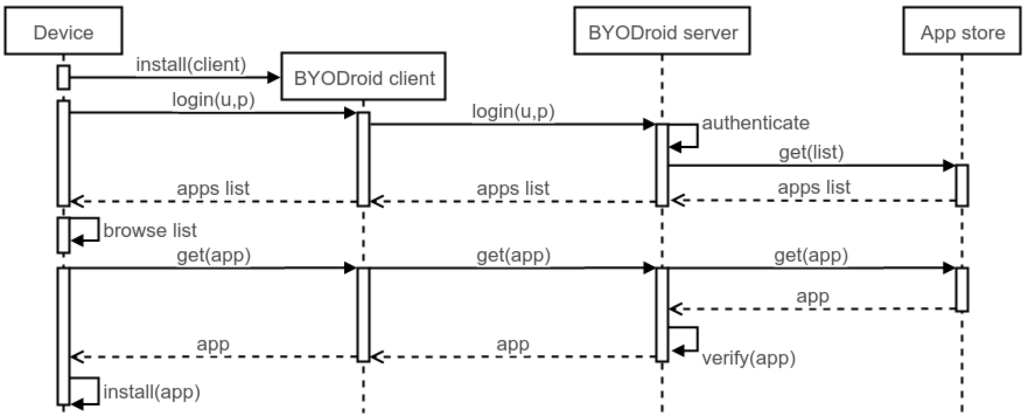
Above I presented the structure of the policy language and how to obtain a security specification form a natural language description. Nevertheless, under realistic assumptions, complex organizations have several policies, usually targeting a specific class of users, namely a *role*. In fact, role-based access control (RBAC) is often used by organizations with a precise role hierarchy.

The NCI Agency uses roles to categorize the capabilities and access privileges of employees and guests. New roles can be created and canceled depending on organizational changes. Users are assigned to one or more roles depending on their position and activity.

Clearly, a natural way to support RBAC is to assign a security policy to each role. Moreover, since a new role can be created from an existing one, e.g., *civilian employee* as a specialization of *employee*, policies should be inherited. To permit multiple roles and policy inheritance, the policy framework must provide provide compositionality guarantees. Once the set of policies to be applied is identified, compositionality guarantees that a unique policy can be synthesized that corresponds to the logical conjunction of the original ones.

ConSpec natively provides policy compositionality. Roughly, given two or more policies, their composition amounts to the sequence of their security state

Figure 7.2: BYODroid workflow for NCI Agency.



and rules. This operation might require to rename the policy variables (to avoid name conflicts). Also, since policy rules are evaluated according to their declaration order, policy composition can cause rule hiding. By assumption, when composing the policies of a role hierarchy we prioritize the children rules to the parent one.

7.3 BYODroid policy enforcement

In this section I present the BYOD security infrastructure implementing the policy framework described above. The core components are inherited from BYODroid. However, they have been enhanced to cope with the specific security requirements of the NCI Agency environment.

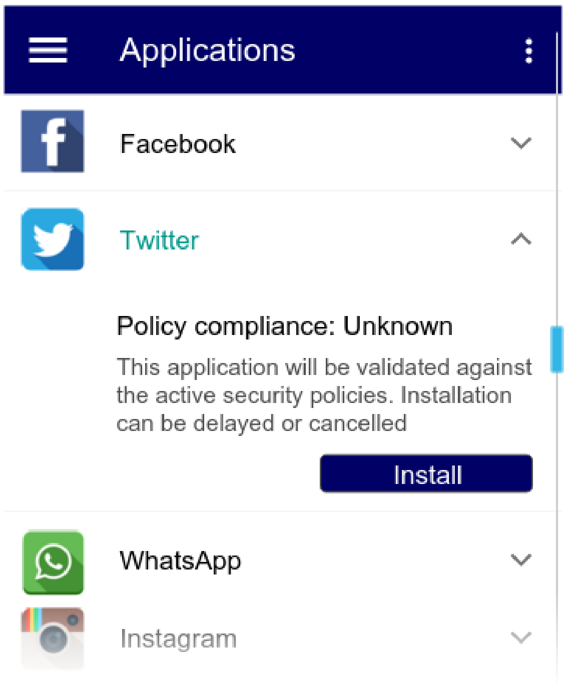
7.3.1 Overall architecture

The BYODroid architecture for the NCI Agency consists of three components, i.e., the administration portal, the BYODroid Server and the BYODroid Client. Both the administration portal and the BYODroid Server are part of the NCI Agency infrastructure. New users access the web service and submit their request to bring their own device. This step includes an authentication and authorization step based on user identity/credentials. Finally, the user receives and installs the BYODroid Server. The sequence diagram of Figure 7.2 shows the interactions between the BYODroid Client and Server.

When launching the BYODroid Client, the user is requested to authenticate (operation login(u,p)). Then, username and password are sent to the BYODroid

Server. All the network communications (i.e., client-server and server-store) pass through HTTPS connections. If the authentication succeeds, the server retrieves and returns the list of available applications (apps list), which is eventually displayed to the user (see Figure 7.3).

Figure 7.3: BYODroid Client interface for NCI Agency.



Policy compliance: Unknown

This application will be validated against the active security policies. Installation can be delayed or cancelled

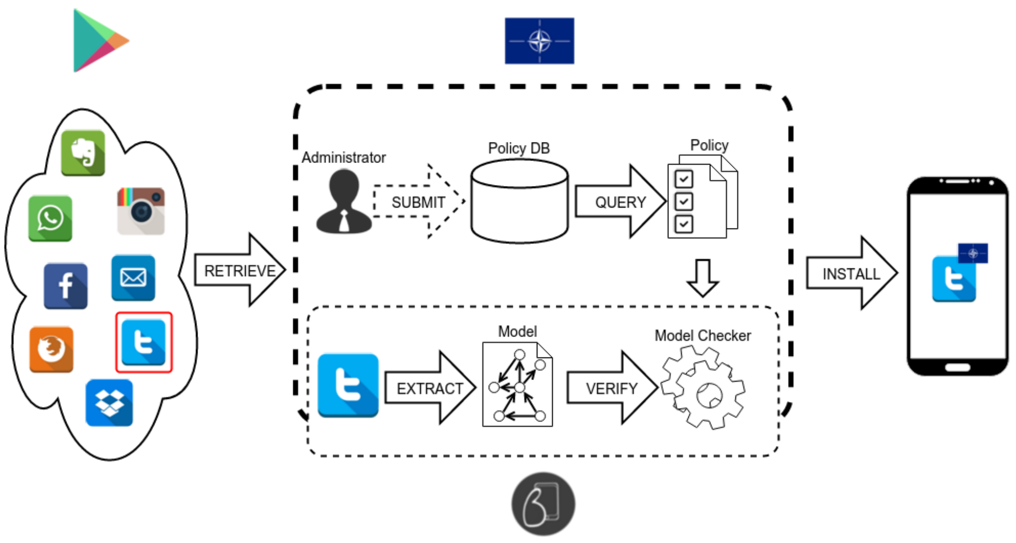
When a new application is requested (get(app)) for installation, the client submits its query to the server. Thus, the server retrieves the application package from the App Store and executes the verification procedure. If the verification fails, the installation in canceled. Otherwise, the application is returned to the BYODroid Client which installs it.

7.3.2 Development

The B*YODroid Client* provides access to the application installation/removal operation through a user interface which resembles standard market client applications. In details, the BYODroid Client implementation is based on F-Droid*, an open-source application market for Android. The BYODroid Client requires the login credential provided by the IT Administrator of the organization and allows

*https://f-droid.org/

Figure 7.4: BYODroid interoperability model.



registered devices to assess the organization market hosted by the BYODroid Server. The login procedure relies on the Apache Shiro Security Framework[*], which allows authentication and session management.

The *administration portal* allows the IT Administrator to manage the accounts and policies for the organization. Indeed, the portal allows for the creation of users and the management of roles and policies maintaining the hierarchical structure discussed in Section 7.2.1. Due to security requirements, the access to the administration portal is restricted to specific IP addresses inside the organization network.

The B*YODroid Server*, representing the core of the infrastructure, is responsible for performing the verification of compliance of requested applications w.r.t. the organization policy. The implementation of the verification procedure has been previously reported in [29] and [14] and widely described in Chapter 6.

The overall BYODroid interoperability model is is schematically depicted in Figure 7.4. The main difference with the original scheme of Figure 7.1 is the active role of the BYOD infrastructure which supports the definition and enforcement of the security policies.

[*]http://shiro.apache.org/

8

Experimental Results

This chapter presents a thorough experimental evaluation aimed at assessing both the effectiveness and the feasibility of the proposed approach. Previous works like [24] and [8] reported preliminary experimental results confirming the feasibility of some components in isolation (e.g., model extraction and model checking). Here the whole verification chain is considered, namely model extraction, model checking and (eventually) instrumentation.

8.1 Experimental setup

To evaluate the performances and the effectiveness, the prototype described in Chapter 6, is tested against a large number of applications and a real-world BYOD policy. Below, each aspect of the experimental setup is discussed.

Security Policy Each app was tested against a security policy drawn by the BYOD security guidelines of the White House [9]. The policy consists of few rules stating how the mobile devices should access and manage the resources of the organization. Albeit simple, it is a significant excerpt of a real-world BYOD security policy. Informally, it can be summarized as follows:

(R1) Never download business data on the mobile device.

(R2) Always delete locally stored sensitive data (e.g., emails).

(R3) Never transfer data to non-agency devices.

The policy can be seen as restricting the usage of some security-relevant APIs[†]

[†]For brevity only few of them are considered here.

Listing 8.1: ConSpec encoding of US government BYOD rules.

```
SECURITY STATE
  SESSION Str agency_host = "agency.gov/";
  SESSION Obj agency_url = null;
  SESSION Bool connected = false;
/*(R1) Never download business data on the mobile device*/
AFTER Obj url = java.net.URL.<init>(Str addr) PERFORM
  (addr.contains(agency_host)) -> { agency_url := url; }
  ELSE -> { skip; }
BEFORE java.net.URL.openConnection() PERFORM
  (this.equals(agency_url)) -> { connected := true; }
  ELSE -> { skip; }
BEFORE java.io.FileOutputStream.write(Nat i) PERFORM
  (!connected) -> { skip; }
/* (R2) Always delete locally stored sensitive data */
AFTER Obj file = java.io.File.createTempFile() PERFORM
  (true) -> { file.deleteOnExit(); }
/* (R3) Never transfer data to non-agency devices */
BEFORE android.bluetooth.BluetoothSocket.getOutputStream()
  PERFORM
  (!connected) -> { skip; }
```

and can be formally recast to the ConSpec specification in Listing 8.1.[*]

For instance (R1), data download consists of network requests, e.g., HTTP-GET, for some critical URL, e.g., `https://agency.gov/`, followed by a file system writing action. The involved Java APIs are `java.net.URL.<init>(String s)`, which creates an URL object from a string, `java.net.URL.openConnection()`, which sets up a connection to the given URL, and `java.io.FileOutputStream.write(...)`, which writes data in a given file object. The sequence of these actions may cause a policy violation. To represent the security state, which changes as a consequence of the observed actions, the policy exploits *state variables*. The syntax of ConSpec statements and expressions resembles that of most imperative programming languages. The policy uses a three-variable state (`agency_host`, `agency_url`, and `connected`). Variables are labeled as **`SESSION`** as they only refer to a single execution of a single program (as opposed to **`MULTISESSION`** and **`GLOBAL`**). Policy rules refer to events, i.e., guarded API invocations, and can be activated **`BEFORE`** or **`AFTER`** the event is observed. When this happens, the clauses below the rule are evaluated and, if their guards are satisfied, executed. For instance, the first rule says

[*]The syntax has been slightly simplified for the sake of readability.

that, after observing a `java.net.URL.<init>(String addr)` operation, the policy evaluates whether the url address contains the string `"agency.gov/"`. If it is the case, the returned `URL` object is stored for checking its usage in future events. Otherwise (**`ELSE`**) nothing happens (**`skip`**). Similarly, the second rule states that, before permitting a `java.net.URL.openConnection()` invocation, the policy compares the current url **`this`** with that stored in variable `agency_url`, e.g., as a result of the application of the previous rule. If the two objects are equal, the policy assigns **`true`** to `connected`. Otherwise, the state is not changed. Finally, notice that the third rule has a single guard for the case `!connected`. This implies that the policy is violated, since no transition is allowed, whenever the rule is triggered in a state such that `connected = ` **`true`**.

Application test-set To assess the scalability of the proposed approach, I used Android applications drawn from the Google Play Store. The sheer size of the store, which contains more than 1 million applications [62], does not permit an exhaustive analysis of all possible cases. A smaller, but still interesting set is represented by the "top free chart" applications, which contains the most popular free applications in Google Play*. This includes over 800 applications from a number of heterogeneous categories, e.g., entertainment, productivity, and games.

Evaluation criteria The experiments have been run using a Dell Optiplex 9010 server with an Intel Core i7 3.40 GHz and 16 GB of RAM, equipped with Java JDK version 1.7. The BYODroid Client has been installed on a Samsung Galaxy S3 equipped with a Quad-core 1.4 GHz Cortex-A9 and 1 GB of RAM. The time out for PAN verification was set to 5 minutes.

The main goal of the experiments is to highlight the feasibility of the verification process. In particular, two aspects are crucial:

Model Size. Model checkers using an explicit representation of the state space (as Spin does) can rapidly occupy the available memory. Complex applications mean larger models and increased memory usage.

Execution time. Model checking requires a significant time span for validating an application. Also, other tools involved in the verification chain may cause further delay. Understanding how the tools contribute to the whole

*`https://play.google.com/store/apps/collection/topselling_free` (accessed on September 30, 2014)

execution time can help in identifying bottlenecks and possible optimizations.

8.2 Experimental results

I measure the time needed for each operation carried out by the BYODroid Server and Client. Table 8.1 and Table 8.2 show an excerpt of the experimental results[*]. A detailed explanation of each experimental phase is provided below.

Model extraction At first, the BYODroid Server extracts the model of each application, using the *Model Extractor* component. The CFG is generated parametrically w.r.t. the system action expressed in the security policy and all inter-application communications. Then, the server performs an optimization step consisting in simplifying the CFG by pruning unreachable nodes. Column T_{ext} reports the time needed (in seconds) for the generation of the final CFG while the size of the model is given in terms of number of nodes (column N) and edges (column E).

Figure 8.1 describes the global results of the extraction procedure of the applications dataset. The x-axis shows the T_{ext} (clustered in ranges) required for the model extraction, while the y-axis counts the number of applications processed within that range. It is worth noticing that the model extraction is completed within 60 seconds in the 70,56% of the applications (i.e. 609 out of 862 applications).

Thanks to the *Model Extractor* optimization techniques, the size of the model, represented as the size of the CFG (i.e. number of nodes and edges of the graph), can be limited thus reducing the complexity of the verification process. Figure 8.2 shows that, regardless of the size of the applications, model sizes, computed as the sum of nodes N and edges E of the CFG, have nearly a logarithmic grow.

Promela encoding and model checking As explained in Section 6.4, the use of the Spin model checker passes through two consecutive phases, i.e., Promela encoding and verification. The duration of the encoding phase for each application is listed under T_{enc} in Table 8.1. The duration of the model checking verification step is reported in column T_{mc}. Moreover, I provide in column *Valid*

[*]The complete results can be accessed online at `http://www.ai-lab.it/byodroid/experiments.html`.

Figure 8.1: BYODroid Model Extraction.

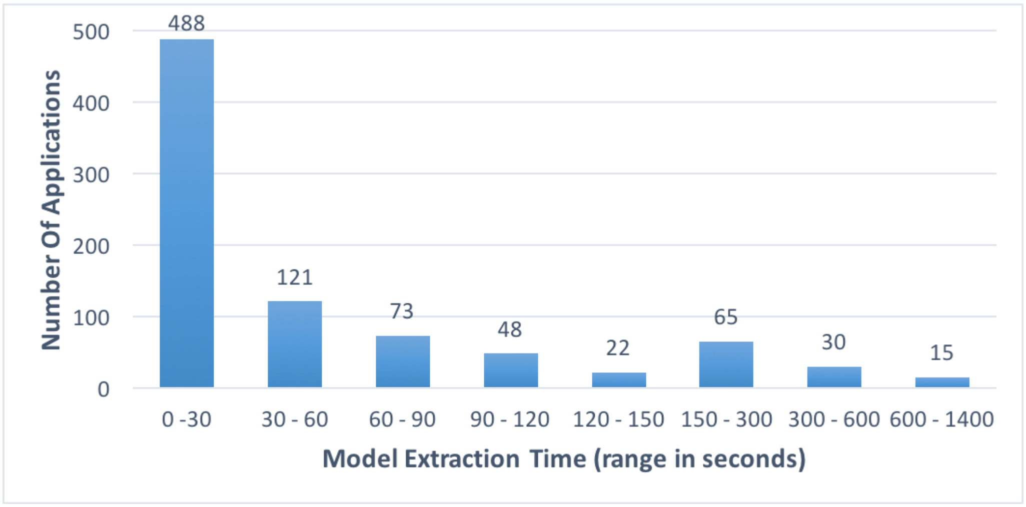
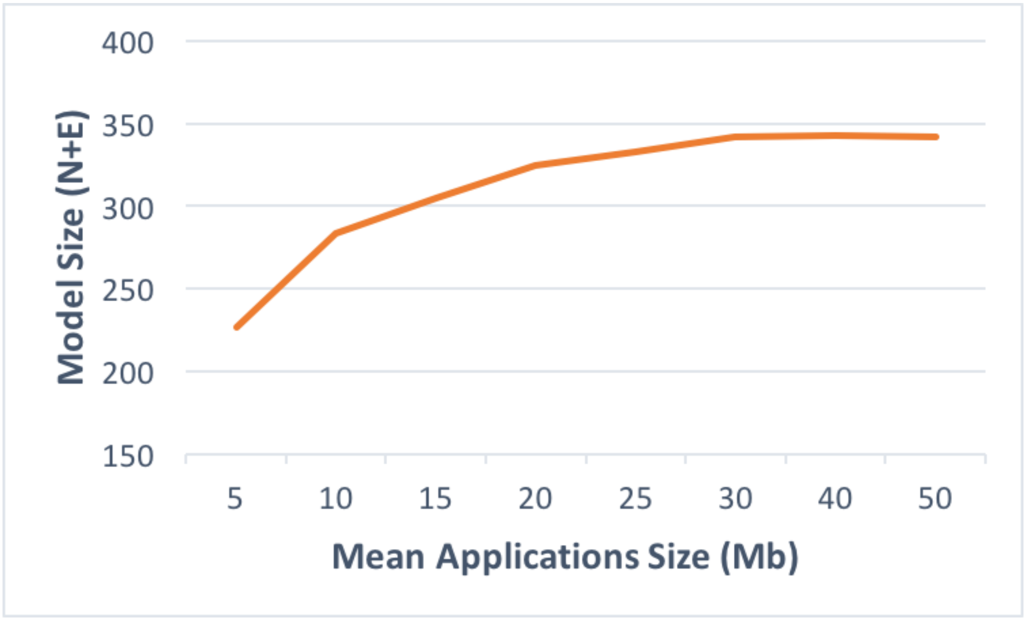


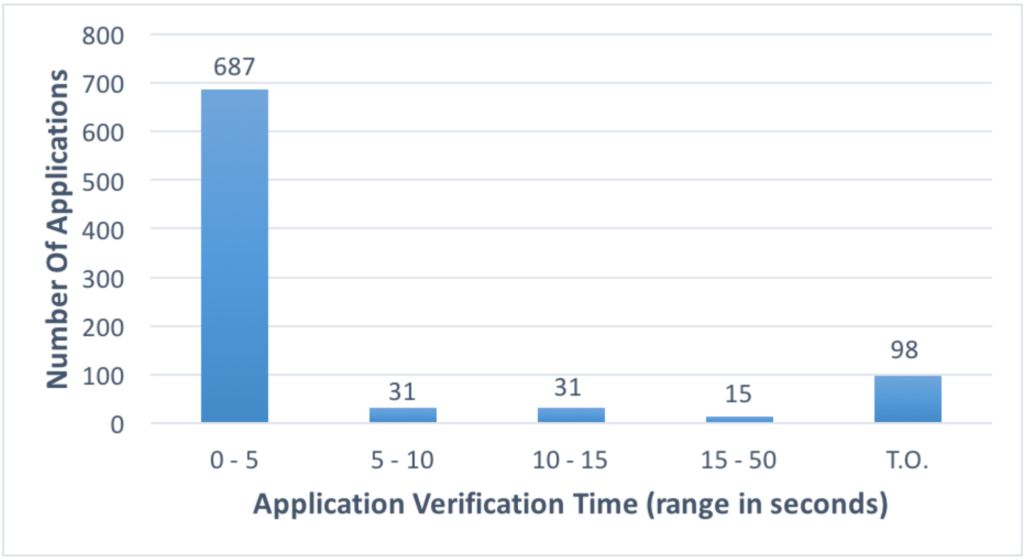
Figure 8.2: Applications model size.



the outcome of the model checker. I write YES to denote that the application successfully passed the verification, NO for the applications failing the verification, and Time Out (T.O.) for analysis that reached the time threshold.

Figure 8.3 provides an overview on the time required by this activity, depicting the number of applications verified with a specific range of time (expressed

Figure 8.3: BYODroid Application Verification.

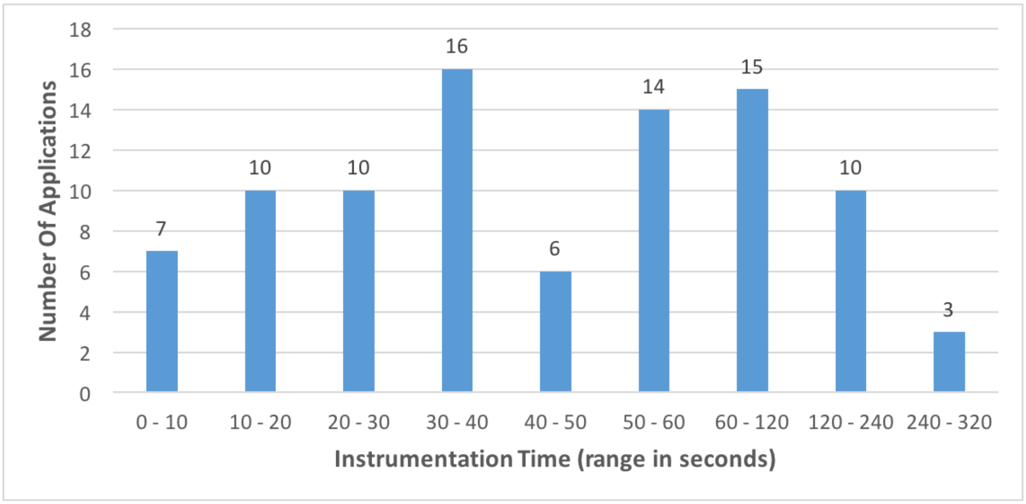


in seconds). I also report the range $T.O.$ for those that reaches the verification threshold. The results show that the verification procedure terminates for the 89,55% (i.e. 772 out of 862 applications) of applications within the system time-out.

Application instrumentation Application instrumentation is performed when the model checking fails in validating the application. In details, only the 10,67% of the applications needs the instrumentation phase (i.e. 92 out of 862), thus proving that the verification phase is effective in nearly the 90% of the dataset. The instrumentation execution times are reported under T_{ins} in Table 8.2. Moreover, since instrumentation may enlarge the original dimension of the software packages, we also report the package size change (column G), in percentage.

Figure 8.4 depicts the time required by this activity, showing the number of applications instrumented by the BYODroid Client within a specific range of time (expressed in seconds). Although this activity is performed on a commercial mobile device with limited resources (i.e. a Samsung Galaxy S3), it is worth noticing that 64 out of 92 applications are instrumented by the *Application Inliner* within 60 seconds.

Figure 8.4: BYODroid Instrumentation.



8.3 Discussion

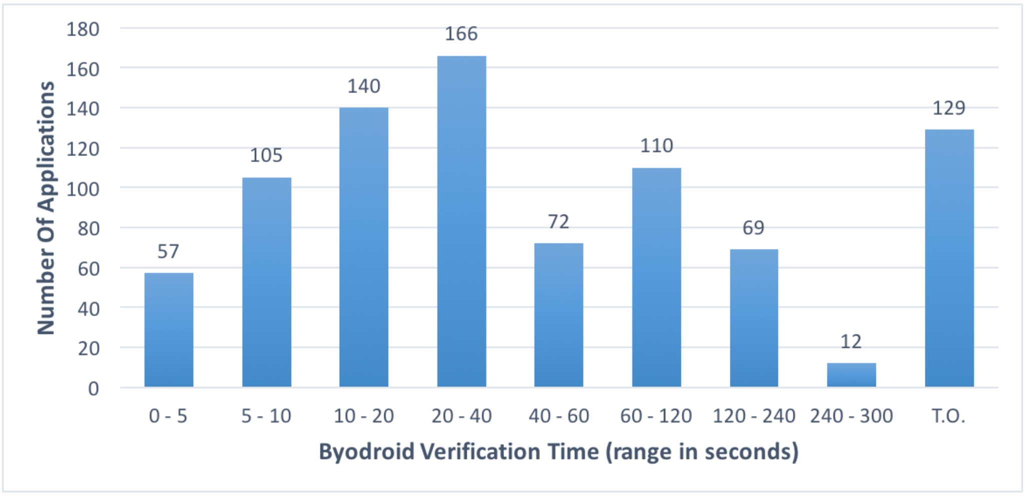
The global values in Table 8.1 and Table 8.2 summarize the overall statistics of our experiments. Although preliminary, the experimental results are promising. As a matter of fact, BYODroid validates more than 89% (column Y/N/TO) of applications. The remaining 11% (i.e. 92 applications), being rejected or leading to a time out, was successfully instrumented. The instrumentation process took an average time of 72 seconds (column μT_{ins}) and slightly enlarges the application packages by less than 0,1% on average (column μG).

Moreover, it is worth noticing that within 4 minutes, as depicted in Figure 8.5, more than 83% of the applications (i.e. 764 out of 862) are successfully verified and secure by the BYODroid architecture.

Summing up, the experimental results look promising and show applicability of the BYODroid architecture to a real scenario. Comparing the structure of my prototype and the outcome of the experiments with the purposes of my framework, I can make the following observations.

1. Most of the security analysis procedure is transparent to the user. Nevertheless, users can notice a delay at install-time. Although, I can expect that they are correctly informed about this process, real implementations should reduce as much as possible this overhead. Since my prototype admits further optimizations, I am optimistic on the possibility of reducing it.

Figure 8.5: BYODroid Total Verification Time.



2. The verification process is fully automatic. Both users and organizations are not required to take any critical decision.

3. Application collusion is disabled. Security policies are evaluated against models in which the notion of application is abstracted away. Model composition is obtained by simply juxtaposing the corresponding Promela encodings. Clearly, model checking performances can suffer from the model size increment. In this respect, Partial Model Checking (see Conclusion for more details) may provide important simplifications that deserve to be evaluated through further experiments.

4. Standard users cannot compromise the safety of their device with respect to the BYOD policy. Nevertheless, malicious (clumsy) users could attempt to tamper the device intentionally (by mistake). Hence, monitoring facilities must be considered for preventing dangerous actions like the manual installation of application packages.

Table 8.1: BYODroid Server computation times and outcomes (excerpt).

Application	Size(Mb)	T_{ext}(s)	N	E	T_{enc}(s)	T_{mc}(s)	Valid
Adobe Connect Mobile	11,71	1,5	11	63	0,1	2,1	YES
Adobe Reader	6,63	11,5	36	157	0,6	2,1	YES
AndrOpen office	49,59	20,2	39	164	0,4	2,3	YES
Angry Birds	33,9	106,7	93	367	1,7	300	T.O.
Angry Birds Rio	32,61	106,6	93	367	1,7	300	T.O.
Angry Birds Seasons	42,27	112,3	94	373	1,7	300	T.O.
Candy Crush Saga	38,45	8,3	41	169	21	3,1	YES
Dropbox	5,59	32	33	168	11,1	4,9	YES
Extreme racing	8,02	62,5	112	403	2,1	2,4	YES
Facebook	15	30,2	25	112	0,2	2,3	YES
FB Messenger	12	314,3	41	180	12	4,3	YES
Fruit Ninja free	18,34	33,8	78	287	1,1	300	T.O.
Gems Journey	11,19	53,6	51	201	0,8	300	T.O.
Gmail	3,57	23	36	157	0,4	2,1	YES
Google Chrome	24,6	36,3	54	230	10,1	2,3	YES
Google Play Music	7,53	68,1	64	276	1	2,1	YES
Google Street View	0,25	1,8	13	57	0,1	2,1	YES
Instagram	14,87	93,1	60	244	4	3,8	YES
LinkedIn	6,56	65,1	69	292	4,2	2,1	YES
Microsoft Lync 2010	3,61	3,5	8	41	0,1	2,1	YES
Microsoft Remote Desktop	4,50	16,622	15	64	0,1	2,1	YES
Mozilla Firefox	23,13	36,7	60	226	1,1	2,2	YES
OpenDocument Reader	1,61	45,8	44	188	0,6	3,0	YES
PocketCloud Remote	11,25	73,2	198	660	4,8	2,4	YES
Tiny Flashlight	1,28	16,4	61	223	0,8	3,2	YES
T.N.T. Italy	5,54	77,7	54	209	0,7	2,9	YES
TripAdvisor	6,31	24,1	74	269	1,1	2,1	YES
TuneIn Radio	6,58	154,4	174	649	8,3	13,3	NO
Twitter	5,18	42,7	62	256	10,7	2,3	YES
SMS backup & restore	1,33	35,5	74	298	1,3	2,9	YES
Skype	14,73	56	34	147	2,8	5,5	YES
Splashtop 2 Remote Desktop	17,78	58,46	90	357	1,7	2,2	YES
Spotify	3,65	8,1	18	79	2,3	2,1	YES
WhatsApp	9,75	369,4	223	715	8,8	2,1	YES

Global values							
# Applications	μSize	μT_{ext}	μN	μE	μT_{enc}	μT_{mc}	Y/N/TO
862	11,53	69	71	271	4,8	114,9	89/0,2/10,8

Table 8.2: BYODroid Client computation times and outcomes (excerpt).

Application	T_{ins}(s)	G(%)
Adobe Connect Mobile	-	-
Adobe Reader	-	-
AndrOpen office	-	-
Angry Birds	38,7	0.14
Angry Birds Rio	38,2	0.14
Angry Birds Seasons	43,2	0.11
Candy Crush Saga	-	-
Dropbox	-	-
Extreme racing	-	-
Facebook	-	-
FB Messenger	-	-
Fruit Ninja free	29	<0,01
Gems Journey	33	0,23
Gmail	-	-
Google Chrome	-	-
Google Play Music	-	-
Google Street View	-	-
Instagram	-	-
LinkedIn	-	-
Microsoft Lync 2010	-	-
Microsoft Remote D.	-	-
Mozilla Firefox	-	-
OpenDocument Reader	-	-
PocketCloud Remote	-	-
Tiny Flashlight	-	-
T.N.T. Italy	-	-
TripAdvisor	-	-
TuneIn Radio	66	0,71
Twitter	-	-
SMS backup & restore	-	-
Skype	-	-
Splashtop 2 Remote D.	-	-
Spotify	-	-
WhatsApp	-	-
Global values		
# Instrumented Applications	μT_{ins}	μ
92	72,6	0,08

9

Related Works

Since the BYOD paradigm has been proposed only recently, the literature addressing the associated security concerns is very limited at present. Nevertheless, several works propose solutions for applications security analysis which might be considered for implementing the BYOD security frameworks.

Android application security The Android permission system is the basic element for defining application privileges and enforce them at runtime. However, since its first appearance it received many criticisms, e.g., see [63], and many authors, e.g., see [64, 65, 66], proposed modifications and improvements, mainly focused on enhancing the security for user's activities (e.g. E-commerce [67]) as well as reducing the exploitability of device vulnerabilities (e.g. [68]). Still, permissions are not sufficient to define fine-grained policies as those needed for BYOD systems.

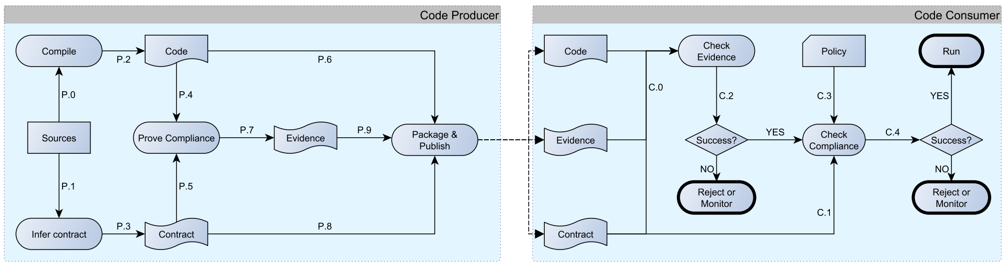
Several authors presented approaches that extend or redefine the Android security framework with a particular attention to data flow analysis. For instance, FlowDroid [69] analyses data-flows over Android applications and, its extensions [70, 71] also deal with multiple Android applications interacting through IPC channels. A similar approach explicitly dealing with enterprise data is put forward in [72]. Instead, [73] checks data flows through applications against security specifications extracted from the manifest, while [74] applies a reachability analysis to discover whether applications leak sensitive information through the network. Runtime enforcement solutions include TaintDroid [75], i.e., a system-wide dynamic taint tracking system capable to simultaneously track multiple sources of sensitive data, and Kynoid [76], i.e., a monitoring framework for user-defined security policies for data-items. Although relevant, data flow analysis does not cover usage control aspects which are central in BYOD environments. Moreover, these approaches mostly target the validation of single applications while we are also interested in the analysis of applications composition. Recently, a security enhanced version of Android, namely SEAndroid [77], has been released for defining system-wide mandatory access control (MAC) policies. Furthermore, in [78] it has been extended to support security policies specified through timed automata. Although SEAndroid permits to define global MAC policies over the OS resources of the device, it lacks the sufficient level of abstraction to deal with BYOD environments.

Policy enforcement relying on formal methods Formal security analysis techniques require a rigorous, mathematical description of the security policy and the behavior to be validated against it. For instance, a common practice consists of a runtime monitor which compares the execution trace of an application against the security policy. Security frameworks based on this approach include [39] for Java Standard Edition, [79] for .NET and [80] for Java Micro Edition. Runtime monitoring can effectively control that the execution of programs comply with a formally defined policy. Policy languages provided with a formal semantics include temporal logics, e.g., LTL [81], and finite state machines, e.g., security automata [82]. Still, runtime enforcement must be carried out by the execution environment, i.e., the mobile device, at the cost of undesirable computational overhead and energy consumption.

On the other hand, static enforcement techniques typically rely on a mathematical structure describing the *behavior* of the application, namely a *model*. Models abstract away details that are considered to be irrelevant for the security analysis. Usually, very abstract models are more compact and easier to handle/-analyze while detailed ones lead to longer computations and, in certain cases, can cause intractability. A common practice requires to define a *type and effect system* [25] to infer a model from a piece of code. In [24, 7] I followed this approach on a minimal core language [20]. Although simplified, such framework included most of the relevant aspects of the Android IPC and components. The type and effect system infers a *history expression* [30] from each Android component. Also, in [24] I showed that the type and effect system generates *safe* history expressions, i.e., they over-approximate the actual behavior of the application.

Although feasible, redefining the type system for the entire Java language (and also for its bytecode) can be a cumbersome process (due to its reach and redundant syntax) and many proposals only address fragments of the full Java language. In [33] the authors opted for generating history expressions out of *control*

Figure 9.1: The Security-by-Contract workflow including code producer and consumer.



flow graphs (CFG). A proof of the soundness of this approach is not available. Nevertheless, several authors worked on the formalization of the Java semantics (e.g., see [20, 83, 84]) which can be used for such a proof. Remarkably, the CFG correctness has been already obtained for Java 1.4 [85].

Building a CFG from a Java program is a rather common operation and some tools exist. For instance, ConFlEx [31] translates bytecode programs into BIR, an intermediate, stack-less language, and creates a CFG without loosing critical information. Similarly, Soot [32] relies on Jimple (Java sIMPLE), a stack-less, three address language, to carry out a similar procedure. Also, the generation of a CFG is a rather efficient process as discussed in [86].

Mobile Code Security Frameworks. Four main approaches to mobile code security can be broadly identified in literature: *sandboxes* [87] limit the instructions available for use, *code signing* [88] ensures that code originates from a trusted source, *proof-carrying code* (PCC) [89] carries explicit proof of its safety, and model-carrying code (MCC) [90] carries security-relevant behavior of the producer mobile code. Although relevant, those approaches are narrowed to specifich part of the application life cycle. To solve such issues, Security-by-Contract has been prosed in [91].

The *Security-by-Contract* (Security-by-Contract) is an architectural paradigm addressing the problem of including untrusted, mobile components in a security-critical environment, e.g., web services [91] and mobile applications [79]. Briefly, it composes several formal techniques, e.g., *model checking* [92] and *proof-carrying code*, addressing specific aspects of the security assessment in a unique framework. At the best of our knowledge, Security-by-Contract is the only proposal for an integrated, formal security framework covering each state of the mobile application life-cycle, from development to execution.

Figure 9.1 depicts the Security-by-Contract workflow. Two actors participate in the Security-by-Contract workflow, i.e. an untrusted code producer and a code consumer. The code consumer wants a security policy to be respected by the code provided by the producer. From left to right, the code producer compiles application's sources (P.0) to obtain a corresponding, executable piece of code (P.2) and generates a contract for it (P.1 and P.3). Here I assume contracts to be inferred from sources, but other approaches are also viable, e.g. using handwritten contracts or inferring them from compiled code. Then, the producer proves that the contract is sound w.r.t. the code (P.4 and P.5), i.e. the contract correctly represents the behavior of the code, and creates an evidence (P.7). The evidence contains information allowing for efficiently verifying the proof steps. Finally, code (P.6), contract (P.8) and evidence (P.9) are packaged and published.

The consumer retrieves (dashed arrow) the code package and extracts its content (possibly after checking its integrity through signature verification). Subsequently, it uses the attached evidence to verify whether the contract is valid w.r.t. the application code (C.0). If the test fails (C.2 → NO), the contract is proved to be invalid. Otherwise (C.2 → YES), the consumer validates the contract against its own policy (C.1 and C.3). If the policy is satisfied (C.4 → YES), the consumer obtains a formal proof that the code complies with the policy and can run it safely (bold state "Run"). Again, if this step fails (C.4 → NO), the process terminates with a failure. The reaction to failures depends on the customer preferences, e.g. the code can be rejected. Alternatively, the code can be secured by other means, e.g. through runtime monitoring.

The Security-by-Contract approach offers several advantages. Mainly, it formally guarantees that the mobile code will not violate the consumer's security policy. Moreover, in most cases it replaces the online cost of monitoring an application with an offline formal verification step.

Although it theoretically solves the problem of the secure execution of mobile code, some issues limit the applicability of this model to the actual mobile applications distribution systems. For instance, contract-policy compliance must be proved by means of computationally costly procedures, e.g. via model checking or automatic theorem proving, that the code consumer must execute on her device. This activity violates the assumptions about the resource limitations of the mobile devices. Instead, my proposal both (i) provides a formal security assessment framework and (ii) is tailored for actual mobile application distribution models.

10

Conclusion and Future Works

In this thesis I described a novel mobile code distribution framework, called **Secure Meta-Market** that supports the formal and automatic analysis of applications in order to enhance the security of Bring Your Own Device environments.

This thesis also presented an implementation of the Secure Meta-Market for the Android OS, called **BYODroid**. BYODroid supports Android-based devices and allows the security assessment of applications taken from the Google Play Store against corporate BYOD policies.

The experimental activities were performed by checking the compliance of **860 Android applications** taken from the Google Play Store against a real world BYOD security policy. The results confirm the effectiveness of the proposed approach, showing that more than 80% of the applications are successfully validated within 4 minutes.

Finally, I presented a version of BYODroid that has been prepared for the **NATO Communication and Information (NCI) Agency**, also discussing the validating activities performed at the agency premises in The Hague, Netherlands. This provided a further confirmation in the usability and scalability of the proposed approach.

Future Works Future directions of this research include the extension of the experimental activity and the applicability of the Secure Meta-Market. Indeed, a crucial aspect of the Secure Meta-Market is its capability to scale over a large number of devices and applications. In particular, I want to extend the verification of the Secure Meta-Market to application configurations, i.e. finite sets of applications.

To this aim, *partial model checking (PMC)* should be considered. In [93] PMC was proposed as a technique for extending the applicability of model checking to large models consisting of the parallel composition of simpler ones. PMC can be used to mitigate the state-explosion problem and considerably improve the scalability of the SMM paradigm.

Moreover, I plan to investigate different instances of BYOD policies involving various aspects of the security of mobile devices. Among them, contextual policies, e.g., those referring to the geographical location of the device, need further investigation.

A

Appendices

A.1 Technical Proof

Lemma 2. *For each* closed *(i.e., without free variables) expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \rhd H$ then either E is a value or $\omega, E \to \omega', E'$ (for some ω', E').*

Proof. By induction over the structure of E.

- Case `null`, u, x. Trivial.
- Case `new C`$(\bar{E})$. If each $E_i = v_i$ then E is a value and the property holds. Otherwise it suffices to apply the inductive hypothesis to the first $E_i \neq v$.
- Case E'`.f`. If $E' \neq v$ I apply the inductive hypothesis. Otherwise, by `T-FLD` I know that $E' =$ `new C`$(\bar{v})$ and I can conclude by applying $\texttt{FLD}_2$.
- Case E'`.m`(E''). If both $E' = v'$ and $E'' = v''$ I assume the premises of `T-METH` and I can apply $\texttt{METH}_3$. Instead if either $E' \neq v'$ or $E'' \neq v''$ I can directly apply the inductive hypothesis to $\texttt{METH}_1$ and $\texttt{METH}_2$.
- Case $\texttt{system}_\sigma\, E'$. If E' is not a value, I apply the inductive hypothesis and rule $\texttt{SYS}_1$. Otherwise, by `T-SYS` I know that $\Gamma \vdash E' : \mathcal{U} \rhd H'$ and the only suitable value for E' is a resource u. Thus, I conclude by applying $\texttt{SYS}_2$.
- Case `icast` E' and `ecast C` E'. Similar to the previous step.
- Case $I_\alpha(E')$. If $E' = v$ then E is also a value. Instead, if E' is not a value, I apply the inductive hypothesis and rule `INT`.

- Case E'.data. If E' is a value, by T-DATA it must be $E' = I_\alpha(v)$. Hence, I conclude by applying rule DATA$_2$. Otherwise, I simply apply the inductive hypothesis and rule DATA$_1$.
- Case if $(E' = E'')$ then $\{E_{tt}\}$ else $\{E_{ff}\}$. If one between E' and E'' is not a value, I conclude by applying the inductive hypothesis. Otherwise, I can apply either IF$_3$ or IF$_4$. In both cases, the property holds.
- Case $E'; E''$. Here I have two possibilities: either E' is a value or not. In both cases, E admits reduction (through SEQ$_1$ and SEQ$_2$, respectively).
- Case (C)E$'$. If E' is not a value, it suffices to apply the inductive hypothesis and rule CAST$_1$. On the other hand, by rule T-CAST I know that $E' =$ new D$(\cdots)$ and $D <: C$. Thus, I can apply CAST$_2$ and conclude.
- Case thread $\{E'\}$ in $\{E''\}$. Again, if either E' or E'' are not values, I use the inductive hypothesis and rule PAR$_1$ or PAR$_2$. Instead, if $E' = v'$ and $E'' = v''$ I conclude by applying rule PAR$_3$.

□

Definition 9.

$$[\![H]\!] = \{\omega \mid \exists H'.H \xrightarrow{\omega}{}^{*} H'\}$$

Lemma 3. *For each* closed *expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \triangleright H$ and $\omega, E \to^* \omega', E'$ then $\Gamma \vdash E' : \tau \triangleright H'$ and for all $\dot\omega' \in [\![H']\!]$ there exists $\dot\omega \in [\![H]\!]$ such that $\omega\dot\omega = \omega'\dot\omega'$.*

Proof. I first (1) prove the property for single step and then (2) I extend it to arbitrary long derivations.

1. By induction on the structure of E.
   - Case null, u, x. Trivial.
   - Case new C$(\bar{E})$. If each $E_i = v_i$ then E is a value and the property holds. Otherwise I type E in this way

$$\frac{\Gamma \vdash E_i : \tau_i \triangleright H_i}{\Gamma \vdash \texttt{new C}(\bar{E}) : \tau \triangleright \varepsilon \cdots H_j \cdots}$$

   where E_j is the first non-value in $\bar{E}$. Applying Lemma 2 I know that $\omega, E_j \to \omega', E'_j$. Hence, I apply the inductive hypothesis obtaining

$\Gamma \vdash E'_j : \tau_j \rhd H'_j$ and $\forall \omega'_j \in [\![H'_j]\!].\exists \omega_j \in [\![H_j]\!].\omega\omega_j = \omega'\omega'_j$. Then, by instantiating rule `NEW` I have

$$\omega, \texttt{new}\,\texttt{C}(\bar{v}, E_j, \ldots) \to \omega', \texttt{new}\,\texttt{C}(\bar{v}, E'_j, \ldots)$$

Since $A = [\![\varepsilon \cdots H_j \cdots]\!] = \{\omega_j\omega_{j+1}\cdots \mid \omega_j \in [\![H_j]\!] \wedge \omega_{j+1} \in [\![H_{j+1}]\!] \wedge \cdots\}$ and $B = [\![\varepsilon \cdots H'_j \cdots]\!] = \{\omega'_j\omega_{j+1}\cdots \mid \omega'_j \in [\![H'_j]\!] \wedge \omega_{j+1} \in [\![H_{j+1}]\!] \wedge \cdots\}$ (where the $\cdots$ parts are equal) I can conclude by observing that $\forall \dot{\omega}' = \omega'_j \cdots \in B$ I can find a trace $\dot{\omega} = \omega_j \cdots \in B$ such that $\omega\dot{\omega} = \omega'\dot{\omega}'$

- Case $E'.\texttt{f}$. If $E' \neq v$ I just apply the inductive hypothesis. Otherwise, by `T-FLD` I know that $E' = \texttt{new}\,\texttt{C}(\bar{v})$ and by applying $\texttt{FLD}_2$ I obtain $\omega, \texttt{new}\,\texttt{C}(\bar{v}).\texttt{f} \to \omega, v_f$ which trivially satisfies the property.
- Case $E'.\texttt{m}(E'')$. If either $E' = v'$ or $E'' = v''$ I assume the premises of `T-METH` and I can apply the inductive hypothesis and $\texttt{METH}_1$/$\texttt{METH}_2$. Instead if both $E' = \texttt{new}\,\texttt{C}(\bar{v})$ and $E'' = v'$ I apply $\texttt{METH}_3$ and I have

$$\frac{mbody(\texttt{m}, \texttt{C}) = \texttt{x}, E_m}{\omega, (\texttt{new}\,\texttt{C}(\bar{v})).\texttt{m}(v') \to \omega, E_m[v'/x, (\texttt{new}\,\texttt{C}(\bar{v}))/\texttt{this}]}$$

  However, typing the two sides of this transition I obtain the same history expression. Indeed, for each history expression H_m produced by typing the right side, I have for the left part $\varepsilon \cdot \varepsilon \cdot H_m$ which is trivially equivalent.
- Case $\texttt{system}_\sigma\, E'$. If E' is not a value, I apply the inductive hypothesis and rule $\texttt{SYS}_1$. Otherwise, by $\texttt{SYS}_2$ I have $\omega, \texttt{system}_\sigma\, u \to \omega \cdot \sigma(u), \texttt{null}$. Also by `T-SYS` I have $\Gamma \vdash E : \mathbf{1} \rhd \sigma(u)$. Since $\Gamma \vdash \texttt{null} : \mathbf{1} \rhd \varepsilon$, I have to show that $\exists \dot{\omega} \in \{\sigma(u)\}.\omega\sigma(u) = \omega\dot{\omega}$ which trivially holds.
- Case $\texttt{icast}\, E'$. Similarly to the previous step, if E' is not a value, I can simply apply the inductive hypothesis. Otherwise, by `T-IMPC` I must have $E' = I_\alpha(u)$ and $H = \alpha_?(u)$. Also, by $\texttt{IMPC}_2$ I have

$$\frac{\texttt{new}\, C(\bar{v}) \in receiver(\alpha)}{\omega, \texttt{icast}\, I_\alpha(u) \to \omega, \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

  and I type $\Gamma \vdash \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u)) : \mathbf{1} \rhd \tilde{H}$. Hence I must prove that $\forall \dot{\omega}' \in [\![\tilde{H}]\!].\exists \dot{\omega} \in [\![\alpha_?(u)]\!]$ such that $\omega\dot{\omega} = \omega\dot{\omega}'$. However, by the semantics of history expressions I have

$$\frac{\alpha_?(u) \xrightarrow{\alpha_?(u)} \varepsilon \qquad \dot{H} = \sum H'\{\alpha_?(u)/h\} \text{ s.t. } \bar{\alpha}_C h.H' \in \rho(\alpha) \text{ and } \chi \succcurlyeq C}{H \dot{\to} \dot{H}}$$

Then, by definition of ρ, I have that $\tilde{H} \sqsubseteq \dot{H}$ which suffices to conclude.

- Case $\texttt{ecast}\,\texttt{C}\,E'$. Analogous to the previous case.
- Case $I_\alpha(E')$. If $E' = v$ then E is also a value. Instead, if E' is not a value, I apply the inductive hypothesis and rule INT.
- Case $E'.\texttt{data}$. If E' is a value, by T-DATA it must be $E' = I_\alpha(v)$. Hence, I conclude by applying rule $\texttt{DATA}_2$. Otherwise, I simply apply the inductive hypothesis and rule $\texttt{DATA}_1$.
- Case $\texttt{if}\,(E' = E'')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{f\!f}\}$. If either E' or E'' are not values, I conclude by applying the inductive hypothesis. Otherwise, I can apply either $\texttt{IF}_3$ or $\texttt{IF}_4$ and the inductive hypothesis to E_{tt} and $E_{f\!f}$, respectively.
- Case $E'; E''$. Here I have two possibilities: either E' is a value or not. In both cases, E admits reduction (through $\texttt{SEQ}_1$ and $\texttt{SEQ}_2$, respectively). The property holds by the inductive hypothesis.
- Case $(\texttt{C})\texttt{E}'$. If E' is not a value, it suffices to apply the inductive hypothesis and rule $\texttt{CAST}_1$. On the other hand, by rule T-CAST I know that $E' = \texttt{new}\,\texttt{D}(\cdots)$ and $D <: C$. Thus, I can apply $\texttt{CAST}_2$ and conclude.
- Case $\texttt{thread}\,\{E'\}\,\texttt{in}\,\{E''\}$. If either E' or E'' are not values, I use the inductive hypothesis and rule $\texttt{PAR}_1$ or $\texttt{PAR}_2$. Instead, if $E' = v'$ and $E'' = v''$ I conclude by applying rule $\texttt{PAR}_3$.

2. By induction on the derivation length.

   - Base case. For zero-long derivations the property is trivially satisfied by the identity $\omega\dot{\omega} = \omega\dot{\omega}$.
   - Induction. I assume the property holds for n-long derivations and I apply (1) to prove that the property is preserved by a further step.

$\square$

Theorem 2. *For each* closed *expression E, history expression H, type τ and trace ω, if $\varnothing \vdash E : \tau \rhd H$ and $\cdot, E \to^* \omega, E'$ then there exist H' and ω' such that $H \xrightarrow{\omega'}{}^* H'$, $\varnothing \vdash E' : \tau \rhd H'$ and $\omega = \omega'$.*

Proof. A corollary of Lemma 3. $\square$

Lemma 4.

$$(A) \quad H \vDash \varphi \Longleftrightarrow H \nvDash \neg\varphi \qquad\qquad (B) \quad \varphi \equiv \neg\neg\varphi$$

Proof.

(A) For the right implication $H \vDash \varphi \Rightarrow H \nvDash \neg\varphi$. I assume $H \vDash \neg\varphi$ and by negation rule I have the contradiction $H \nvDash \varphi$. The other direction is similar.

(B) By applying (A) two times.

□

Theorem 3. $H \vDash \varphi_{//H'} \Longrightarrow H \parallel H' \vDash \varphi$

Proof. By induction over the structure of φ.

- Case tt. Trivial.
- Case $\neg\varphi$. I have $H \vDash (\neg\varphi)_{//H'} \Leftrightarrow H \vDash \neg(\varphi_{//H'}) \Leftrightarrow H \nvDash \varphi_{//H'}$. By Lemma 4 and inductive hypothesis I have $H \parallel H' \nvDash \varphi$ which suffices to conclude.
- Case $\varphi \wedge \varphi'$. From the PMC and conjunction rules I have $H \vDash \varphi_{//H'}$ and $H \vDash \varphi'_{//H'}$. Then I apply the inductive hypothesis and I obtain $H \parallel H' \vDash \varphi$ and $H \parallel H' \vDash \varphi'$ which imply $H \parallel H' \vDash \varphi \wedge \varphi'$.
- Case $\langle\sigma(u)\rangle.\varphi$. The PMC rule says that $H \vDash (\langle\sigma(u)\rangle.\varphi)_{//H'}$ reduces to

$$H \vDash \overbrace{\langle\sigma(u)\rangle.\varphi_{//H'}}^{A} \vee \overbrace{\bigvee_{H' \xrightarrow{\sigma(u)} H''} \varphi_{//H''}}^{B}$$

If A holds then by c-diamond rule both (i) $H \xrightarrow{\sigma(u)} \dot{H}$ and (ii) $\dot{H} \vDash \varphi_{//H'}$. Hence, by inductive hypothesis on (ii) I have that $\dot{H} \parallel H' \vDash \varphi$ and by concurrency rule using (i) premise I have that $H \parallel H' \xrightarrow{\sigma(u)} \dot{H} \parallel H'$. These two facts suffice to conclude that $H \parallel H' \vDash \langle\sigma(u)\rangle.\varphi$. Instead, if B holds, there must exist at least one H'' such that $H' \xrightarrow{\sigma(u)} H''$ and $H \vDash \varphi_{//H''}$. The latter implies (by inductive hypothesis) that $H \parallel H'' \vDash \varphi$, the former implies $H \parallel H' \xrightarrow{\sigma(u)} H \parallel H''$. These two facts imply (c-diamond) $H \parallel H' \vDash \langle\sigma(u)\rangle.\varphi$.

- Case $\langle\sigma(x)\rangle.\varphi$. I proceed similarly to the previous case. Applying the PMC rule to $H \vDash (\langle\sigma(x)\rangle.\varphi)_{//H'}$ I have

$$H \vDash \overbrace{\langle\sigma(x)\rangle.\varphi_{//H'}}^{A} \vee \overbrace{\bigvee_{H' \xrightarrow{\sigma(u)} H''} \varphi\{x/u\}_{//H''}}^{B}$$

If A holds then by a-diamond rule there exists u such that (i) $H \xrightarrow{\sigma(u)} \dot{H}$ and (ii) $\dot{H} \vDash \varphi\{x/u\}_{//H'}$. Hence, by inductive hypothesis on (ii) I have that $\dot{H} \parallel H' \vDash \varphi\{x/u\}$ and by concurrency rule using (i) premise I have that $H \parallel H' \xrightarrow{\sigma(u)} \dot{H} \parallel H'$. These two facts suffice to conclude that $H \parallel H' \vDash \langle\sigma(x)\rangle.\varphi$. Instead, if B holds, there exist H'' and u such that $H' \xrightarrow{\sigma(u)} H''$ and $H \vDash \varphi\{x/u\}_{//H''}$. The latter implies (by inductive hypothesis) that $H \parallel H'' \vDash \varphi\{x/u\}$, the former implies that $H \parallel H' \xrightarrow{\sigma(u)} H \parallel H''$. Applying (a-diamond) to these two facts I obtain $H \parallel H' \vDash \langle\sigma(u)\rangle.\varphi$, that is the thesis.

□

A.2 Promela Specifications

Listing A.1: Promela encoding of the policy of Example 13

```
proctype policy() {
/* Policy variables */
String agency_host = "agency.gov/";
Object agency_url = NULL;
bool connected = false;
/* Action parameters */
Object url;
String spec;
String host;
Object file;
Object this;
/* Channels */
mtype act;
Parameter par;

/* Main loop */
endwait:
system?act ->
if
:: (act == java_net_URL__init__s_) ->
  sys_par?par; spec = par.s;
  sys_par?par;
  sys_par?par; url = par.o;
  if
  :: (string_prefix(agency_host, spec)) ->
    { agency_url = url; goto endwait }
  :: else -> { goto endwait }
  fi;
:: (act == java_net_URL__init__s_s_i_s_) ->
  sys_par?par;
  sys_par?par; host = par.s;
  sys_par?par;
  sys_par?par;
  sys_par?par;
  sys_par?par; url = par.o;
  if
  :: (string_contains(agency_host, host)) ->
    { agency_url = url; goto endwait}
  :: else -> { goto endwait }
  fi;
:: (act == java_net_URL_openStream__) ->
  sys_par?par; this = par.o;
  if
  :: (this == agency_url) -> { connected = true;
                               goto endwait }
  :: else -> { goto endwait }
  fi;
:: (act == java_io_FileOutputStream_write__b_) ->
  sys_par?par; sys_par?par;
  if
  :: (!(connected)) -> { goto endwait }
  :: else -> goto fail
  fi;
:: (act == java_io_File_createTempFile__) ->
  sys_par?par;
  if
  :: true -> { goto endwait }
  fi;
:: (act ==
    android_bluetooth_BluetoothSocket_getOutputStream__
    ) ->
  sys_par?par;
  if
  :: (!(connected)) -> { goto endwait }
  :: else -> goto fail
  fi;
else goto fail
fi;
fail: assert (false);
}
```

Listing A.2: Promela encoding of the CFG of Example 14.

```
proctype activity() {

Object url;
Object file;
Object uis;
int i;
Object o;

/* Channels */
mtype intent;
Parameter par;

/* Activity is paused */
start:
broadcast?intent ->
  url = new_object();
  system!java_net_URL__init__s_;
  par.s = "https://agency.gov/data";
  sys_par!par;
  par.o = url;
  sys_par!par;

  system!java_net_URL_openConnection__;
  par.o = url;
  sys_par!par;

  file = new_object();
  system!java_io_File_createTempFile__;
  par.o = file;
  sys_par!par;

  do
  :: i = new_int();
     o = new_object();
     system!java_io_FileOutputStream_write__b_;
     par.i = i;
     sys_par!i;
     par.o = o;
     sys_par!par

  :: break
  od;
  goto start;
}
```

A.3 NCI Agency Policy

Listing A.3: Conspec encoding of the policy for NCI Agency project

```
SECURITY STATE

SESSION Str[20] pub_url = "http://agency/public";
SESSION Str[20] int_url = "http://agency/intern";
SESSION Str[20] cnf_url = "http://agency/confid";
SESSION Str[20] sec_url = "http://agency/secret";
SESSION Bool pub = false;
SESSION Bool i = false;
SESSION Bool cnf = false;
SESSION Bool sec = false;
SESSION Bool connected = false;
SESSION Bool deleting = false;
SESSION Bool encrypt = false;
SESSION Bool external = false;

BEFORE android.bluetooth.BluetoothDevice.
  createInsecureRfcommSocketToServiceRecord(*)
PERFORM
(false) -> { skip; }

BEFORE java.net.URL.<init>(Str[20] target)
PERFORM
(target.startWith(pub_url)) -> { pub = true; }
(target.startWith(int_url)) -> { i = true; }
(target.startWith(cnf_url)) -> { cnf = true; }
(target.startWith(sec_url)) -> { sec = true; }

BEFORE java.net.URLConnection.connect(Obj url)
PERFORM
(true) -> { connected = true; }

BEFORE java.io.File.<init>(*)
PERFORM
((!cnf && !sec) || !connected) -> { skip; }
```

```
BEFORE java.io.File.createTempFile(*)
PERFORM
(!connected || (!cnf && !sec)) -> { deleting = true; }

BEFORE java.io.File.deleteOnExit()
PERFORM
(true) -> { deleting = true; }

BEFORE java.io.FileOutputStream.<init>(*)
PERFORM
(deleting) -> { skip; }

BEFORE javax.crypto.Cipher.doFinal(*)
PERFORM
(true) -> { encrypt = true; }

BEFORE java.io.FileOutputStream.write(*)
PERFORM
(encrypt) -> { skip; }

BEFORE javax.crypto.Cipher.getInstance(Str[16] alg)
PERFORM
(!alg.strContains("SHA1")) -> { skip; }

BEFORE javax.crypto.Cipher.getInstance(Str[16] alg,
  Obj p)
PERFORM
(!alg.strContains("SHA1")) -> { skip; }

BEFORE javax.crypto.Cipher.getInstance(Str[16] alg,
  Str[0] p)
PERFORM
(!alg.strContains("SHA1")) -> { skip; }

BEFORE android.os.Environment.getExternalStorageDirectory()
PERFORM
(true) -> { external = true; }
```

```
BEFORE java.io.FileOutputStream.write(*)
PERFORM
(external) -> { skip; }

BEFORE org.apache.commons.net.ftp.FTPClient.<init>(*)
PERFORM
(false) -> { skip; }
```

Bibliography

[1] R. Ballagas, M. Rohs, J. G. Sheridan, and J. Borchers, “BYOD: Bring Your Own Device,” in *Proceedings of the Workshop on Ubiquitous Display Environments*, 2004. [cited at p. 1]

[2] I. Cook, “BYOD – Research findings released,” Nov. 2012. http://cxounplugged.com/2012/11/ovum_byod_research-findings-released/. [cited at p. 1]

[3] Apple Inc., “Apple App Review Guidelines,” Oct. 2013. https://developer.apple.com/appstore/guidelines.html. [cited at p. 1]

[4] Google Inc., “Google Play Developer Program Policies,” Oct. 2013. https://play.google.com/about/developer-content-policy.html. [cited at p. 1]

[5] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, “Jekyll on iOS: When Benign Apps Become Evil,” in *Proceedings of the 22nd USENIX Security Symposium*, pp. 559–572, 2013. [cited at p. 1]

[6] G. Costa, A. Merlo, and L. Verderame, “Market-based security for mobile devices,” *ERCIM News*, vol. 2013, no. 93, 2013. [cited at p. 2]

[7] A. Armando, G. Costa, and A. Merlo, “Bring your own device, securely,” in *SAC* (S. Y. Shin and J. C. Maldonado, eds.), pp. 1852–1858, ACM, 2013. [cited at p. 2, 18, 21, 23, 39, 51, 84]

[8] A. Armando, G. Costa, A. Merlo, and L. Verderame, “Enabling byod through secure meta-market,” in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec ’14, (New York, NY, USA), pp. 219–230, ACM, 2014. [cited at p. 2, 22, 36, 51, 73]

[9] Digital Services Advisory Group and Federal Chief Information Officers Council, "Bring Your Own Device – A Toolkit to Support Federal Agencies Implementing Bring Your Own Device (BYOD) Programs," tech. rep., White House, Aug. 2013. Available at `http://www.whitehouse.gov/digitalgov/bring-your-own-device`. [cited at p. 3, 55, 73]

[10] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. [cited at p. 8]

[11] J. R. Büchi, *The Collected Works of J. Richard Büchi*, ch. On a Decision Method in Restricted Second Order Arithmetic, pp. 425–435. New York, NY: Springer New York, 1990. [cited at p. 9]

[12] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, Feb. 2000. [cited at p. 11]

[13] A. Ghezzi, R. Balocco, and A. Rangone, "How a new distribution paradigm changes the core resources, competences and capabilities endowment: the case of mobile application stores," in *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010 Ninth International Conference on*, pp. 33–42, IEEE, 2010. [cited at p. 17]

[14] A. Armando, G. Costa, L. Verderame, and A. Merlo, "Securing the "bring your own device" paradigm," *Computer*, vol. 47, no. 6, pp. 48–56, 2014. [cited at p. 17, 22, 36, 72]

[15] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988. [cited at p. 17]

[16] H. Lockheimer, "Android and security," 2012. [cited at p. 18]

[17] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 559–572, USENIX Association, 2013. [cited at p. 18]

[18] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012. [cited at p. 18]

[19] A. Armando, G. Costa, A. Merlo, and L. Verderame, "Formal modeling and automatic enforcement of bring your own device policies," *International Journal of Information Security*, vol. 14, no. 2, pp. 123–140, 2015. [cited at p. 19]

[20] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," in *ACM Transactions on Programming Languages and Systems*, pp. 132–146, 1999. [cited at p. 22, 26, 29, 84, 85]

[21] U. Erlingsson, *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. [cited at p. 22]

[22] I. Aktug and K. Naliuka, "ConSpec – A formal language for policy specification," *Science of Computer Programming*, vol. 74, pp. 2–12, Dec. 2008. [cited at p. 22]

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, Apr. 1986. [cited at p. 22]

[24] A. Armando, G. Costa, and A. Merlo, "Formal Modeling and Reasoning about the Android Security Framework," in *Trustworthy Global Computing* (C. Palamidessi and M. D. Ryan, eds.), vol. 8191 of *Lecture Notes in Computer Science*, pp. 64–81, Springer Berlin Heidelberg, 2013. [cited at p. 23, 73, 84]

[25] F. Nielson and H. R. Nielson, "Type and effect systems," in *Correct System Design* (E.-R. Olderog and B. Steffen, eds.), vol. 1710 of *Lecture Notes in Computer Science*, pp. 114–136, Springer, 1999. [cited at p. 23, 84]

[26] M. Bartoletti, P. Degano, and G. L. Ferrari, "History-based access control with local policies," in *FoSSaCS* (V. Sassone, ed.), vol. 3441 of *Lecture Notes in Computer Science*, pp. 316–332, Springer, 2005. [cited at p. 23]

[27] C. Skalka, S. Smith, and D. Van Horn, "A Type and Effect System for Flexible Abstract Interpretation of Java," *Electronic Notes in Theorical Computer Science*, vol. 131, pp. 111–124, May 2005. [cited at p. 27, 30]

[28] M. Bartoletti, P. Degano, and G. L. Ferrari, "History-based access control with local policies," in *FoSSaCS*, pp. 316–332, 2005. [cited at p. 29, 30]

[29] A. Armando, G. Costa, A. Merlo, and L. Verderame, "Bring your own device, securely," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, (New York, NY, USA), pp. 1852–1858, ACM, 2013. [cited at p. 29, 72]

[30] C. Skalka and S. Smith, "History effects and verification," in *Second ASIAN Symposium on Programming Languages and Systems (APLAS)*, pp. 107–128, Springer, 2004. [cited at p. 30, 84]

[31] P. de Carvalho Gomes, A. Picoco, and D. Gurov, "Sound control flow graph extraction from incomplete java bytecode programs," in *Fundamental Approaches to Software Engineering* (S. Gnesi and A. Rensink, eds.), vol. 8411 of *Lecture Notes in Computer Science*, pp. 215–229, Springer Berlin Heidelberg, 2014. [cited at p. 32, 35, 85]

[32] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, (New York, NY, USA), pp. 27–38, ACM, 2012. [cited at p. 32, 85]

[33] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino, "Securing Java with Local Policies," *Journal of Object Technology*, vol. 8, pp. 5–32, June 2009. [cited at p. 32, 36, 84]

[34] R. Milner, "The Polyadic π-Calculus: a Tutorial," tech. rep., Logic and Algebra of Specification, 1991. [cited at p. 34]

[35] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino, "Model checking usage policies," *Trustworthy Global Computing: 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, pp. 19–35, 2009. [cited at p. 34]

[36] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *SIGPLAN Not.*, vol. 21, pp. 152–161, July 1986. [cited at p. 35]

[37] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 181–210, Apr. 1991. [cited at p. 35]

[38] U. Erlingsson, *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521. [cited at p. 36]

[39] L. Bauer, J. Ligatti, and D. Walker, “Composing security policies with polymer,” *SIGPLAN Not.*, vol. 40, pp. 305–314, June 2005. [cited at p. 36, 84]

[40] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe, “A flexible security architecture to support third-party applications on mobile devices,” in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW ’07, (New York, NY, USA), pp. 19–28, ACM, 2007. [cited at p. 36]

[41] I. Aktug, M. Dam, and D. Gurov, “Provably correct runtime monitoring,” in *FM 2008: Formal Methods* (J. Cuellar, T. Maibaum, and K. Sere, eds.), vol. 5014 of *Lecture Notes in Computer Science*, pp. 262–277, Springer Berlin Heidelberg, 2008. [cited at p. 36]

[42] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, “Security monitor inlining for multithreaded java,” in *ECOOP 2009 – Object-Oriented Programming* (S. Drossopoulou, ed.), vol. 5653 of *Lecture Notes in Computer Science*, pp. 546–569, Springer Berlin Heidelberg, 2009. [cited at p. 36]

[43] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan, “Matching in security-by-contract for mobile code,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 340–358, 2009. [cited at p. 36]

[44] I. Aktug and K. Naliuka, “ConSpec: A formal language for policy specification,” *Sci. Comput. Program.*, vol. 74, pp. 2–12, Dec. 2008. [cited at p. 37]

[45] I. Aktug and K. Naliuka, “Conspec–a formal language for policy specification,” *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 45–58, 2008. [cited at p. 38]

[46] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013. [cited at p. 40]

[47] G. Holzmann, “Tracing protocols,” *AT T Technical Journal*, vol. 64, pp. 2413–2433, Dec. 1985. [cited at p. 40]

[48] G. J. Holzmann, “An improved protocol reachability analysis technique,” *Softw. Pract. Exper.*, vol. 18, pp. 137–161, Feb. 1988. [cited at p. 40]

[49] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972. [cited at p. 40]

[50] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," in *FORMAL METHODS IN SYSTEM DESIGN*, pp. 275–288, 1992. [cited at p. 40]

[51] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe, "The S3MS .NET Run Time Monitor," *Electronic Notes in Theoretical Computer Science*, vol. 253, pp. 153–159, Dec. 2009. [cited at p. 45]

[52] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter, "Runtime monitoring for next generation Java ME platform," *Computers & Security*, vol. 29, no. 1, pp. 74–87, 2010. [cited at p. 45, 47, 62]

[53] N. Reddy, J. Jeon, J. A. Vaughan, T. Millstein, and J. S. Foster, "Application-centric security policies on unmodified Android," Tech. Rep. UCLA TR 110017, University of California, Los Angeles, Computer Science Department, July 2011. [cited at p. 47, 61]

[54] K. D. Cooper, T. J. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," tech. rep., Department of Computer Science, Rice University, 2002. [cited at p. 53]

[55] A. Amighi, P. de Carvalho Gomes, and M. Huisman, "Provably Correct Control-Flow Graphs from Java Programs with Exceptions," in *Formal Verification of Object-Oriented Software*, vol. 26 of *Karlsruhe Reports in Informatics*, (Karlsruhe, Germany), pp. 31–48, Karlsruhe Institute of Technology, Oct. 2011. [cited at p. 53]

[56] A. D. Guide, "Android Application Fundamentals," Oct. 2013. Available at `http://developer.android.com/guide/components/fundamentals.html`. [cited at p. 53]

[57] G. Holzmann, *The Spin model checker*. Addison-Wesley Professional, first ed., 2003. [cited at p. 54]

[58] R. V. Koskinen and J. Plosila, "Applications for the SPIN Model Checker – A Survey," Tech. Rep. 782, Turku Centre for Computer Science, Lemminkäisenkatu 14 A, 20520 Turku, Finland, Sept. 2006. [cited at p. 54]

[59] B. Long, J. Dingel, and T. N. Graham, "Experience Applying the SPIN Model Checker to an Industrial Telecommunications System," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), pp. 693–702, ACM, 2008. [cited at p. 54]

[60] G. Holzmann and D. Bosnacki, "The design of a multicore extension of the spin model checker," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 659–674, 2007. [cited at p. 54]

[61] H. Erdogmus, "Verifying Semantic Relations in SPIN," in *Proceedings of the 1st SPIN Workshop*, pp. 1–15, 1995. [cited at p. 55, 60]

[62] Chris Welch, "Google: Android app downloads have crossed 50 billion, over 1M apps in Play." Accessed on October 2013. [cited at p. 75]

[63] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pp. 627–638, 2011. [cited at p. 83]

[64] Y. J. Park, D. Chung, M. H. Dwijaksara, J. Kim, and K. Kim, "An Enhanced Security Policy Framework for Android," *Symposium on Cryptography and Information Security (SCIS 2011)*, 2011. [cited at p. 83]

[65] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, (New York, NY, USA), pp. 328–332, ACM, 2010. [cited at p. 83]

[66] A. Armando, R. Carbone, G. Costa, and A. Merlo, "Android permissions unleashed," in *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pp. 320–333, July 2015. [cited at p. 83]

[67] A. Castiglione, R. Prisco, and A. Santis, *E-Commerce and Web Technologies: 10th International Conference, EC-Web 2009, Linz, Austria, September 1-4, 2009. Proceedings*, ch. Do You Trust Your Phone?, pp. 50–61. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. [cited at p. 83]

[68] A. Merlo, M. Migliardi, N. Gobbo, F. Palmieri, and A. Castiglione, "A denial of service attack to UMTS networks using sim-less devices," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, pp. 280–291, May 2014. [cited at p. 83]

[69] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, pp. 259–269, June 2014. [cited at p. 83]

[70] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, (New York, NY, USA), pp. 1–6, ACM, 2014. [cited at p. 83]

[71] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015. [cited at p. 83]

[72] P. Kodeswaran, V. Nandakumar, S. Kapoor, P. Kamaraju, A. Joshi, and S. Mukherjea, "Securing Enterprise Data on Smartphones Using Run Time Information Flow Control," in *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, 2012. [cited at p. 83]

[73] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications," tech. rep., 2009. [cited at p. 83]

[74] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing*, pp. 291–307, Springer Berlin Heidelberg, 2012. [cited at p. 83]

[75] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010. [cited at p. 83]

[76] D. Schreckling, J. Köstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android," *Information Security Technical Report*, 2013. [cited at p. 83]

[77] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible MAC to android." [cited at p. 84]

[78] A. Merlo, G. Costa, L. Verderame, and A. Armando, "Android vs. seandroid: An empirical assessment," *Pervasive and Mobile Computing*, pp. –, 2016. [cited at p. 84]

[79] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe, "Security-by-contract on the .NET platform," *Information Security Technical Report*, vol. 13, pp. 25–32, Jan. 2008. [cited at p. 84, 85]

[80] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter, "Runtime monitoring for next generation Java ME platform," *Computers & Security*, vol. 29, no. 1, pp. 74–87, 2010. [cited at p. 84]

[81] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, Oct. 1977. [cited at p. 84]

[82] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000. [cited at p. 84]

[83] I. Attali, D. Caromel, and M. Russo, "A formal executable semantics for java," in *Princeton University*, 1990. [cited at p. 85]

[84] G. Bierman, G. M. Bierman, G. M. Bierman, M. Parkinson, M. J. Parkinson, M. J. Parkinson, A. M. Pitts, A. M. Pitts, and A. M. Pitts, "MJ: An imperative core calculus for Java and Java with effects," tech. rep., University of Cambridge, 2003. [cited at p. 85]

[85] D. Bogdanas and G. Roşu, "K-java: A complete semantics of java," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 445–456, ACM, 2015. [cited at p. 85]

[86] A. Amighi, P. de C. Gomes, D. Gurov, and M. Huisman, "Sound control-flow graph extraction for java programs with exceptions," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, (Berlin, Heidelberg), pp. 33–47, Springer-Verlag, 2012. [cited at p. 85]

[87] T. Blasing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," pp. 55–62, 2010. [cited at p. 85]

[88] J. R. Michener and T. Acar, "Managing system and active-content integrity," *Computer*, vol. 33, pp. 108–110, July 2000. [cited at p. 85]

[89] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, (New York, NY, USA), pp. 106–119, ACM, 1997. [cited at p. 85]

[90] R. S. V. V. S. SandeepBhatkar and D. DuVarney, "Model-carrying code: A practical approach for safe execution of untrusted applications," 2003. [cited at p. 85]

[91] N. Dragoni, F. Massacci, T. Walter, and C. Schaefer, "What the heck is this application doing? - A security-by-contract architecture for pervasive services," *Computers & Security*, vol. 28, no. 7, pp. 566–577, 2009. [cited at p. 85]

[92] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999. [cited at p. 85]

[93] H. R. Andersen, "Partial model checking," in *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pp. 398–407, IEEE, 1995. [cited at p. 88]