UNIVERSITÀ CA' FOSCARI – VENEZIA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Master Degree in Computer Science

Graduation Thesis

Author: Alessandro Frazza

An Information Flow Type System for Android

Supervisor:
Professor Michele Bugliesi

Assistant Supervisors:
Alvise Spanò, Ph.D
Stefano Calzavara, Ph.D

Academic Year 2011/12

To Elsa and Renato

Abstract

The growing adoption by mobile devices of the Android operating system has increased interest in security mechanisms which are able to ensure secrecy and integrity of user data. This thesis describes a new type system which, exploiting the Decentralized Label Model, allows developers to guarantee that their application respects the former security properties, in a way which is totally transparent to the end-user. The proposed framework tackles some peculiar challenges raised by the Android platform, such as a non-standard control flow, a non-typed communication system based upon Intents and the introduction of a support to some of the most recent features of Java, such as Generics. Being a mostly static analysis, the impact on performance is negligible. The thesis also describes as the hereby proposed solution blends in with the Lintent framework, which statically analyzes Android applications in order to detect privilege escalation attacks and verify inter-component communication.

Acknowledgments

There are indeed several people whose help allowed me to complete this work, and as such achieve my master's degree. First, I start by thanking my supervisor Michele Bugliesi, whose efforts in starting a research group made up of graduate and doctoral students was the spark that generated this thesis. His patience and willingness to help have been invaluable ever since my bachelor's degree thesis. Equally important were Alvise Spanò, the first of my two assistant supervisors. His undeniable design and programming skills were a key factor in implementing the type system in time for the due date. Priceless was also the help of my second assistant supervisor, Stefano Calzavara, as he never denied his help or its bright thoughts, no matter how many silly idea I bounced at him.

I have lost count of how many exams were made easier thanks to the help of some wonderful fellow graduate student, that is why I must not forget Alessio, Andrea, Giuseppe, Simone and Stefano. There is only one reason because I am not a mad, solitary workaholic, and it is because of the support of my friends. A laugh with them has always been the best way to wash problems away: Alice, Andrea, Anna, Chiara, Daniela, Daniele, Davide, Elisa, Emil, Giulia, Manuel, Mattia and Stefania.

A special mention must be made for my girlfriend Sara and her patience, who endured the sight of my back bent over my laptop computer for several months, always cheering me up in moments of discomfort. All my relatives do indeed deserve to be thanked, for their lovely care and support, especially my grandmother Alida and my sister Jessica, whose delightful food nourished me throughout all these years. At last, but first in order of importance, come my parents Catia and Paolo, which I will never thank enough for their support, both moral and financial, despite my undeniable grumpiness. They have always raised me with those sound values that let me be happy about the person I became to be.

Contents

List of Figures

List of Tables

1 Introduction

The ubiquitous presence of smartphones and tablets in our lives is nowadays a fact that cannot be ignored. People are every day more connected, with all sort of devices that are continuously communicating through the Internet. Albeit this represents a great achievement for mankind, it is not a mystery that cybercrimes are increasing, both in numbers and in variety. Hundreds of millions of handheld devices already contain sensitive personal, government and corporate data, huge numbers that are bound to increase. Hence, it is clear that IT security cannot be limited to desktop computers anymore, as protecting mobile data is felt as an increasingly urgent need ([32],[9]). Android, that as of now is the most wide-spread mobile Operating System[28], is no exception to this. Although system resources and user data are protected by permissions, they heavily relies on users carefulness in order to work properly. It has been already pointed out that security systems that do count on users for their effectiveness often end up being vulnerable[21]. Therefore, it is of the uttermost importance that devices are protected from harm in a way that is transparent to the end-users, relieving them from the burden of protecting sensitive information.

The goal of computer security is to guarantee two essential properties, *secrecy* and *integrity*. The former, also known as *privacy*, refers to ensuring that private data is not leaked to unauthorized parties. The latter, instead, requires sensitive data to be protected from any sort of damage that may be caused by other entities. Existing Android security mechanisms do not work well with untrusted code. When installing an application, indeed, the user grants it the required permissions. Once granted, though, these permissions do not limit the application anymore. If given the possibility to access the Internet, an application is free to misuse it without the user even knowing it. Alas, bona-fide code is often more dangerous than malicious one. As an example, hundreds of Android applications use IMEI[1] numbers and user locations to display ad-hoc advertisements[9], without informing the user. Whilst this may be borne by some, most users are likely to be annoyed, preferring not to be tracked.

[1]International Mobile Equipment Identity, a code that univocally identifies a mobile device.

Predictably, much effort has been made throughout the years to improve the situation. Enhanced policy systems[12], data-flow analysis[13] and real-time taint tracking[7] are just the tip of the iceberg of all proposed techniques to check the Android security problem. Regrettably, they all have some flaws. Run-time approaches do have serious overhead, making devices slower and more cumbersome. Static analyses, on the other hand, often only prevent a limited set of leakages. Besides, almost all proposals found in the literature are aimed at detecting weird behaviours on already deployed applications. Surprisingly enough, little-to-no effort has been put in detecting errors when they are being created, that is at compile-time, when the developer is programming.

The goal of this work is to explore the field of static Information Flow analysis in the Android world, by means of a new type system specifically tailored for it. Chapters 2 and 3 give the reader a brief, yet complete explanation of what Information Flow is and which is the model that has been adopted. Chapter 4 talks about the Android Operating System, pointing out the challenges that are posed by the non-standard behaviour of some of its features. In Chapter 5 it is possible to find a formalization for the proposed type system, *Android Flows Checker* , containing all the typing rules that allow to prevent illegal information flows. Finally, Chapter 6 highlights the most interesting facts about the implementation of the hereby proposed type system, which is integrated in a larger statical analysis framework called Lintent. At the moment in which this document has been written, both Lintent and Android Flows Checker were under active development, counting more than ten thousands line of code. For these reasons the source code has not been attached to this thesis, nevertheless all interested readers may find the source code for the F# implementation of Android Flows Checker at [38].

2

Information Flow

Security models have to ensure that secrecy and integrity of information is preserved. To preserve secrecy of data, any kind of unintended propagation must be prevented, be it accidental or malicious. To protect integrity, instead, it must be ensured that data is not overwritten or destroyed without the explicit authorization of their owner. Nowadays, most common models (e.g., *discretionary access control*) are able to wholly guarantee only the integrity of information, while they cannot offer the same guarantees regarding secrecy. As a matter of fact, they protect data disclosure, but once information exits the system they do not have the tools to prevent its unauthorized propagation[1]. This is due to the fact that even if access control mechanisms can help to prevent data from being released to unauthorized third parties, they cannot stop entrusted parties from mishandling confidential information. To better understand the limits of most common security models you could think of them as perfectly-crafted strongboxes: valuables inside them are effectively protected from harm and from being stolen, but once the owner retrieves them they immediately become vulnerable, and thus their safety relies entirely on how they are handled and used. It is evident that such a system cannot be deemed as safe, it is in fact common for sensitive information to be leaked due to the carelessness of people expected to protect it.

The ability of retrieving valuables from a safe and freely using them, without lowering their safety as if they still were in their container, would surely be an amazing feat. Even if such a prowess with common world objects is beyond our possibilities, it is exactly what Information Flow models enable in the digital world. This is achieved by monitoring the flow of data inside an application: Information Flow mechanisms, in fact, do not limit themselves in controlling that only authorized entities access confidential sources of information, but also prevent data from reaching untrusted sinks. This is achieved by tagging data as either secret or public, disallowing secret data to flow into public destinations. By doing so, the application is guaranteed to be safe both from direct attacks and bad programming habits, which are often even more harmful than most malicious theft attempts.

Looking at the literature we can distinguish two main groups of Information Flow models, *Dynamic* and *Static* ones. Even if radically different in the way they are implemented, they share most of the key concepts on how to get the job done. In both approaches, as a matter of fact, the programmer attaches a label[1] to confidential sources, pinpointing them as such, and to sinks, in order to identify them as either trustworthy or not. All these models simply propagate source labels to every value that is affected - even if only partially - by a source or another previously *labelled* value. Code is thus allowed to be executed only if no *tainted* [2] value reaches a sink marked as not trustworthy.

2.1 Dynamic Information Flow

A big cut of all the models to be found in the literature fall within the definition of a *Dynamic* Information Flow security model. Their key concept is to test the code at run-time, looking for any potential flow from a confidential data source to an untrusted sink. The reason why this solution is so widely spread is that you can test one instruction at a time, having the full control over what is being computed. In fact, in a *static* environment, whenever you encounter a branch in the flow of your code you have to take a strict conservative approach, assuming that the actual computation will fall in the worst case possible (from a security point of view), otherwise your static model would be broken. In a *dynamic* environment, instead, there is no point in analyzing all possibilities in a branch, as you are able to correctly predict which one will be executed. This is particularly useful in situation such as the one shown in Listing 2.1, where there's no point in **always** rejecting a piece of code which would be dangerous only in a small amount of cases.

The ability to eradicate false positives comes at a price, though. Run-time checks are costly and, in the average, they slow down the application by a 13-15%, as stated empirically in [6] and [7]. What hinders them most, however, is probably the necessity to incorporate these run-time checks within the executing environment, be it an operating system, a virtual machine or a sandbox interpreter. To patch such an environment, as a matter of fact, could be a troublesome operation, since many of them are proprietary softwares and to modify them could lead to copyright issues. Finally we must not forget that, even if we are dealing with open source software, such a cumbersome patch would be quite an intrusive one, therefore it is likely that many users would find it to be annoying, at the very least.

[1] Many names are given to this kind of metadata attached to sources and values, such as tags, labels, coloured taints and so on. For the sake of simplicity and clarity hereby we will refer to all this types of metadata as labels.

[2] Data labelled as secret, thus not to be propagated to unreliable entities.

Listing 2.1: This piece of code would be accepted at run-time as long as x is not 0

```
1  int m(int x){
2
3      if (x == 0){
4          ... information leakage ...
5      }
6      else{
7          return x+1;
8      }
9
10 }
```

2.2 Static Information Flow

As opposed to performing run-time tests, statical information flow analysis relies on a series of compile time checks that closely resemble the ones performed by a typical type checker[1]. Even though statical soundness sometimes forces them to reject legit programs - as shown earlier in 2.1 - they allow to certify a piece of code as trustworthy, without adding some run-time overhead, neither in space nor in time. It also comes with the pleasant side effect that no bits of information can be learned at run-time whenever a test fail, since they are all executed before the application is deployed to the final user.

Another important aspect that makes static models differ from dynamic ones is their typical target user. To check an application at run-time means it is likely ready to be deployed to the end-user device. To check the same application statically, instead, you need to have its source code, which, in general, is available only to the developer. Thus, while a dynamic information flow checker could be used both as a debugger by the developer and as a sort of anti-malware patch by the end-user, a static one is more likely to be used as a certifier from the developer.

3

Decentralized Label Model

Many information flow models limit themselves to labelling information as either secret or public. Others, instead, allow to specify a set of accredited parties to manipulate data. Both cases rely on a centralized trusted authority to label data. Although easy to model, it is clear that it is a quite simplistic abstraction. As a matter of fact, in real-world scenarios it is rather difficult to give a centralized notion of secret or public. People usually consider their own data as private, and are understandably less concerned about the secrecy of the data of other users. Nevertheless, for a security model all policies must be treated as equally important. For this reason the *Decentralized Label Model* focuses on providing security guarantees to different users and groups (or *principals*), rather than monolithic entities[1]. Hence, any principal might express its own security policy for every piece of information.

The key concept of this model is the support for *mutual distrust* between all the entities involved in the computation. This is done to allow the representation of all those scenarios where all principals may have conflicting interests on data, with possibly different opinions on each other. Principals can express requirements on sources of information, computed values and data sinks (or channels). The goal is to statically ensure that every possible flow of information respects *all* security requirements, from all involved principals. However, there are many plausible situations in which a principal may feel that a label has become overly restrictive, thus preferring to temporarily relax its restrictions. To handle this situation, at any point of the computation the Decentralized Label Model allows principals to declassify information, so as to be able to carry out operations that would not be allowed otherwise. By allowing a principal to lower its security requirements, it is possible to add expressivity to the code without hampering data security. It must not be forget, in fact, that every principal expresses its own policies. For this reason, if both Alice and Bob express their requirements on a value v, relaxing Alice's policy does not concern Bob. Indeed, the model contemporaneously enforces all restrictions, so Bob constraints are still in effect.

3.1 Motivating Example

When the DLM was first proposed, the authors of the article shown some motivating examples to help the readers to better understand the usefulness of their model. One of those depicts a group of researchers (***R***) which are performing a medical study on some data collected from the patients (***P***) of an hospital (***H***). It is reasonable that even if the patients gave permission to the researchers, they do not want specific details - such as identifying or wealth information - to be leaked. It would be understandable from the patients to do not fully trust the researchers, especially if they were part of a for-profit organization (e.g., a pharmaceutical company), as they could try to exploit patients' medical and personal information to their advantage (e.g, by contacting the patients and trying to sell them their products). Instead, it is more likely that they would give their trust to an automated data extractor (***E***) - which would be distributed by a highly reliable entity (such as the government's health ministry) - that strips the database from any kind of confidential information.

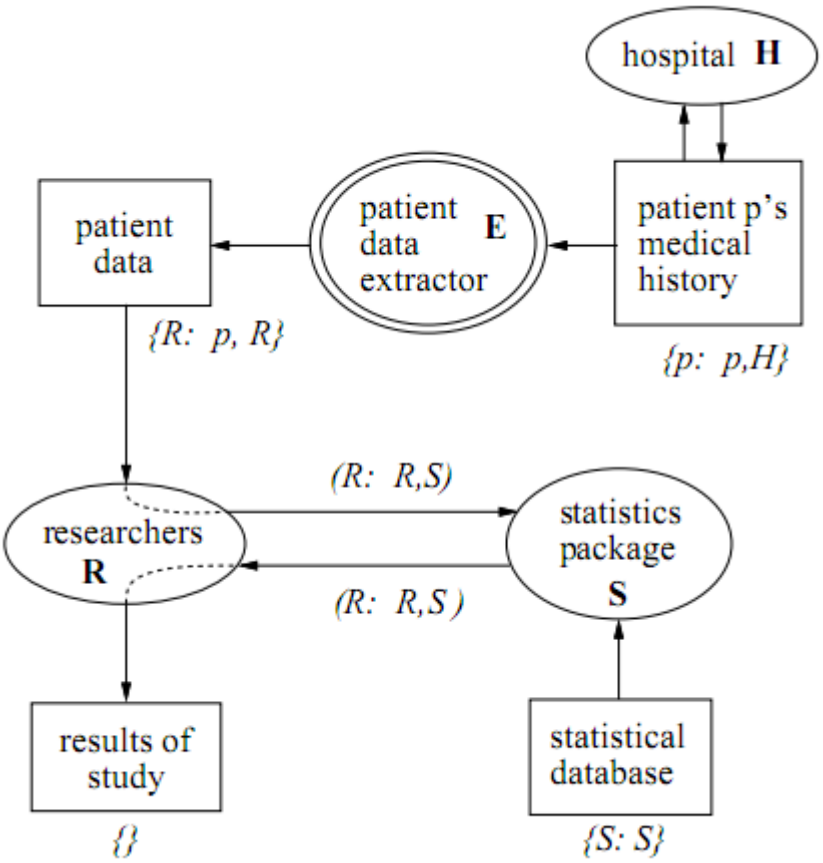

Figure 3.1: Medical Study Scenario

The scenario is depicted in Figure 3.1, with the medical history of every patient that is labeled as being owned by the patient (p) itself - the left hand side of the label { *p:p,H* } - and as being accessible by the patient and the hospital as described by the right hand side. The extractor then operates with the authority granted by both the patient and the hospital, producing a new record retained by the researchers but still readable by p. The researchers would like to supply those data to a statistics package ($\boldsymbol{S}$) they obtained from a third party, but they do not want it to perform a leakage of any sort. Thus, they relabel their data as { *R: R,S* } which means that they will still be owned by the researchers, but that also the statistics package can access them. This will work because any *principal* at the right of the colon does **not** have the ability to distribute the information to anyone outside those permitted by the owner (i.e, $\boldsymbol{R}$ and $\boldsymbol{S}$). Since the results of the computation are influenced by both the medical data and the statistical database, the information produced at the end will be labeled with the *joint label* { *R: R,S; S: S* }, which corresponds to enforcing **both** policies at the same time. However, this joint label only permits flows to $\boldsymbol{S}$[1] and, so as to return the results of the computation to the researchers, they *declassify* their data to { *R: R,S* } (basically removing the second label) just before sending them.

3.2 Principals

A key role in the model is played by *principals*. A principal can be an entity of any kind, be it a person, an user group, a role, a company or any sort of legal person. Any principal is given the possibility to express a security requirement, or *policy*, for any information related to the program. Should it **not** express a requirement on certain data, then it would consider them to be public, thus distributable to anyone. If two principals p and q are both entitled to express policies on the same information, they can exclude the other one from the list of readers, thus disallowing them to access it. This is considered to be perfectly fine, as it models the case in which two conflicting owners are not able to agree upon the release of their information. A conflicting and troublesome scenario, indeed, yet absolutely legit and plausible. It is important to notice that excluding a principal is completely different than removing its policy. Excluding an entity from an information means that it cannot read or write it any more, but its requirement are still enforced. Removing one's policy means that it may still be able to read or write data, but its restrictions are no longer applied.

[1] This is due to the fact that the first label permits flows to R and S, while the second to S. Since both policies must be enforced, we can obtain all flows allowed by the joint label by applying the intersection between the flows allowed by each label, i.e., $\{R, S\} \cap S \equiv \{S\}$

3.2.1 Acts-For Relationship

There is a reflexive and transitive relation between principals, called *acts-for* relationship, whose symbol is $\succeq$. We say that $p \succeq q$, or p *acts for* q, when the principal p has been granted authority by q and that every action taken by p is implicitly considered to be authorized by q. The *acts-for* relationship can be used to implement common concepts such as *groups* or *roles*. As an example, a company could create a principal *Employee* to represent all of its workers, in addition to a personal one for any of them (e.g, *Alice* and *Bob*). In such a situation, both $Alice \succeq Employee$ and $Bob \succeq Employee$ would hold, as both *Alice* and *Bob* are employees. Notice that a principal can act for many different principals at the same time so, should the same company have a manager called *Charlie*, then both $Charlie \succeq Employee$ and $Charlie \succeq Manager$ would be true.

3.2.2 The Top Principal

The first of two special pre-defined principals is the *Top Principal*, which is basically a fictional principal that can act for everybody, but that *nobody* can act for it. Should an information be labeled as being owned - and readable - by *Top* alone then no one would be able to access it. It is represented by the symbol $\top$.

3.2.3 The Bottom Principal

The *Bottom Principal* is basically the opposite of Top. It is a principal for whom anybody **can** act for, but it cannot act for any other principal. Should an information be labeled as being owned by *Bottom* alone then it would be considered as public, since any principal is implicitly considered to have been granted at least the same authority as Bottom. It is represented by the symbol $\bot$.

3.3 Confidentiality

It has already been said that principals may express two kinds of security requirements on information. The first of these regards the *confidentiality*, or *secrecy*, of information. Any principal p that expresses a confidentiality policy on some data, defines that, for what concern itself, only the principals listed in the policy are allowed to read that information. Please notice that this does not mean that p believes data have been read or edited by those readers, but it simply gives them its own authorization to access them.

3.3.1 Reader Policies

These requirements on confidentiality are expressed through security policies called *reader policies* (or *confidentiality policies*). Such a policy is written as

$$o : r_1, r_2, ...r_n$$

where o represents the principal that is the *owner* of that policies and r_1 up to r_n represent the list of allowed *readers*. The owner of a policy is the principal that imposes the security requirements represented by the policy itself and is *always* considered to be listed amongst the readers of that policy, even if not explicitly defined as such. A reader r of the policy is trivially a principal that is allowed to access the information. Thus the only difference between the owner and a reader is that the former also has the power to modify the policy itself, be it by modifying the set of readers or by deleting the policy altogether. It is to be noted that every principal acting for a principal listed in the policy is implicitly added to the list with the same role of the principal it is acting for, even if not expressively mentioned.

Policy	**Owner**	**Allowed readers**
Alice:	Alice	Alice
Alice: Alice	Alice	Alice
Alice: Bob, Charlie	Alice	Alice, Bob, Charlie
Alice: Bob ($Charlie \succeq Alice$)	Alice, Charlie	Alice, Charlie, Bob
Alice: Bob ($Charlie \succeq Bob$)	Alice	Alice, Bob, Charlie
$\top$:	Top	No one can read
$\top$: Alice	Top	Alice
$\bot$:	Everyone	Everyone
$\bot$: Alice	Everyone	Everyone
Alice, Bob: Charlie	Invalid policy: only one principal can be listed as explicit owner	
: Bob	Invalid policy: at least one principal must be listed as owner	

Table 3.1: Examples of reader policies.

Definition 3.3.1. (Readers Function). Let o be the owner of a confidentiality policy c. Let q be another generic principal. We define as *readers(o,c)* the set of principals allowed to access a certain information, based on the requirements imposed by the owner o and the policy c. This list of readers is composed of the owner o, the readers $r_1, r_2, ...r_n$ listed in the policy and any other principal that can act for either o or one of the readers r_i.

$$\text{readers}(o, c) \triangleq \{q \mid q \succeq o \;\; or \;\; \exists i \in [1, n] \; s.t. \;\; q \succeq r_i\}$$

3.3.2 Conjunction of Reader Policies

It is allowed to conjugate reader policies, so as to enforce warranties requested by multiple principals. The resulting joint confidentiality policy is defined as the policy that enforces all policies and it is written as $c \sqcup d$, where c and d are two confidentiality policies. In order to speed up the writing process of a list of conjugated policies, they conjugation is often also represented by a semicolon (**;**).

In terms of the *readers* function, we define the set of readers of a conjunction as the set of all readers that are part of the reader set of all the policies involved in the conjunction. It is, in other words, nothing more than the common intersection operation of the Set Theory. The most careful readers will have already noticed that even the owner of a policy will be excluded from the joint reader set if just one of the policies does not list it as reader. Thus, it is safe to say that $c \sqcup d$ is at least as restrictive as both c and d.

$$\text{readers}(o, c \sqcup d) \triangleq \text{readers}(o, c) \cap \text{readers}(o, d)$$

Policies	**Reader Set**
Alice: ;	{Alice}
Alice: Bob; Bob:	{Bob}
Alice: ; $\bot$:	{Alice}
Alice: Bob; Bob: Alice	{Alice, Bob}
Alice: Bob; Bob: Alice, Charlie; Charlie: Bob	{Bob}
Alice: Bob; $\top$:	{$\top$} (*No one*)
Alice: ; Bob:	{$\top$} (*No one*)
Alice: Bob; Bob: Charlie; Charlie: Alice	{$\top$} (*No one*)

Table 3.2: Examples of conjunctions between reader policies.

3.3.3 Disjunction of Reader Policies

The disjunction of two or more reader policies is intuitively the opposite of the conjunction, and it is written as $c \sqcap d$. Therefore, the resulting policy is defined as the policy that lists all readers that are allowed by at least one of the policies in question. The set of all readers authorized by a disjunction policy is thus the set of all readers that are part of at least one reader set of the involved policies. In terms of Set Theory operations, it corresponds to the union operator. We can also say that $c \sqcap d$ is at most as restrictive as either c or d.

$$\text{readers}(o, c \sqcap d) \triangleq \text{readers}(o, c) \cup \text{readers}(o, d)$$

Policies	Reader Set
Alice: $\sqcap$	{Alice}
Alice: $\sqcap$ Bob:	{Alice, Bob}
Alice: Bob $\sqcap$ Bob:	{Alice, Bob}
Alice: Bob $\sqcap$ Bob: Alice	{Alice, Bob}
Alice: Bob $\sqcap \top$:	{Alice, Bob}
Alice: $\sqcap \bot$:	{$\bot$} (*Everyone*)
Alice: Bob $\sqcap$ Bob: Charlie $\sqcap$ Charlie: Alice	{Alice, Bob, Charlie}
Alice: Bob $\sqcap$ Bob: Alice, Charlie $\sqcap$ Charlie: Bob	{Alice, Bob, Charlie}

Table 3.3: Examples of disjunctions between reader policies.

3.3.4 Ordering on confidentiality

There exists a relation over confidentiality policies, which is written as $c \sqsubseteq_C d$ and is read *c no more restrictive than d*. We define it in the following way:

$$\sqsubseteq_C (c, d) \triangleq \text{ readers}(o, c) \supseteq \text{readers}(o, d)$$

If, in other words, c is *no more restrictive than d*, then the set of readers allowed by the first policy is a superset of the set of readers allowed by the second policy. The relation $\sqsubseteq_C$ forms a pre-order over confidentiality policies and thus a lattice, where the **least** restrictive policy (or *bottom level*) is written as $\bot$:$\bot$, while the **most** restrictive one (or *top level*) is written as $\top : \top$. For what concerns the conjunction and the disjunction previously described, the former is the *join* operator of this lattice, while the latter is the *meet* operator. It is obvious that an information labeled with a low level confidentiality policy, such as the *bottom* one, can be accessed by more principals and can be used in more contexts than a higher level one. Viceversa, the higher the policy, the lesser the places that information can be used into.

Alice:	$\sqsubseteq_C$	Alice:
Alice: Bob	$\sqsubseteq_C$	Alice:
Alice:	$\not\sqsubseteq_C$	Alice: Bob
Alice: Bob	$\sqsubseteq_C$	Alice: ; Bob:
Alice: Bob	$\sqsubseteq_C$	Charlie: Bob (*iff Charlie* $\succeq$ *Alice*)
Alice: Bob	$\not\sqsubseteq_C$	Charlie: Bob (*generally can't relate, different owners*)
Alice: Bob, Charlie	$\sqsubseteq_C$	Alice: Bob; Alice: Charlie
Alice: Bob; Alice: Charlie	$\not\sqsubseteq_C$	Alice: Bob, Charlie
Alice: Bob	$\not\sqsubseteq_C$	Charlie: David
Alice: Bob $\sqcap$ Alice: Charlie	$\sqsubseteq_C$	Alice: Bob, Charlie
Alice: Bob, Charlie	$\sqsubseteq_C$	Alice: Bob $\sqcap$ Alice: Charlie
Alice: $\sqcap$ Bob: Charlie	$\sqsubseteq_C$	Alice: Charlie ; Bob: Charlie
Alice: $\sqcap$ Bob:	$\not\sqsubseteq_C$	Alice: Bob, Charlie

Table 3.4: Examples of ordering over reader policies.

3.4 Integrity

Integrity has long been shown to be the dual of Confidentiality[8] and so are integrity and confidentiality policies. They represent the integrity of the information in terms of its believed provenance.

3.4.1 Writer Policies

Writer policies (just another name for integrity policies) are used by principals to specify their beliefs on who may have influenced some data. It is written as:

$$o \leftarrow w1, w2, ..., wN$$

Likewise to reader policies, they have an owner, but instead of being followed by a colon, it is followed by a left-faced arrow that precedes a list of principals known as *writers*. It is extremely important to understand that, through a writer policy, the owner does **not** specify which principals are allowed to alter information, but tells to the model which principals the owner *believes* that **may** have affected the information in question. Like confidentiality ones, integrity policies do automatically include in the list of writers both the owner and any principal that can act for the owner or any of the listed writers.

Policy	**Owner**	**Influencing writers**
Alice←	Alice	Alice
Alice← Alice	Alice	Alice
Alice← Bob, Charlie	Alice	Alice, Bob, Charlie
Alice← Bob ($Charlie \succeq Alice$)	Alice, Charlie	Alice, Charlie, Bob
Alice← Bob ($Charlie \succeq Bob$)	Alice	Alice, Bob, Charlie
⊤←	Top	No one have influenced the datum
⊤← Alice	Top	Alice
⊥←	Everyone	Everyone
⊥← Alice	Everyone	Everyone
Alice, Bob← Charlie	Invalid policy: only one principal can be listed as explicit owner	
← Bob	Invalid policy: at least one principal must be listed as owner	

Table 3.5: Examples of writer policies.

Definition 3.4.1. (Writers Function). Let o be the owner of an integrity policy i. Let q be another generic principal. We define as *writers(o,i)* the set of principals that the owner o believes may have affected the information labelled with policy i. This list of writers is composed of the owner o, the writers $w_1, w_2, ...w_n$ listed in the policy and any other principal that can act for either o or one of the writers w_j.

$$\text{writers}(o, i) \triangleq \{q \mid q \succeq o \;\; or \;\; \exists j \in [1, n] \; s.t. \;\; q \;\succeq\; w_j\}$$

3.4.2 Conjunction of Writer Policies

Dually with respect to reader policies, the conjunction of writer policies is represented by $c \sqcap d$, where c and d are two writer policies. Despite the swap in symbols, the meaning is altogether the same of the conjunction of reader policies. In terms of the *writers* function, we define the set of writers of a conjunction as the set of all writers that are part of the writer set of all the policies involved in the conjunction. Thus, exactly for confidentiality, it is the intersection of two or more sets.

$$\text{writers}(o, c \sqcap d) \triangleq \text{writers}(o, c) \cap \text{writers}(o, d)$$

Policies	**Writer Set**
Alice← ;	{Alice}
Alice← Bob; Bob←	{Bob}
Alice← ; ⊥←	{Alice}
Alice← Bob; Bob← Alice	{Alice, Bob}
Alice← Bob; Bob← Alice, Charlie; Charlie← Bob	{Bob}
Alice← Bob; ⊤:	{⊤} (*No one*)
Alice← ; Bob←	{⊤} (*No one*)
Alice← Bob; Bob← Charlie; Charlie← Alice	{⊤} (*No one*)

Table 3.6: Examples of conjunctions between writer policies.

3.4.3 Disjunction of Writer Policies

In a manner similar to the conjunction, when we talk about the disjunction of *writer* policies we use the opposite symbol with respect to the disjunction of *reader* policies: $c \sqcup d$. Nevertheless the meaning is exactly the same, a writer is believed to have influenced a piece of information if it appears in any of the policies involved, as you would do with the union of two or more sets. In order to speed up the writing process, their conjugation is often represented with a semicolon (**;**).

$$\text{writers}(o, c \sqcup d) \triangleq \text{writers}(o, c) \cup \text{writers}(o, d)$$

Policies	**Writer Set**
Alice← ⊔	{Alice}
Alice← ⊔ Bob←	{Alice, Bob}
Alice← Bob ⊔ Bob←	{Alice, Bob}
Alice← Bob ⊔ Bob← Alice	{Alice, Bob}
Alice← Bob ⊔ ⊤←	{Alice, Bob}
Alice← ⊔ ⊥←	{⊥} (*Everyone*)
Alice← Bob ⊔ Bob← Charlie ⊔ Charlie: Alice	{Alice, Bob, Charlie}
Alice← Bob ⊔ Bob← Alice, Charlie ⊔ Charlie: Bob	{Alice, Bob, Charlie}

Table 3.7: Examples of disjunctions between writer policies.

3.4.4 Ordering on Integrity

The *no more restrictive* relation $\sqsubseteq_I$ on integrity is defined dually with respect to the one on confidentiality:

$$\sqsubseteq_I (c, d) \triangleq \text{ writers}(o, c) \subseteq \text{writers}(o, d)$$

This means that for what concerns the lattice built upon the relation on integrity, it is the disjunction that corresponds to the *join* operator, while the conjunction is the *meet*. Therefore the **least** restrictive policy is the one with the highest level of integrity, that is $\top \leftarrow \top$. As a matter of fact, this policy represents that an information can have been influenced by Top alone. The **most** restrictive policy, instead, is $\bot \leftarrow \bot$, because it is the one with the lowest level of integrity. This is due to the fact that anyone could have altered the information, thus it is not to be trusted. The reader should be careful about the fact that while a piece of information with confidentiality labeled as $\bot$:$\bot$ can be used in any context, a datum labeled with an integrity of $\bot \leftarrow \bot$ is most restricted..

Alice←	$\sqsubseteq_I$	Alice←
Alice← Bob	$\not\sqsubseteq_I$	Alice←
Alice←	$\sqsubseteq_I$	Alice← Bob
Alice← Bob	$\not\sqsubseteq_I$	Alice← ; Bob←
Alice← Bob	$\sqsubseteq_I$	Charlie← Bob (*iff Alice* $\succeq$ *Charlie*)
Alice← Bob	$\not\sqsubseteq_I$	Charlie← Bob (*In general can't relate, different owners*)
Alice← Bob, Charlie	$\not\sqsubseteq_I$	Alice← Bob; Alice← Charlie
Alice← Bob; Alice← Charlie	$\sqsubseteq_I$	Alice← Bob, Charlie
Alice← Bob	$\not\sqsubseteq_I$	Charlie← David
Alice← Bob ⊔ Alice← Charlie	$\sqsubseteq_I$	Alice← Bob, Charlie
Alice← Bob, Charlie	$\sqsubseteq_I$	Alice← Bob ⊔ Alice← Charlie
Alice← ⊔ Bob← Charlie	$\not\sqsubseteq_I$	Alice← Charlie ; Bob← Charlie
Alice← ⊔ Bob←	$\sqsubseteq_I$	Alice← Bob, Charlie

Table 3.8: Examples of ordering over writer policies.

3.5 Labels

As hinted in Section 3.1, principals express their security requirements on data labelling them. A *label* is a set of confidentiality *and* integrity policies. It is written as a list of policies, separated by a semicolon, within curly brackets:

$$\{o : a, b; o \leftarrow c\}$$

Notice that while a semicolon separating two confidentiality policies represents their conjunction, a semicolon separating integrity policies represents their disjunction. While this choice of notation may seem to be confusing at the least, it will become clearer to the reader once he will have read of the *join* of two labels. For now it will suffice to understand that a label containing policies separated by semicolons is *at least as restrictive* as all the policies it contains.

It is possible for a label to do not have any specified integrity (or secrecy) policy, in which case it receives a default one with the Bottom principal as lone owner and writer (reader). Should a label have no policy - or should it not exist at all - then it would be called *empty label*, i.e., $\{\bot\colon \bot\ ;\ \bot\leftarrow \bot\}$. Notice that while there is no strict need to follow any order between policies, with the possibility to mix integrity policies of a label in the middle of a list of confidentiality policies, from now on we will always describe labels by writing all the confidentiality policies first, followed by the integrity ones, for the sake of readability.

3.5.1 Ordering on Labels

Regarding labels, the *no more restrictive than* relation $\sqsubseteq$ is defined by exploiting $\sqsubseteq_C$ and $\sqsubseteq_I$. In detail, $\{c; d\} \sqsubseteq \{c'; d'\}$ if and only if $c \sqsubseteq_C c'$ and $d \sqsubseteq_I d'$. Let us call $C(L)$ the confidentiality projection of a label - i.e., the confidentiality policy of that label - and as $I(L)$ the integrity project. We can now redefine the *join* ($\sqcup$) and *meet* ($\sqcap$) operations over the lattice of labels in the following way.

$$L_1 \sqcup L_2 \triangleq \{C(L_1) \sqcup C(L_2); I(L_1) \sqcup I(L_2)\}$$

$$L_1 \sqcap L_2 \triangleq \{C(L_1) \sqcap C(L_2); I(L_1) \sqcap I(L_2)\}$$

In other words, the join of two labels is the conjunction of its confidentiality policies and the disjunction of its integrity policies; the disjunction of two labels works vice-versa.

{}	$\sqsubseteq$	{⊥: ; ⊥←}
{⊥: ; ⊥←}	$\sqsubseteq$	{}
{}	$\sqsubseteq$	{⊥: ; ⊤ ←}
{⊥: ; ⊤ ←}	$\sqsubseteq$	{⊤ : ; ⊥←}
{Alice: Manager}	$\sqsubseteq$	{Alice: Charlie}
{Alice← Manager}	$\not\sqsubseteq$	{Alice← Charlie}
{Alice: Employee; Employee← }	$\sqsubseteq$	{Alice: Charlie; Manager←}

Table 3.9: Examples of label ordering with the hierarchy found in Figure 3.5.1.

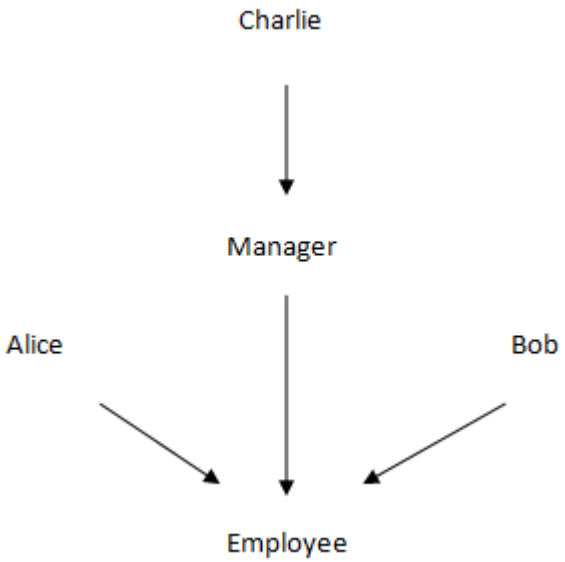

Figure 3.2: Example hierarchy, arrows from the acting-for principal to the acted-for.

3.5.2 Labels Relabeling

The primary goal of information flow models is to avert the application from leaking information. Whenever information is transferred (e.g., by applying a variable as argument for a function or through variable assignment) the old label of the data is lost and, from that moment onwards, those data will have the same label as their destination's. This operation is called *relabeling*. Therefore, the new label must be at least as restrictive as the former one, otherwise, after the relabeling, there may occur some information flows that were previously prohibited. Any relabeling where this condition holds is called *safe*. As we have seen before, we say that $L_1 \sqsubseteq L_2$ is true as long as L_1 is less restrictive than - or equal to - L_2; whenever this is true then a relabeling from L_1 to L_2 is to be considered safe. In other words, every time that the label of an information is *restricted*, then the relabeling is safe.

3.5.3 Declassification

As already introduced, they may be some situation where a certain policy is too strict to allow an essential information flow, as, for example, in an email client application where a confidential email - written by Alice and to be read only by Bob - must be sent over the internet, which, being an untrustworthy medium, must be labeled as public. Such a relabeling would never be allowed, preventing the application from sending the email, rendering it useless. In order to solve this problem, the client can encrypt the message with a key to be known by Bob alone and *declassify* the data (i.e., lower the security requirements expressed by the label) so that they could be sent over the network.

At any moment of its execution, the application runs with the *authority* to act on behalf for a list of principals, possibly empty. Being a *decentralized* model based on mutual distrust, it is obvious that a principal may declassify only those policies that he owns or that are owned by a principal it acts for, thus the process must run with the authority of that principal. This way it is granted that a malicious - or bad programmed - application will not hamper the secrecy of data with labels of principals who have not bestowed on it their authority, because even if it removes all labels it is allowed to remove, the other principal will still have the guarantee that their requirements will still be enforced.

4

Android

Android is - as of now - the most widespread operating system to be designed primarily for touch-screen mobile devices such as smart phones and tablets[28]. It is an open source project built upon a Linux kernel, with low-level libraries and APIs written in a combination of Java and C/C++ languages, acting as a middleware through which applications can access the phone[7]. Developed by Android, Inc. - initially funded by and now property of Google, Inc. - it allows third party developers to write custom *apps* which are written in Java and compiled in a custom byte-code known as **DEX**[1].

4.1 Security Overview

Every application is executed in a sandbox environment, that is its own interpreter instance of a *Dalvik Virtual Machine*[9], so it is isolated from the rest of system's resources. In order to access them, it must do so through the system's APIs, which will accept calls for a given resource only from those applications that have been granted the specific permission. An application can list its required permissions by adding them into its own *manifest file*, i.e., an XML file that contains miscellaneous information about the application itself and to be included in the deployment package. Every time the user installs a new app through the Google's Play Store[2] it is presented the list of permissions requested by the app itself, so as to verify that it does not require suspicious permissions. Generally, as an example, a game application should not ask for sending SMSes, therefore, listing such a permission should lead the user to believe that the game could maliciously send messages to premium-rate telephone numbers and thus avoid installing the application. Otherwise, by confirming its download, the user *implicitly* grants all of the permissions that have been listed by the app.

Despite this double security layer, an average Android device is far from being safe from harms. First of all, the whole permission system relies on user choices,

[1] Dalvik EXecutable.

[2] The one and only official channel for downloading and installing Android applications.

who could fail to notice possible threats due to lack of carefulness or knowledge. Additionally, the initial developer confusion and incomplete documentation reduced the effectiveness of this system, with about one-third of Android apps being over-privileged[10]. At last, the system has been proven to be vulnerable to threats such as *privilege escalation* attacks[11]. The framework which our proposed tool is part of detects and prevents all of these problems, warning the developer whenever his application is found to be over-privileged or vulnerable. For further information the reader is referred to [2].

4.2 Apps Structure

An Android application is composed of

- A group of standard Java classes,
- A group of resources (e.g., icons, sounds),
- A manifest file.

However, the type checker proposed in this thesis needs only to access the Java source code, therefore we will not cover how the manifest and the resources work. The essential building blocks of an Android application are called *components*. There are four kind of components, each one fulfilling a specific role and acting as entry point for the system into the application.

4.2.1 Activities

An Activity is a single, focused thing that the user can do and it generally corresponds to one single screen of the application, automatically managing most of the *User Interface* for the developer. An app can be composed of several Activities and each one of them can be started by any other component to be found in the system, as long as the developer allows it. When it is the user that starts the application, though, the first Activity to be invoked at the beginning of the application is called *main* (or *launcher*) Activity. Thus, a simple Android application with a main background screen and one pop-up menu window would be generally made up of two Activities, with the background screen being the *main* one.

4.2.2 Services

A Service is a component designed to run in the background and perform long-running or remote operations, such as downloading data from the internet or playing some music while the user might well be using a complete different application. They

do not provide any kind of user interface, but they can either be *started* by or *bounded* to any Activity. A service can be started by calling its own *startService()* method and it will run indefinitely until *stopService()* is invoked, even if the Activity that first started it had been killed in the meantime. Otherwise, an application that has a service bound to it is offered a client-server interface that allows the application to interact with the Service by **IPC**[3]; such a service runs as long as it has at least one living process bounded to it.

4.2.3 Broadcast Receivers

A Broadcast Receiver is a component that replies to broadcast messages sent throughout the whole system. Many broadcast messages are sent from the system (e.g., low battery alert), but they can also be generated by custom components. Although they do not have an user interface, they can create and display notifications in the status bar. Most commonly, however, they act as a sort of *gateway* for other components and are intended to perform basic operations such as starting an Activity or a Service in response for the propagation of a certain event. Broadcasts can either be registered dynamically at runtime through the system API or statically by publishing the request in the application's manifest. It is possible to enforce a single permission both in the sending and in the receiving process. By sending a broadcast with a certain permission it is ensured that it will be received only by those applications that have been granted that permission, as to not send confidential data to those that should not have access to it. By contrast, on the receiving end of a broadcast message, it is possible to filter out all those broadcasts that have been sent by components that do not have the specified permission, as to have a certain assurance about the integrity of data.

4.2.4 Content Providers

The goal of Content Providers is to manage a shared set of application data, not necessarily on the file-system, as they can also be saved over the internet or in any other persistent storage location. It allows to access information through a database-like interface, querying and modifying data. They can be either used to be shared by different applications (e.g., the system's provider for contacts information) or to store data relevant to the application alone (e.g., a note-pad app could use a content provider to store user's notes).

[3]InterProcess Communication

4.2.5 Intents

Components communicate through objects called *Intents*. An Intent is an abstract description of an operation to be performed by the receiver. It offers an interface through which Activities and Services are started and broadcasts are sent. Generally it is composed of a *dictionary* containing key-value pairs and a target that describes which component should receive those data. In particular you can either start a new Activity to perform a task (e.g., send an email) by passing the Intent as argument for the *startActivity()* method or to get a result (e.g., pick a contact and return its data) by supplying it to a *startActivityForResult()*. Services are started by invoking *startService()*, while they are bound with a call to *bindService()*.

4.3 Activity Lifecycle

Most examples about challenges and solutions related to the Android lifecycle are done on Activities, as they are the most used Component. Nevertheless, Services are modeled likewise, and the few implementation differences do not alter the bulk of the underlying reasoning. Hence, everything that will be said on Activities regarding this topic is to be considered valid for Services either.

In a typical Android scenario, the user opens many applications in a brief period of time and navigates through multiple Activities. As a consequence, Activities are continuously swapped in and out of focus, rendering their lifecycle quite unusual. To manage this, Android gives the developer the opportunity to extend the code of seven *callback*[4] methods, each of them invoked by the system whenever the Activity reaches a certain state.

Whether it is started by the user or by another component, an Activity's entry point is always its *onCreate()* method. As illustrated in Figure 4.3, it is followed by an escalation of callbacks that create the Activity, which are the *onStart()* and the *onResume()*. Once an Activity goes on the background, but is still partially visible, it becomes paused and the system invokes its *onPause()* method. If the Activity instead becomes fully hidden then the system marks it as stopped and will stay as such until it is killed either by the user or by the system due to memory shortage. Of course, if an hidden Activity returns to the foreground then its state will be reverted to the *Resumed* one. This is obtained by calling the *onResume()* callback if the Activity was previously *paused*, or by invoking its *onRestart()* if it was *stopped*.

[4] A function or piece of code to be passed as argument to another function or to the operating system, usually to be called in response to a certain event.

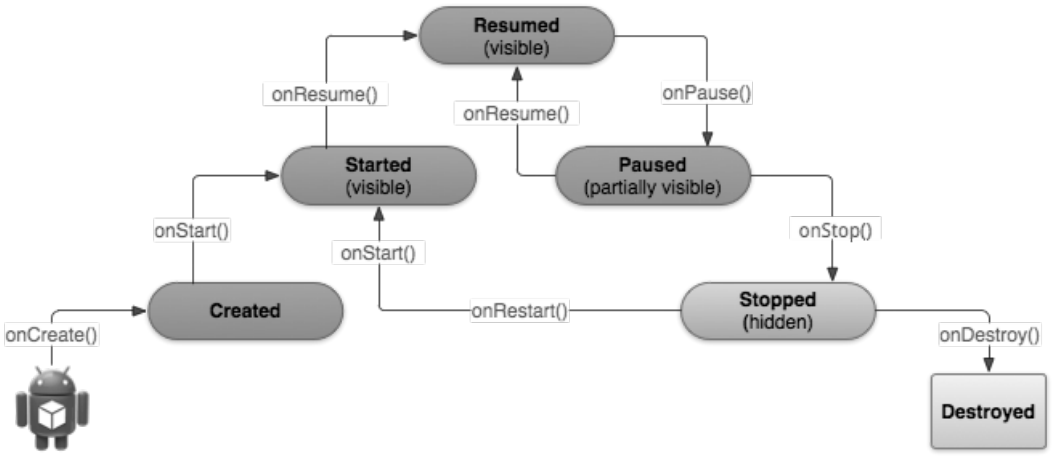

Figure 4.1: Lifecycle of an activity.

4.4 Peculiar Challenges

There is a whole set of peculiar challenges that needs to be tackled in order to successfully implement an Information Flow model for the Android operating system. Their proposed solutions will be detailed later on at Chapter 5.

4.4.1 Creating an Activity

There are two API methods that allows the developer to instantiate a new Activity: *startActivity()* and *startActivityForResult()*. They both take an Intent as argument in order to allow the system to discern which activity must be created. In a general Java application, an Information Flow checker analyzes the invoked method with the purpose of verifying if it is a legal call. This would not work as expected in an Android application, because, at the end of the day, those method calls consist in an invocation of the target activity's *onCreate()* method. So, every time an Activity creation is encountered, rather than analyzing the *startActivity()* method, the correct method to be checked is the *onCreate()* of the Intent's target Activity. Therefore, it must be devised a mechanism that allows the checker to recognize and exploit this pattern.

4.4.2 Returning Results

Every time an Activity must return any kind of response to whoever created it, the *setResult()* method is supposed to be used. However, returning results arises problems similar to those encountered when creating Activities, because the corresponding code is to be found at the caller's *onActivityResult()* method. Likewise, a similar mechanism that recognizes this second pattern must be implemented, as

performing checks exploiting it would allow the tool to maintain its correctness.

4.4.3 Terminating an Activity

Whenever an Activity completes its job, it is expected to invoke the *finish()* method, which carries out all the preliminary tasks required to shut down an Activity. Nevertheless, this API method does not abruptly end the computation, but it returns the control of the computation back to the Activity that invoked it, which will end its current lifecycle callback method. What happens next depend on the context, in fact there are three possibilities:

- The last lifecycle callback to be executed was the ***onCreate()***. Then, once the computation of the *onCreate()* is completed, the ***onDestroy()*** is invoked.
- The last lifecycle callback to be executed was the ***onStart()***. In such circumstances, the following method to be executed is the ***onStop()***, with the *onDestroy()* as its consequent.
- The last lifecycle callback to be executed was the ***onResume()***. In this case, the next callback to be executed would be the ***onPause()***, followed by *onStop()* and *onDestroy()*.

The unusual control flow that follows a *finish()* could lead to leaks due to implicit flows[5] As an example, in Listing 4.1 the value of the confidential boolean variable *secret* should never be assigned to the public boolean variable *leak*. With the purpose of showcasing a possible attack, a conditional *finish* invocation is performed based on the value of *secret* and the value of *leak* is adjusted based on whether or not the *onStart()* callback is executed. As a matter of fact, should *secret* be **true**, then the *finish()* would not be performed and the value of *leak* would be assigned to **true** inside the *onStart()* method. Otherwise, should *secret* be **false**, then the value of *leak* would be set to **false** after the execution of the if branch, without being further modified later.

4.4.4 Tracking Key-Value Pairs

Whenever the system destroys an Activity due to exceptional events (such as a memory shortage), it needs a way to recreate it as soon as the user navigates back to it. This is achieved by using a set of saved data - called the *instance state* - that describes the state of the activity prior to its destruction. Normally, this

[5]Implict flows will be covered in detail in Section 5.2. For the moment it will suffice to know that it is a possible indirect gain of information by an attacker with respect to confidential data, due to the control flow of the program.

Listing 4.1: Example of implicit flow exploiting the finish() method.

```
protected void onCreate(Bundle savedInstanceState){

    if (!secret)
        finish();
    leak = false;

}

protected void onStart(){

    // Here we are sure that secret is equal to true.
    leak = true;

}
```

process of state saving and restoring is completely automatic and transparent to the developer. In some occasions, though, it may be necessary for the developer to save additional information. Should that need arise, he can store the data within an instance of a *Bundle* object. The developer obtains this object on a special method called *onSaveInstanceState()*, where it can be filled with all the necessary data. The Bundle thus populated is then a formal parameter for the onCreate() and the onRestoreInstanceState() methods. It works like a dictionary that stores key-value pairs. While keys must be subtypes of *String*, values can be of any built-in type (such as *int*) or subtypes of *Serializable*. The *Bundle* class is not the only one that represents a built-in dictionary in Android, though, as Intents work in the same manner. If not taken care of, these dictionaries could be used as a way to launder restrictive labels into more permissive ones. Therefore, it is crucial to keep track of every information that enters both dictionaries, so as to be able to compute the correct label for every value that is retrieved from them.

4.4.5 Handling Intents

Intents themselves can be used as dictionaries in the same way as Bundles. For that reason, they must be taken care of likewise. As previously hinted in section 4.4.1, though, they are much more than mere dictionaries. Amongst everything, they also contain a target, which is the activity to be started and expected to retrieve their data. This rises the additional problem of understanding which component is the target of an Intent. In most cases it is a trivial task, since it is good practice to use *class literals* (e.g., **MyActivity.class**) or *final static* variables[6] for intra-

[6]Constants fields whose value is guaranteed to be immutable during the computation.

application communication, but no language construct prevents the developer to use expressions, *non-final* variables or to ask the user at run-time which should be the target of the Intent. It is thus necessary a mechanism that allows the checker to determine the target of an Intent with the highest degree of success possible.

4.4.6 Supporting Generics

Although the *Android Software Development Kit* does not exactly relate to any *Java Standard Edition* version, as of now it uses a subset of *Apache's Harmony SE 6* libraries[27], which can be roughly approximated to *Java SE 1.6* version, implementing most of its features and functionalities. Amongst these functionalities there are Generic Types, which allow to define types to be parameters in the definition of a class, interface or method. Hence, a type checker specifically tailored for Android needs to support Generics, adding a further layer of complexity in its implementation.

5

Information Flow Typing

In previous chapters we had a glance at the environment in which our newly devised type system works. Here, instead, we explain how it is conceived, formally detailing how the concepts of the Decentralized Label Model are translated into it. Built on top of the Java type system, its purpose is to represent and monitor all information flows within Android applications, describing how the challenges shown in Section 4.4 are tackled and solved. It works under the lone assumption that the source code has been already checked by the Java compiler or, at the least, that it would not be rejected by it. We believe this assumption to not pose any meaningful restriction, as there is no point in checking Information Flow properties in an application that would not be compiled nevertheless. Further details regarding the implementation are deferred to Chapter 6.

5.1 Motivations

There is a good amount of literature regarding Android security, with plentiful techniques developed to ensure some information flow properties at system level. Amongst the many solutions, it is possible to find techniques such as run-time leakage detection mechanisms [7], operating system modifications [12] and data-flow analysis [13]. However, all these techniques have severe limitations, such as a significant run-time overhead[1] or the necessity to have a custom version of the operating system, which is likely to cause annoyances such as stability or compatibility issues. Surprisingly enough, typing techniques have instead received little to no attention, with few notable exceptions such as [15]. Unfortunately, these few tools that use an off-line approach to statically check Android applications always deal with bytecode. While this approach allows - in principle - to analyze software just before the deployment on the device, it does not help the developer to avoid errors and potential issues while they are still being created, i.e., during the development. This gap in the literature is rendered more critical by the fact that the need to improve the development process of Android applications is widely recognized as being of

[1] In addition to the obvious encumbrance of the computation, an higher CPU load causes a sensible battery life reduction, where nowadays there is a quest for reducing energy consumption.

uttermost importance ([17] and [18]). Furthermore, there is not even some kind of proposal to impose an acceptable degree of discipline. Thus, our goal is to make a first step towards the filling of this lacuna, allowing the developer himself to certify its application as safe and well-typed, solving all previously mentioned issues upstream. We believe this approach to be a better one than the devise of fancy mechanisms to be used downstream, as all they can do is to limit themselves at limiting damages.

5.2 Implicit Flows

We have already seen that it is relatively easy to apply the DLM with the purpose of avoiding *explicit* illegal flows, such as direct assignments. Although, there are more subtle information flows called *implicit* flows. As the name suggests, an implicit flow consists in an indirect transfer of information from a source to a sink, caused by the control flow of the program itself. Implicit flows may allow attackers to gain knowledge about confidential data without having to directly access them. In Listing 4.1 we have already seen one of the most common examples of implicit flows, that is a side-effect occurring inside an if-then-else statement, whose if-clause is based on the value of confidential data.

There is a well-known technique used to prevent implicit flows, as first described in [16]. It consists in using the so-called *program counter label* ($\tilde{pc}$). It describes *the maximum amount of information* that is possible to learn just because a certain statement is computed in the given context. Every time an expression is computed, the program counter label is joined with the label of the expression itself and the resulting one substitutes the expression's former label. Therefore, the label of an expression now also carries information about the security requirements of the context, preventing implicit flows. Applying this technique to the example 5.1, the beginning of the computation itself gives no knowledge about confidential data, and for this reason the starting $\tilde{pc}$ is $\{\}$. This means that no security requirement risks to be broken by an implicit flow yet. The first assignment ($public = false$) still does not influence the program counter label, since nothing can be discovered by the simple fact that the assignment is performed. When the *if*-statement is encountered, however, the program counter label must be updated, because every computation performed inside the *then*-branch will happen only if secret is equal to two. This reveals one bit of information, which in general is much less than the whole information carried by a variable. Although, static checkers need to take a cautious approach, so every expression computed inside the branch is considered to carry as much information as *secret* does. The $\tilde{pc}$ is then raised to $\{secret\}$, that is the same label attached to the variable *secret*. The label of the literal *true* is joined with the program counter label

Listing 5.1: Basic pseudo-code example of an implicit flow.

```
1 main (){
2
3     public = false;
4     if (secret == 2) then
5         public = true;
6     another_public = 4
7
8 }
```

and is the same of the variable *secret*. Thus, the following assignment is rejected by the tool, because $\{secret\}$ is more restrictive than $\{public\}$.

At last, exiting from a conditional statement reverts the $\tilde{pc}$ to the value it had before entering it. The assignment to the variable *another_public* can be considered legit, since the $\tilde{pc}$ is reverted to $\{\}$ just after the if-statement is performed. The obvious reason is that the context at this point of the computation does not carry any information whatsoever, because whether or not the program executed the previous *if-statement* is irrelevant, the latest assignment is performed in any case. It is important to notice that at any time the program counter label can either be raised by the join with another label or can be reverted to an old value. Hence, it is safe to assume that during the type-checking phase *it will never be less restrictive* than its starting value.

There are some scenarios in which $\tilde{pc}$ cannot be reverted to its previous value. This happens, as an example, when a return statement occurs inside a conditional statement. Every time that a return is executed, the method terminates and gives control back to its caller. As such, every statement or expression that appears after a return can be executed only if its conditional branch is skipped, otherwise the method would have been terminated by the return. This allows to learn some information about the conditional expressions of all such branches, generating implicit flows, because the program counter label is reverted at the end of a conditional statement. To avoid creating overly complicated rules for $\tilde{pc}$, the problem is solved by introducing a new label, the *termination label* (or $\tilde{tl}$). Starting from the Bottom Label, every time that a return is encountered, it is joined with the current program counter label. Unlike $\tilde{pc}$, though, it is never relaxed, so it represent the maximum amount of information that can be learned by knowing that the method did not terminate yet. Hence, every time that a label is joined with the program counter label, it is also joined with the termination label.

5.3 Labels

As seen in Section 3.5, DLM labels are treated pretty much the same way as types are treated in an average programming language. Any standard Java type τ may be labeled with any label expression L. Information flows are permitted only between those locations where the type at the source is a subtype of that at the destination. In other words, in addition to standard Java type-checking rules, the type system requires the destination label to be at least as restrictive as the source label.

5.3.1 Label Syntax

The formal syntax for labels is more complex than the one informally described in previous chapter. This is because the type checker allows the developer to extend the label attached to another variable or class field, with the purpose of making the process of labeling data faster and more clear. The complete grammar for generating labels can be found at Table 5.1.

TERMINALS:

IDENTIFIER	::=	[‘a’–‘Z’ | _ | ‘$’] [‘$a$’–‘$Z$’ \| ‘0’–‘9’ \| _ \| ‘$’]*
SEMICOLON	::=	;
READ	::=	: | →
WRITE	::=	←

NON-TERMINALS:

label	::=	tyargs element (SEMICOLON element)*]
tyargs	::=	(< label (COMMA label)* > SEMICOLON)?
element	::=	IDENTIFIER | policy
policy	::=	IDENTIFIER (READ | WRITE) IDENTIFIER*

Table 5.1: The syntax for labels used in our type system.

5.3.2 Method Labels

During the type checking phase, every method invocation needs to be checked so as to verify that it may not generate any illegal flow. A first naive attempt could be to look for the method to be invoked and type-check it on-the-fly. However, this would lead into an explosion in complexity, requiring some muddled strategy to cope with mutually recursive functions, which would produce an infinite loop otherwise. Therefore, the necessity of finding a mechanism that would allow to type-check a method only once is to be considered of uttermost importance.

During the execution of a program, any method call can be seen as a sort of black box that takes some data, operates some kind of magic and then returns a result. Each of these three steps could hide one or multiple illegal information flows, so they must be checked accordingly. In the first place, it must be ensured that no information given as argument to the method ends up in a destination with a less restrictive label. Then, it must be verified that also the body of the method does not generate any illegal flow, with side-effects as the prime suspects. At last, the result itself must not be labeled as more restrictive than its destination within the caller. It is easy to see that this partition itself is not enough, because even if the second step is context independent, the first and the third are not. Hence, it is still necessary to type-check methods at every call site.

In order to test for legality a method as a module on its own, as a matter of fact, there is still a lack of information about the caller's $\tilde{pc}$ and the labels of the arguments. The safest and easiest approach is to simply assume both to be at *Top* level, but this would make the type system so much restrictive to become completely useless in every real world situation. A much wiser idea is to behave in the same way as the type system of Java does, i.e. to add types for the $\tilde{pc}$, the input arguments and the return value in the method declaration. These are respectively the *Begin Label* (L_{BG}), the *Parameter Labels* and the *Return Label* L_{RT}.

The Begin Label can be considered as an upper bound of the $\tilde{pc}$ at the moment of the invocation. It means that the invocation is legal only if the $\tilde{pc}$ of the caller is less restrictive than - or equal to - the Begin Label. The method is thus checked with the value of the Begin Label used as starting value for the $\tilde{pc}$. By having the guarantee that the caller's $\tilde{pc}$ will be equal or lower than the method's starting $\tilde{pc}$, it is safe to say that any implicit flow that would be considered illegal at the call site is considered illegal also within the body of the method.

In any sound type system, if the method by itself is well-typed then any invocation is well-typed as long as any supplied argument is a subtype of the corresponding parameter. Likewise in our type system, *parameter labels* are used as supertypes of the actual argument labels. If there is no information flow within the method using these upper bounds as labels for its parameters, any invocation of the method is legal as long as the supplied arguments are at most as restrictive as the corresponding parameters.

The same is true for the Return Label. At the call site the type of the result of a given method is the one specified in the Return Label, which can flow to any value which is *at least* as restrictive as itself. Within the method, instead, every return

statement is checked so as to verify that its expression is *at most* as restrictive as the Return Label.

5.3.3 Label Inference

Standard java field, variables and methods all require to have a label attached to them. To relieve the programmer from the dull task of writing unnecessary labels, our type system is designed to perform some degree of inference[2]. Whenever a variable declaration is not supplied with a label by the developer, it is inferred to have the same label of the initializing expression. If the declaration does not have an initializer, then it is temporarily considered to be untyped. As soon as a value is assigned to the variable, its label is inferred and updated accordingly. As unsafe as it could seem, it is indeed a sound approach. As a matter of fact, should the need of knowing the type of a variable arise before it had been assigned a value, then it would mean that the programmer is trying to use an uninitialized variable. Not only this should be treated as an error in any case, but is also forbidden by the Java compiler. Unlike variables, class fields are not inferred. It is due to the fact that fields can be used outside the class they are defined into, hence their type could be needed when their class has not been type-checked yet.

5.3.4 Default Labels

In real world scenarios it is likely that a big chunk of the data manipulated by the application is not to be considered confidential. Should the developer be forced to define every label that the type checker is not able to infer, even the irrelevant ones, he would probably consider it a nuisance. For this reason, every time that a required label cannot be inferred, it is used a default value in its stead, depending on the kind of location the label should be attached to:

- **Field.** Class fields are given the Bottom Label, i.e. $\{\bot\colon \bot\ ;\ \top \leftarrow\ \top\}$, which is the least restrictive label. It conservatively ensures that no confidential data is stored within them. Should the need of storing confidential data inside a class field arise, then it would be necessary from the programmer to manually specify its label. This is to prevent unsafe and undesired side-effects, as the lack of a supplied label could be as well accidental as intentional.

- **Begin Label.** If not supplied, the method Begin Label is considered to be the Top Label, i.e. $L_{BG} := \{\top :\ \top\ ;\ \bot\leftarrow \bot\}$. It is the most restrictive label. Being an upper bound on the caller's $\tilde{pc}$, it means that the method can be

[2]Please notice that the label supplied by the programmer *always* takes precedence over the inferred one

invoked from anywhere in the program, but it cannot perform any side-effect except those whose sink is labeled with the Top Label.

- **Parameter Label.** By default, method parameters are labeled with the Top Label ($\{\top : \top ; \bot \leftarrow \bot\}$). As for the Begin Label, it is an upper bound for the actual argument label, meaning that the default label allows every expression to be assigned as value for the parameter, maximizing at the same time method usability and security.

- **Return Label.** In general the result of a function can be considered to depend directly on its input data. Thus, if missing, the Return Label is computed as the join of all parameter labels.

$$L_{RT} := L_1 \sqcup L_2 \sqcup ... \sqcup L_n, \quad where L_i is the label of the i-th parameter.$$

5.3.5 Dynamic Labels

Despite the effort to type-check the application in a purely statical fashion, there are certain use-cases that need some degree of dynamic type-checking. As an example, an application that requires an user to authenticate, may want to permit or deny certain flows, based on the user that logged in. This would be indeed impossible if all labels were static, because a static type checker needs to know all the principals at run-time. For this reason AFC also handles *dynamic labels* (or run-time labels). At compile-time, when the type-checking of a method begins, every dynamic label is given both a lower and an upper bound, representing the range of legal values for the label. Should a dynamic label have a run-time value falling outside this range, then the application would be terminated. The starting lower and upper bound are respectively the Bottom and the Top label, standing for a value which is yet unknown. As the type checker sifts through the method, these constraints are progressively updated. To be more precise, at any point of the type-checking, these constraints are the most permissive ones that would have passed all previous checks. Once the type-checking of the method is completed, all dynamic labels are evaluated. If, for at least one label, the lower bound is not a subtype of the upper bound, then a type error is reported. In any other case - included the one in which the two constraints are not comparable - they are saved as meta-data for the method. Hence, every time that the method is executed, all dynamic labels are checked against these constraints. Should even only one constraint be failed, then the whole application would be shut down.

5.3.6 Authority

It has been already hinted that at any point of the computation, the application runs with the authority of some principals. Basically it is the ability to *act-for*

a certain set of principals. As a matter of fact, inside a method it is possible to compromise the security of a principal p only if that method has been granted the authority by p. As an example, the developer is *not* allowed to relax a policy expressed by p through declassification in a method that does not have the authority to act for p.

A method can receive authority from two entities, which are the class it has been defined into and the method that has called it. The set of principals a method is allowed to *act-for* is called *authority set*. Every time that a method needs some authority, it must specify it in its own declaration. Hence, a call to that method is considered legal if and only if the union of the authority set of the caller and the authority set of the callee's class is a superset of the authority required by the callee.

5.4 Type System Model

In this Section a complete formalization of the proposed type system is given. From now on, it will be referred as **AFC** , from *Android Flows Checker* . In AFC , expressions have a type θ, which is a pair made up of a standard Java type τ and a DLM label L.

$$\theta \Leftrightarrow (\tau, L)$$

5.4.1 Definitions

Prior to explaining rules for typing Java statements and expressions, some definitions are required. Please recall that an in-depth definition of subtyping between DLM labels has been given in Subsection 3.5.1.

Definition 5.4.1. (Fully Qualified Types). We define a *reification* function $\lceil x \rceil$ that maps types to their fully qualified form. A fully qualified type is a type whose name has been appended to its package, as in the following examples:

$$\lceil \text{String} \rceil = \text{java.lang.String}$$
$$\lceil \text{Intent} \rceil = \text{android.content.Intent}$$
$$\lceil \text{ArrayList} \rceil = \text{java.util.ArrayList}$$

Definition 5.4.2. (Java Subtyping). We say that τ_1 is a subtype of τ_2, written as $\tau_1 \sqsubseteq_T \tau_2$, iff $\tau_1 = \tau_2$ or τ_1 extends a type τ_3 such that $\tau_3 \sqsubseteq_T \tau_2$.

Definition 5.4.3. (DLM Subtyping). We say that L_1 is a subtype of L_2, written as $L_1 \sqsubseteq_L L_2$, iff L_1 is at most as restrictive as L_2.

Definition 5.4.4. (Subtyping). We say that the pair (τ_1, L_1) is a subtype of (τ_2, L_2), written $(\tau_1, L_1) \sqsubseteq (\tau_2, L_2)$, iff $\tau_1 \sqsubseteq_T \tau_2$ and $L_1 \sqsubseteq_L L_2$.

Java classes form a tree hierarchy where *Object* can be found at its root, being a supertype for every other Java type[3]. An expression whose type is *Object* can be assigned only to recipients with type *Object*. It is formally represented by the *Top Type* $\top_\tau$. The literal *null*, instead, consists in a blank object reference and can thus be assigned to any recipient with any type in the Java class hierarchy. This behaviour is modeled by the *Bottom Type* $\bot_\tau$. The same reasoning can be applied to DLM labels and our system's types, as shown in the following definitions.

Definition 5.4.5. (Top types). We define the following three *Top* types:
$\top_\tau \in T$ *such that* $\forall\ \tau,\ \ \tau \sqsubseteq_T \top_\tau$
$\top_L \in L$ *such that* $\forall\ L,\ \ L \sqsubseteq_L \top_L$
$\top \in \theta$ *such that* $\forall\ \tau, L,\ \ (\tau, L) \sqsubseteq \top$

Definition 5.4.6. (Bottom types). We define the following three *Bottom* types:
$\bot_\tau \in T$ *such that* $\forall\ \tau,\ \ \bot_\tau \sqsubset_T \tau$
$\bot_L \in L$ *such that* $\forall\ L,\ \ \bot_L \sqsubseteq_L L$
$\bot \in \theta$ *such that* $\forall\ \tau, L,\ \ \bot \sqsubseteq (\tau, L)$

Java classes can be seen as a collection of fields, methods, nested inner classes and inner interfaces. They all have unique identifiers within the scope of a class, except for methods which are subject to overloading. Overloaded methods are methods with the same identifier but that differ in number and type of formal parameters. Mind that in Java actual arguments passed to a method do not have to be of the same type of the corresponding parameter, because all its subtypes are accepted. Additionally, overloaded methods might well have different begin or Return Labels. Whilst the tool does not perform most of the standard Java type-checks, it needs to be able to discern which of the possibly several overloaded methods is targeted, in order to verify the legality of a method invocation. For all these reasons, two distinct look-up functions have been defined and implemented in AFC . For further details about them, please refer to [5].

Definition 5.4.7. (Lookup). Let τ be a Java type, $\tilde{\sigma}$ be its direct supertype and $\tau \downarrow x$ be the look-up members function defined as follows:

$$\tau \downarrow x = \begin{cases} \epsilon & \tau = \top \\ \theta_x & (x : \theta_x) \in \tau \\ \tilde{\sigma} \downarrow x & x \notin \tau \wedge \tau \neq \top \end{cases}$$

where ϵ represents a fail of the Lookup function and thus an unexpected error[4].

[3]Please remember that even built-in types such as int and boolean do have wrappers (namely java.lang.Integer and java.lang.Boolean) in order to represent them in the Java class hierarchy.

[4]Such an error should never occur, because it is assumed that the source code has been already type-checked by Java. Hence, all member accesses are supposed to be correct.

Definition 5.4.8. (Distance Between Methods). Let m be a method with type $(\tau_1 \times \tau_2 \times ... \times \tau_n \to \tau_0)$ and m' be a method with type $(\tau'_1 \times \tau'_2 \times ... \times \tau'_n \to \tau'_0)$. Then $\Delta(m,\ m')$ is defined as:

$$\sum_{i=1}^{n} \delta(\tau_i,\ \tau'_i)$$

where $\delta(\tau, \tau')$ is the type distance function as described in [5], Section 4.3.

Definition 5.4.9. (Resolve Overloaded). Let τ be a Java type, m be a method with type $(\tau_1 \times \tau_2 \times ... \times \tau_n \to \tau_0)$ and $\mathbb{M}$ be the set of all methods with the same identifier of m. The resolve overloaded methods $\tau \Downarrow m(\tau_1, \tau_2, ...\tau_n)$ is the function that returns a method $m' : (\tau_1 \times \tau_2 \times ... \times \tau_n \to \tau_0)$ such that:

$$\forall \mu \in \mathbb{M} \nexists\ \mu,\ \Delta(\mu,\ m) < \Delta(m', m)$$

5.4.2 Types Environment

Typing rules are defined with the aid of an *environment*, that is a map from identifiers to types. For the sake of simplicity it is assumed that all Java types are fully qualified, as the formalization of the process of qualifying types within imported paths and external libraries is an arduous task that would add little to nothing to the model, but that of course has been implemented in the tool. The environment is represented with the letter Γ. In Table 5.2 is possible to find the syntax used to describe the context in which statements and exceptions are typed.

Γ	**Type Environment**, maps identifier into types $x \mapsto (\tau, L)$
θ	**Type**, a pair made up of a Java type and a label $\theta : (\tau, L)$
$\Gamma \vdash E : \theta$	**Judgment**, Expression E has type θ when judged with environment Γ
$\Gamma[x]$	**Lookup**, looks for the identifier x within environment Γ
$\Gamma, (x := \theta)$	**Binding**, binds the identifier x to type (τ, L) in Γ
$\tilde{pc}$	**Program counter label**, its current value
$\tilde{tl}$	**Termination label**, its current value
$\tilde{\tau}$	**Type of *this***, the class in which the expression/statement is found
$\tilde{\tau}[\tilde{m}]$	**Current method** that is being type-checked
Σ	**Static Authority Map**, maps identifier to sets of principals
$\Sigma[x]$	**Static Authority**, sets of principals for whom method or class x *acts-for*
$\mathcal{G}$	**Go-to Environment**, maps Java label to DLM labels
$\mathcal{G}[\mathcal{L}]$	**Target PC Label**, value of $\tilde{pc}$ before the statement pointed by $\mathcal{L}$
$\Gamma \vdash \tau_1 \sqsubseteq_\tau \tau_2$	**Java subtyping**, Java type τ_1 is a subtype of τ_2 in environment Γ
$\Gamma \vdash L_1 \sqsubseteq_L L_2$	**DLM Subtyping**, label L_1 is at most as restrictive as L_2 in Γ
$\Gamma \vdash \theta_1 \sqsubseteq \theta_2$	**Subtyping**, pair $(\tau, L)_1$ is a subtype of $(\tau, L)_2$ in environment Γ

Table 5.2: Environment and judgments.

5.5 Typing Rules

An Android application, like any other Java application, is made up of a list of compilation units, every of which is a single Java source code file. A compilation unit consists in a set of classes and interfaces. Classes and interfaces are collections of *fields* (or attributes) and *methods*. Methods are entities that contain a list of statements. Statements may be calls to functions, binary operators, unary operators, assignments, expressions and so on. Finally, an expression is either an atomic operation or a combination of multiple expressions that is evaluated to a certain value, whose type (or at least one of its supertypes) is known at compile time. In Table 5.3 it is possible to see a simplified representation of the Java *Abstract Syntax Tree*[5]. Please mind that while some Java constructs - such as the type cast expression - do not appear in this Section, they have been fully handled in the actual implementation. They have been left out to not burden the formalization with notions that are not so meaningful for our purposes.

5.5.1 Type-Checking Java Expressions

Expressions are all those sort of Java constructs that may produce a new value as a result for their computation. Their typing rules do not produce side-effects, which

[5]A tree-like data structure built by the compiler to represent the structure of the application.

τ	:=		**type**
	|	boolean | char | ...	**built-in type**
	|	$\top$	**Top type**
	|	T	**class type**
	|	$\tau[]$	**array of τ**
	|	$\tau < \tau_1, ...\tau_n >$	**application of type args**

e	:=		**expression**
	|	ℓ	**literal**
	|	x	**identifier**
	|	$\tilde{\tau}$	**this**
	|	e_1 ? e_2 | e_3	**conditional expression**
	|	op e	**unary operation**
	|	e_1 op e_2	**binary operation**
	|	$e_1[e_2]$	**array subscript**
	|	new $\tau[]\{e_1, ...e_n\}$	**array construction**
	|	$e.x$	**select**
	|	$e.x(e_1, e_2, \ ..., \ e_n)$	**method invocation**
	|	new $\tau(e_1, e_2, \ ..., \ e_n)$	**object construction**

s	:=		**statement**
	|	;	**empty statement**
	|	s; s	**sequence of statements**
	|	e	**statement expression**
	|	$\tau \ x \ = \ e$	**variable declaration**
	|	$x = e$	**assignment**
	|	return e	**return statement**
	|	if e_c then s_t else s_e	**if statement**
	|	while e do s	**while loop**
	|	do s while e	**do while loop**
	|	for $s_1; e; s_2$ do s_b	**for loop**
	|	for $\tau \ \ x \leftarrow e$ do s	**for-each loop**
	|	switch $e$case$_1 \rightarrow s_1; ...$case$_n \rightarrow s_n;$ default $\rightarrow s_d$	**switch statement**
	|	break $\mathcal{L}$	**break statement**
	|	continue $\mathcal{L}$	**continue statement**
	|	this$(e_1, e_2, \ ..., \ e_n)$	**alternate constructor**
	|	super$(e_1, e_2, \ ..., \ e_n)$	**supertype constructor**
	|	$\mathcal{L} \rightarrow s$	**labeled statement**

Table 5.3: Simplified AST representation of Java types, expressions and statements.

are modifications of the context in which expressions are given a type. Instead, the few expressions that need to temporarily alter the context do it so by a technique called *shadowing*, that consists in hiding the value bound to an identifier by creating a new bind between the same identifier and a new value. As a matter of fact, if the environment contains two bindings with the same identifier, it always give precedence to the newer one - which hides, or *shadows*, the older value.

Literals Amongst all the rules to type Java expressions, the one regarding literals (ℓ) is by far the easiest one. Literals are the representation of a fixed value, such as the number *231* or *"this is a literal string"*. Every kind of literal is syntactically different to literals of a different type, thus it is possible to identify the type of literal even if they do not carry along any information of that kind. Being generated on-the-fly, they do not belong to any principal, but they can still create implicit flows. Therefore, the type of a literal is the pair composed of its Java type and the $\tilde{pc}$.

LITERAL

$$\frac{\text{true}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \ell : (\tau, \tilde{pc})}$$

Identifiers An identifier (x) can be used both to refer to a local variable or to a class field. In order to access to a field with a lone identifier, it must have been declared inside the same class or in any of its superclasses. The type of an identifier is the same of the variable or field it refers to, which is obtained with a simple access to the environment Γ.

IDENTIFIER

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash x : (\tau, L) \in \Gamma}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash x : (\tau, L)}$$

Conditional Expressions. Conditional expressions (e_0 ? e_1 | e_2) work in a way much similar to the ubiquitous *if-then-else* statement, except for the fact that they return a value, instead of executing a statement. As a matter of fact, a run-time test is performed on the expression e_0, to decide whether to return expression e_1 or e_2 as the result of the computation. For what regards type-checking, first of all the label of e_0 is typed and then joined with the program counter label. Then follows the judgment of both e_1 and e_2, whose labels are computed with the previously updated

$\tilde{pc}$. At last, the resulting type of the conditional expression is the join of L_1 and L_2. Note that in this latter join operation, the program counter label does not appear because it has been implicitly included in the previous evaluation of L_1 and L_2.

Conditional Expression

$$\frac{\begin{array}{l}\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_0 : (\text{boolean}, L_0) \\ \tilde{\tau}; \tilde{pc} \sqcup L_0; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 : (\tau_1, L_1' := L_1 \sqcup \tilde{pc}) \\ \tilde{\tau}; \tilde{pc} \sqcup L_0; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_2 : (\tau_2, L_2' := L_2 \sqcup \tilde{pc})\end{array}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash (e_0 \ ?\ e_1 \mid e_2) : (\tau, L_1' \sqcup L_2')}$$

where τ is τ_2 if $\tau_1 \sqsubset_\tau \tau_2$, τ_1 otherwise.

Arithmetic, Boolean and String Operations. The type of an unary operation is, trivially, the type of its operand. Binary operations, instead, are processed by judging the left-hand side expression first, followed by the right-hand expression. Once the label of both expressions is known, they are joined to produce the resulting label.

Unary Operation

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau, L)}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash (\text{unop}\ e) : (\tau, L)}$$

Binary Operation

$$\frac{\begin{array}{l}\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 : (\tau, L_1) \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_2 : (\tau, L_2)\end{array}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash (e_1\ \text{binop}\ e_2) : (\tau, L_1 \sqcup L_2)}$$

Array Subscripts. The first step to type-check the access to an array element ($e_1[e_2]$) is to judge the expression e_1, which returns the label of the base type of the array. Follows the typing of e_2, which gives the label of the index. Since it could be possible to gain some knowledge on which element was accessed by knowing the index (and viceversa), both labels are joined to produce the type label of the element itself.

ARRAY SUBSCRIPT

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e_1 : (\tau_1, L_1) \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e_2 : (\text{int}, L_2)}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash (e_1[e_2]) : (\tau_1, L_1 \sqcup L_2)}$$

Array Constructors. A simple array constructor (new $\tau[n : int]$), returns an array n elements long and is typed to $(\tau[], \tilde{pc})$. If it is also provided with a list of initializer expressions (i.e., new $\tau[]\{e_1, e_2, ...e_n\}$), then it must be ensured that none of these labels reaches a destination with a less restrictive label. If, as an example, an initializer expression e_{top} is labeled with the Top Label, then the whole array must be typed with the Top Label, otherwise it could be assigned to a less restrictive destination, losing track of the high security requirements of e_{top}. Hence, array constructors are typed with the join of the program counter label and the labels of all the initializer expressions.

ARRAY CONSTRUCTOR

$$\frac{\forall i \in [1, n], \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e_i : (\tau_i, L_i)}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash (\text{new } \tau_0[]\{e_1, e_2, ...e_n\}) : (\tau_0, \tilde{pc} \sqcup L_1 \sqcup L_2 \sqcup ... \sqcup L_n)}$$

Selects. The label for a field access $(e.x)$ is the join of the label of the container object with the declared label for the field that is to be accessed. Even if it could be deemed as strange, it is important to take in account the label of the container object. As a matter of fact, even though x could have a much less restrictive label than e, it is still part of a sensitive object and it must be treated as such. Public fields inside confidential object can still be accessed through declassification.

SELECT

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e : (\tau_e, L_e) \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash x : (\tau_x, L_x) = e \downarrow x}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash (e.x) : (\tau_x, L_e \sqcup L_x)}$$

Method Invocations. Type-checking a method call $(e_o.m(e_1, e_2, ...e_n))$ requires multiple stages, starting by typing the expression e together with the arguments $(e_1, e_2, ...e_n)$. Their types are thus learnt and used to resolve method overloading, providing information about the , the *parameter labels* and the Return Label. Not

all method invocations are legal, though, as all argument labels have to be no more restrictive than the corresponding parameter labels. Additionally, the $\tilde{pc}$ at the call site must be no more restrictive than the Begin Label (L_{BG}). At last, it must be ensured that the caller (c) is able to provide the required authority to the callee (m). It is so if the union of c's authority set and the authority set of m's class is a superset of the required authority set of m. Legal invocations are then typed as the Java type declared in the method signature and the Return Label joined with the label of e_o.

Method Invocation

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc_0};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e_o : (\tau_o, L_o) \\ \tilde{\tau};\tilde{pc_1} = \tilde{pc_0} \sqcup L_o;\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash m_{res} = e \Downarrow (m : (\tau'_1, L'_1) \times ... \times (\tau'_n, L'_n)) \\ \tilde{\tau};\tilde{pc_1};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash m_{res}\langle L_{BG}, L_{RT}\rangle : (\tau_1, L_1) \times ... \times (\tau_n, L_n) \to (\tau_R, L_{RT}) \\ \forall i \in [1, n], \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash L'_i \sqsubseteq_L L_i \\ \tilde{\tau};\tilde{pc_1};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \tilde{pc} \sqsubseteq_L L_{BG} \\ \tilde{\tau};\tilde{pc_1};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \Sigma[m_{res}] \subseteq \Sigma \cup \Sigma[\tau_o]\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash (e_o.m(e_1, e_2, ...e_n)) : (\tau_R, L_R \sqcup L_o)}$$

Constructor Invocations. Constructor invocations (new $T(e_1, e_2, ...e_n)$) are much similar to static methods, and as such are checked. However, constructors do not have a declared Java return type and thus no Return Label either. Instead, they are typed with the join of the program counter label with the labels of all their arguments.

Constructor Invocation

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash c = \tau_T \Downarrow (T : (\tau'_1, L'_1) \times ... \times (\tau'_n, L'_n)) \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash c\langle L_{BG}\rangle : (\tau_1, L_1) \times ... \times (\tau_n, L_n) \to (\tau_T, \tilde{pc} \sqcup L_1 \sqcup ... \sqcup L_n) \\ \forall i \in [1, n], \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash L'_i \sqsubseteq_L L_i \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \tilde{pc} \sqsubseteq_L L_{BG} \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \Sigma[c] \subseteq \Sigma \cup \Sigma[\tau_T]\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash (\text{new } T(e_1, e_2, ...e_n)) : (\tau_T, \tilde{pc} \sqcup L_1 \sqcup ... \sqcup L_n}$$

This. The expression *this* refers to the instance of the class that contains the field or the method that is being evaluated. As such it is not possible to know its label statically, because it depends on the statement or the expression that generated it. However, remember that before type-checking a call, the Method Invocations rule joins the program counter label of the caller with the label of the recipient object.

Only if this updated label is at most as restrictive as the *Begin Label* of the method, the call is deemed as legal. For this reason it can asserted that the label of *this* is bounded above by the *Begin Label*. Therefore it is safe to round the label of *this* up to the *Begin Label* of the current method.

THIS

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tilde{\tau}[\tilde{m}]\langle L_{BG}, L_{RT}\rangle : (\tau_1, L_1) \times ... \times (\tau_n, L_n) \to (\tau_T, \tilde{pc} \sqcup L_1 \sqcup ... \sqcup L_n)}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \text{this} : (\tilde{\tau}, L_{BG})}$$

5.5.2 Type-Checking Java Statements

Unlike expressions, Java statements do not return a value, and as such they do not possess a type in its own right. Instead, they are either considered *well-typed* or *ill-typed*. A well-typed statement is a statement that respects all typing rules and does not generate any illegal information flow. On the contrary, ill-typed statements do break at least one type rule and therefore are reported as type errors. Please mind that some statements do produce side-effects. This behaviour is represented by the symbol $\triangleright$, meaning that the context for the new statement will be modified with all the environments or values that appear at the right of the $\triangleright$ symbol.

Empty Statements The rule for empty statements (;) is indeed the most trivial one. An empty statement is, as the name suggests, a statement that does nothing. Hence, it is always considered to be well-typed.

EMPTY STATEMENT

$$\frac{\text{true}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash ;}$$

Statement Expressions. Statement expressions (e) are nothing more than expressions whose return value is discarded. With most expressions, as for example with literals and identifiers, using them as statement does not make any sense, because it is just a waste of computing time. Some of them though, such as method invocations, do have side-effects[6] and thus are commonly used disguised as statements. Their type-checking consists in checking expression itself. If the expression

[6]Side-Effects for Java Expressions and Statements must be not confused with side-effects for typing rules. In the former case, a side-effect is an alteration of the application's state during its execution (e.g., modifying the value of the field of a class), in the latter one it is a change of the state of the type-checker itself at compile time.

presents some illegal flows then the statement is considered to be ill-typed, otherwise it is regarded as well-typed.

STATEMENT EXPRESSION

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau, L)}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e}$$

Variable Declarations. Variable declarations ($\tau \;\; x = e$) are managed differently based on which of the following four cases applies:

- **No developer defined label, no initializer expression.** The variable is added into the environment with a special label ϵ, which is a value that represents an yet unknown value for that variable's label. The first time a value is assigned to the variable, ϵ is replaced with the label of the assigned value. If, instead, a variable with label ϵ is found inside any other expression, it raises a type error. This behaviour do not limit the expressivity of the language, as the Java compiler rejects code with uninitialized variables.

DECL (NO LABEL, NO EXPR)

$$\frac{\begin{array}{l}\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tau \;\; x : (\tau, \epsilon) \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma, (x := (\tau, \epsilon)) \vdash st\end{array}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tau \;\; x; st}$$

- **No developer defined label, with an initializer expression.** The initializer expression is evaluated to obtain its label L. If the initializer does not cause any illegal flow then the variable is added to the environment with label L and the statement is regarded as well-typed. Otherwise, it is reported as ill-typed.

DECL (NO LABEL, EXPR)

$$\frac{\begin{array}{l}\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau_e, L_e) \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash x : (\tau_x, \epsilon) \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma, (x := (\tau_x, L_e)) \vdash st\end{array}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tau \;\; x = e; st}$$

- **Developer defined label, no initializer expression.** The variable is added to the environment with the label supplied by the developer. This declaration

is always considered to be well-typed, as long as there are not syntax errors in the label definition.

DECL (LABEL, NO EXPR)

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash x : (\tau, L) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma, (x := (\tau, L)) \vdash st}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tau \;\; x; st}$$

- **Developer defined label, with an initializer expression.** The initializer expression is typed (L_{init}) and checked against the label supplied by the developer L_{dev}. If L_{init} is no more restrictive than L_{dev} the declaration is considered well-typed, in any other case it is considered ill-typed.

DECL (LABEL, EXPR)

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau_e, L_e) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash x : (\tau_x, L_x) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash L_e \sqsubseteq_L L_x \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma, (x := (\tau_x, L_x)) \vdash st}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \tau_x \;\; x = e; st}$$

Assignments. Assignments ($e_1 = e_2$) are regarded as well-typed as long as the right-hand expression e_2 joined with $\tilde{pc}$ is no more restrictive than e_1. This is done to prevent implicit flows, as previously explained. Please remind that if e_1 evaluates to an identifer with unknown label ϵ, than the environment must be modified so its label is updated to the one that corresponds to the right-hand expression.

Assignment (Unknown Label)

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 : (\tau_1, \epsilon) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_2 : (\tau_2, L_2)}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 = e_2}$$

Assignment (General Case)

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 : (\tau_1, L_1) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_2 : (\tau_2, L_2) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma L_2 \sqsubseteq_L L_1}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_1 = e_2}$$

Return Statements. A *return* statement (return e) terminates the method in which it is found. It is used to pass the value of its expression as a result for the caller of the method. Its type-checking is straightforward, as is it is enough to test L_e, the label of the expression, against the Return Label of the method L_{RT}. As previously discussed, however, return statements may generate implicit flows, if not handled with care. Hence, the termination label of the context is joined with L_e.

Return Statement

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau_e, L_e) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl} \sqcup L_e; \mathcal{G}; \Sigma; \Gamma \vdash s}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \text{return } e; \text{ s}}$$

If Statements. An *if* statement (if e_c then s_t else s_e) consists in a conditional expression used to decide at run-time whether to execute the *then* branch or the *else* branch. The type-checking proceeds by typing the conditional expression e_c, whose label is used to temporarily increase the program counter label. Follows the checking of s_t and s_e, under the environment with the newly restricted program counter label. An *if* statement is well-typed as long as its conditional expression and its branches are well-typed.

If Statement

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e_c : (\text{boolean}, L_c) \quad \tilde{\tau}; \tilde{pc} \sqcup L_c; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash s_t \quad \tilde{\tau}; \tilde{pc} \sqcup L_c; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash s_e}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \text{if } e_c \text{ then } s_t \text{ else } s_e}$$

While Loops. A *while* loop (while e do s) is a control flow statement that continues to repeat a block of code whilst a certain condition e is met at the beginning of every iteration. As such, it is type-checked similarly to an *if* statement, with the condition e evaluated in order to use its label as program counter label for the following block of statements. Two entries are added to the *Go-to Environment*, both bounded to the value of program counter label immediately before the *while* statement. These represent the target for an eventual *continue* or *break* statement. Note that they are bounded to the same value because $\tilde{pc}$ possesses the same value both before and after the *while* statement. A *while* loop is considered to be well-type as long as both its conditional expression and its block of statements are well-typed.

While Loop

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau_e, L_e) \quad \tilde{\tau}; \tilde{pc} \sqcup L_e; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash s \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}, (\mathcal{L}_{cont} := \tilde{pc}), (\mathcal{L}_{brk} := \tilde{pc}); \Sigma; \Gamma \vdash s_2}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \text{while } e \text{ do } s; s_2}$$

Do-While Loops. A *do-while* loop (do s while e) is a statement that continues to repeat a statement until a certain boolean condition - checked *after* every iteration - becomes false. Unlike with plain *while* loops, type-checking *do-while* loops requires to type-check the block of statements first and the conditional expression after. Therefore, the block of statements is evaluated with the starting $\tilde{pc}$.

Do-While Loop

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash s \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash e : (\tau_e, L_e) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}, (\mathcal{L}_{cont} := \tilde{pc}), (\mathcal{L}_{brk} := \tilde{pc}); \Sigma; \Gamma \vdash s_2}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma \vdash \text{do } s \text{ while } e; s_2}$$

For Loops. A *for* loop (for $s_{init}; e_{cond}; s_{incr}$ do s_b) is a complex statement that continuously executes s until the given boolean condition e_{cond} becomes false. Before

executing this block, though, it performs an initializer statement s_{init}. After every iteration of this code block, an additional statement s_{incr} is executed. The body of the *for* loop is checked with the label of the conditional expression as program counter label. As with all other loops, the *Go-to* environment is enriched with targets for *break* and *continue* statements, and is considered well-typed as long as all the statements and expressions it contains are well-typed.

For Loop

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma_0 \vdash s_{init} \rhd \Gamma_1 \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma_1 \vdash e_{cond} : (\tau_{cond}, L_{cond}) \\ \tilde{\tau};\tilde{pc} \sqcup L_{cond};\tilde{tl};\mathcal{G};\Sigma;\Gamma_1 \ \vdash s_b \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G},(\mathcal{L}_{cont} := \tilde{pc}),(\mathcal{L}_{brk} := \tilde{pc});\Sigma;\Gamma \vdash s_2\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{for } s_{init};e_{cond};s_{incr} \text{ do } s_b;s_2}$$

For-Each Loops. A *for-each* loop statement (for $\tau_x \ \ x \leftarrow e$ do s) is a syntactical sugar that allows to quickly write loops to iterate through a collection of items implementing the *Iterable* interface, such as the built-in *List* object.

For-Each Loop

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma_0 \vdash \tau_x \ \ x : (\tau_x, L_x) \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma_1 = \Gamma_0,(x := (\tau_1, L_1)) \vdash e : (\tau_e, L_e) \\ L_e \sqsubseteq_L L_x \\ \tilde{\tau};\tilde{pc} \sqcup L_e \sqcup L_x;\mathcal{G};\Sigma;\Gamma_1 \ \vdash s \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G},(\mathcal{L}_{cont} := \tilde{pc}),(\mathcal{L}_{brk} := \tilde{pc});\Sigma;\Gamma \vdash s_2\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{for } \tau_x x \leftarrow e \text{ do } s;s_2;}$$

Switch-Case Statements. A *switch* statement (switch e $\text{case}_1 \rightarrow s_1$; ... $\text{case}_n \rightarrow s_n$; default $\rightarrow s_d$;) consists in an expression that is checked against several pre-defined cases and a block of statements. If any of the pre-defined cases matches the conditional expression than the computation jumps to the statement labeled with that *case*, otherwise it will jump directly to the *default* case. The initial program counter label for the *switch* body is the label of the conditional expression. Every time a *case* is encountered the $\tilde{pc}$ is further restricted with the label of the *case* expression. This is done because by knowing that a certain statement has been executed we can learn something about the conditional expression (e.g. if the second case is executed and there is a *break* statement at the end of the first case then we know that the conditional expression e is different from the first case expression).

As always, the *switch* statement is well-typed as long as there are no illegal flows within it.

Switch-Case Statement

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc}_0;\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash e : (\tau_e, L_e) \\ \forall i \in [1,n], \quad \tilde{\tau};\tilde{pc}_i = \tilde{pc}_{i-1} \sqcup L_i;\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash case_i : (\tau_e, L_i) \\ \forall i \in [1,n], \quad \tilde{\tau};\tilde{pc}_i;\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash s_i\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{switch } e\text{case}_1 \rightarrow s_1; ...\text{case}_\text{n} \rightarrow s_n; \text{default} \rightarrow s_d}$$

Break Statements. A break statement (break $\mathcal{L}$) interrupts the normal flow of the computation, exiting from on ore more loops and directly jumping to the statement tagged with the target label $\mathcal{L}$[7]. If a break statement is not provided with a target label, then the statement jumps at the end of the innermost loop. A *break* statement is well-typed if $\tilde{pc}$ at the *target site* is at least as restrictive as the program counter label where the break occurs.

Break Statement

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \ \tilde{pc} \sqsubseteq \mathcal{G}[\mathcal{L}]}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{break } \mathcal{L}}$$

Continue Statements. A *continue* statement (continue $\mathcal{L}$) works in the same manner of a *break* statement, with the only exception that instead of exiting from a loop, it jumps at the beginning of the next iteration of the loop identified by the target label. Nevertheless, it is typed likewise to a *break*.

Continue Statement

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \ \tilde{pc} \sqsubseteq \mathcal{G}[\mathcal{L}]}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{break } \mathcal{L}}$$

Alternate Constructor Invocations. The invocation of an alternate constructor (this$(e_1, e_2, ...e_n)$) allows the developer to define multiple constructors with one or more default values for the arguments to be provided. It is exactly what the name means, i.e. the invocation of a different constructor of the same class inside a

[7]NB: it is *not* an information flow label, but just a tag to univocally identify the statement to be targeted by the break

first constructor. It is type-checked as a statement expression whose expression is a static method call that returns no value.

Alternate Constructor Invocation

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash c = \tilde{\tau} \Downarrow (\text{this} : (\tau'_1, L'_1) \times (\tau'_2, L'_2) \times ... \times (\tau'_n, L'_n)) \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash c\langle L_{BG}\rangle : (\tau_1, L_1) \times ... \times (\tau_n, L_n) \rightarrow (\tilde{\tau}, \tilde{pc} \sqcup L_1 \sqcup ... \sqcup L_n) \\ \forall i \in [1, n], \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash L'_i \sqsubseteq L_i \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \tilde{pc} \sqsubseteq_L L_{BG} \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \Sigma[c] \subseteq \Sigma \cup \Sigma[\tau_{\tilde{\tau}}]\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{this}(e_1, e_2, ...e_n)}$$

Supertype Constructor Invocations. A supertype constructor invocation (super $(e_1, e_2, ...e_n)$) is much similar to an alternate constructor invocation, the lone difference being the fact that the alternate constructor is part of the class that is directly extended by the one in which the actual constructor can be found.

Supertype Constructor Invocation

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash sc = \tilde{\tau} \Downarrow (\text{super} : (\tau'_1, L'_1) \times (\tau'_2, L'_2) \times ... \times (\tau'_n, L'_n)) \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash sc\langle L_{BG}\rangle : (\tau_1, L_1) \times ... \times (\tau_n, L_n) \rightarrow (\tilde{\sigma}, \tilde{pc} \sqcup L_1 \sqcup ... \sqcup L_n) \\ \forall i \in [1, n], \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash L'_i \sqsubseteq L_i \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \tilde{pc} \sqsubseteq_L \langle L_{BG}, L_{RT}\rangle \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \Sigma[c] \subseteq \Sigma \cup \Sigma[\tau_{\tilde{\tau}}]\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \text{this}(e_1, e_2, ...e_n)}$$

Labeled Statements. A labeled statement $(\mathcal{L} \rightarrow s)$ is a statement that has been tagged by a Java label. The type-checking of labeled statements is straightforward, as it is enough to add the target label to the environment Γ and then proceed type-checking the statement s as normal. A labeled statement is well-typed as long as its base statement is well-typed.

Labeled Statement

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash s \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G}, (\mathcal{L} := \tilde{pc});\Sigma;\Gamma \vdash s_2\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma \vdash \mathcal{L} \rightarrow s; s_2}$$

5.5.3 Type-Checking Classes and Methods

The number of checks to be performed on Java classes by an Information Flow type system is fairly low. All things considered, a class is not much more than a container, and containers cannot produce information flows on their own. To type-check a class basically means to type-check all of its components, verifying that no illegal flow is generated within the declaration of its members and inner classes. On the other hand, method declarations require a great deal of attention, as there are multiple subtle aspects that can harm the security of an application. For this reason, methods are bound in the environment with some further information that is here summarized to be seen at a glance:

- **Begin Label**. The label that is used as starting value for the program counter label when type-checking the method. If not supplied, it is the Top Label.
- **Return Label**. The label that represents the required secrecy and integrity on the value returned by the method. If not supplied, it is the join of the label of all its parameters. If there is no parameter, then it is the Bottom Label.
- **Parameter Labels**. Every formal parameter has a label associated with it. By default it is the Top Label.
- **Authority Set**. The set of principals for which the method can *act-for*. If not specified it is $\{\bot\}$, meaning that it *acts-for* the principal *Bottom* alone.

Whenever a method overrides or implements another method with the same signature, its labels require some further constraints, otherwise type polymorphism would enable the generation of uncaught illegal flows. An example of this behaviour can be found in Listing 5.2, where a string whose type is the Top Label is assigned to a variable labeled with the Bottom Label, which is clearly unsafe. This is allowed by the type system because the *takeFirstItem()* is invoked on the object c which is bound in the environment as an instance of the **Container** class. The Return Label of the method in Container is the Bottom Label, therefore the type system deems the assignment as legit. However, thanks to polymorphism, the actual type at run-time of the object c will be **Strongbox**. Alas, the Return Label of *takeFirstItem()* found in Strongbox is the Top Label. Hence, if allowed to run, this application would leak sensitive information, breaking the type system. To prevent such mishaps the following restrictions are enforced:

- **The Return Label** of the overriding (subtype's) method must be at *most* as restrictive as the Return Label of the overridden (supertype's) method $\rightarrow L_{sub} \sqsubseteq_L L_{sup}$.

Listing 5.2: Example of an illegal flow due to polymorphism

```
1
2  class Main{
3      (int, {Bottom: ; Top <− }) m(){
4          Container c = new Strongbox();
5          (Object, {Bottom: ; Top <− }) item = c.takeFirstItem();
6      }
7  }
8
9  class Container{
10     private Object[] items = {"first", "second", "third"};
11     (Object, {Bottom: ; Top <− }) takeFirstItem(){
12         return items[0];
13     }
14 }
15
16 class Strongbox extends Container{
17     @Override
18     (Object, {Top: ; Bottom <− }) takeFirstItem(){
19         return items[0];
20     }
21 }
```

- **The Begin Label** of the overriding method must be at *least* as restrictive as the Begin Label of the overridden method $\rightarrow L_{sub} \sqsupseteq_L L_{sup}$
- **All Parameter Labels** of the overriding method must be at *least* as restrictive as the parameter labels of the overridden method $\rightarrow L_{sub} \sqsupseteq_L L_{sup}$

5.6 Tackling Android Challenges

In Section 4.4 we have seen that a standard Information Flow type system cannot cope with Android applications, as there is a number of issues that require a system specifically tailored for them. This section describes how the previously general model is geared to deal with all such challenges, detailing how they are tackled one-by-one. However, the first step is to introduce a technique that is used to achieve the above-mentioned goal.

5.6.1 Partial Evaluation

The actual implementation of AFC is not a stand-alone tool, but is built on top of a larger framework devised to statically check Android applications, called

Lintent. Lintent features a very useful, albeit complex, technique called **Partial Evaluation**. It consists in a sort of data-flow analysis that keeps track of the type and, when possible, the *run-time* value of variables.

To exploit Partial Evaluation, the environment Γ must be enriched. Instead of mapping identifiers into a couple (τ, L), they are now mapped into triples with form (τ, L, v), that is made up of a Java type, a DLM label and a value. While the type-checker sifts through the whole source code, it binds every field or variable Γ to three aforementioned data. Every time that an assignment or any other side-effect modifies the value bound to an identifier, the environment is updated accordingly. In this way it is possible to statically compute the value that most expressions will have at run-time. Important as it is for our system, Partial Evaluation is a complex subject that would make this work drift away from its original purpose. As such, the complete set of rules for Partial Evaluation, along with further details about its implementation, can be found in [5], Chapter 4.

Definition 5.6.1. (Partial Evaluation). We define as $\Gamma \vdash x \rightsquigarrow v$ the application of the Partial Evaluation rules, where x is an identifier, an expression or a statement and v is the value obtained through Partial Evaluation.

5.6.2 Handling Intents

One of the most crucial challenges is described in Section 4.4.5, and regards the necessity to handle both incoming and outgoing Intents correctly. As an example, it is necessary to understand which Component is the recipient of an Intent in order to type-check the corresponding *onCreate()*. At any moment, an Intent can have one of the three following possible kind of recipients:

- **Explicit**. Explicit Intents are Intents that have been given the Class object of their intended target. They are the most common form of communication between components belonging to the same application or between applications of the same developer.
- **Implicit**. Implicit Intents do not have a pre-defined receiver, instead they are given a string that represents the action to be performed. They might also have other sorts of filters such as to which category of applications the recipient should be part of.

- **Undefined**. Intents can be built with a default constructor that requires no arguments. However, it is not possible for an Intent to understand which Component should be its recipient without any specific information provided by the programmer. Thus, any Intent built in this way is temporarily lacking of a recipient, which is set later.

The receiving component of an Intent can change during the execution of the application. To be more accurate, the developer can set the receiving component of an existing Intent with the following three methods: *setComponent(), setClass()* or *setClassName()*. This also allows to transform Implicit and Undefined Intents into Explicit ones. Implicit Intents are indeed a big challenge, as resolving their action string does not allow to univocally identify which Component will be their run-time target. As a matter of fact, their resolution depends entirely on the state of the end-user device, which cannot be predicted at compile time. To avoid making things overly complicated, the rest of this chapter will deal with *Explicit* Intents only, leaving Implicit ones for later. To keep track of all information about Intents, we start by defining a new environment represented with the Greek letter Xi (Ξ).

Definition 5.6.2. (Components Environment). Let $\mathcal{C}$ be an entity known as a component, then we define as Components Environment the environment Ξ that maps fully qualified type names into components.

$$\Xi \vdash T : \mathcal{C}$$

The environment Ξ is used to relate the fully qualified name of a class, that extends either a Service or an Activity, to an object that describes the Component itself, based on the Intents that it sends or receives. Hence, this Component entity is defined as follows:

Definition 5.6.3. (Component) A component $\mathcal{C}$ is defined as the record[8] that contains the following four sets.

$$\mathcal{C} : \mathcal{I}_{\mathrm{R}} \times \mathcal{O}_{\mathrm{R}} \times \mathcal{I}_{\mathrm{A}} \times \mathcal{O}_{\mathrm{A}}$$

where:

- $\mathcal{I}_{\mathrm{R}}$ are the ***I**ncoming **R**equests*, that is the set of all Intents that are obtained from invoking the *getIntent()* method;

[8] A tuple, that is an ordered sequence of values.

- $\mathcal{O}_{\mathrm{R}}$ are the ***O**utgoing **R**equests*, that is the set of all Intents that are sent with a *startActivity()*, *startActivityFromResult()* or *startService()*;
- $\mathcal{I}_{\mathrm{A}}$ are the ***I**ncoming **A**nswers*, that is the set of all Intents reconstructed inside the *onActivityResult()* method.
- $\mathcal{O}_{\mathrm{A}}$ are the ***O**utgoing **A**nswers*, that is the set of all Intents sent by invoking the *setResult()* method.

The previous definitions allow the type-checker to keep track of how an Activity or a Service communicates, which is indeed compulsory to effectively monitor its information flows. The last entity that needs to be formally defined is the Intent. Please mind that all Intents are modeled in the same way, it does not matter whether they represent an outgoing or an incoming request.

Definition 5.6.4. (Intent) Let an Intent i be a record that contains a recipient r and a list of triples. The recipient r can be either a fully qualified type T or a special blank type ϵ. Any item of the list of triples is made up of a String key k, a Java type τ and a label L.

$$i : r \times (k_1, \tau_1, L_1) \times ... \times (k_n, \tau_n, L_n)$$

Finally, a formal definition of the subtype relationship between Intents is required. For mere convenience we assume that, prior to comparing them, Intents are ordered in a way such that any key with the same value that they possess is at the same position in both lists. As an example, if both Intents have a key that evaluates to the string *"aKey"*, then the triple that contains this key must be the first (or the second, or the third and so on) of the list for both Intents. It cannot happen that it is the first of the list for one Intent and the second of the list for the other one.

Definition 5.6.5. (Intents Subtyping) Let $i_1 : (r_{i_1} \times (k_{i_1,1}, \tau_{i_1,1}, L_{i_1,1}) \times ... \times (k_{i_1,n}, \tau_{i_1,n}, L_{i_1,n}))$ and $i_2 : (r_{i_2} \times (k_{i_2,1}, \tau_{i_2,1}, L_{i_2,1}) \times ... \times (k_{i_2,m}, \tau_{i_2,m}, L_{i_2,m}))$ be two Intents. The Intent i_1 is a subtype ($\sqsubseteq_i$) of the Intent i_2 if and only if all the following relations hold:

$$\begin{aligned} m &\geq n \\ k_{i_1,j} &= k_{1_2,f}, \quad \forall j \in (1,n) \\ \tau_{i_1,j} &\sqsubseteq_\tau \tau_{i_2,j}, \quad \forall j \in (1,n) \\ L_{i_1,j} &\sqsubseteq_L L_{i_2,j}, \quad \forall j \in (1,n) \end{aligned}$$

5.6.3 Tracking Key-Value Pairs

Intents' senders and receivers are not the only information that needs to be tracked. Intents and Bundles are dictionaries, and as such are used to store data. It is critical for the type system to be able to retrace the label of those data, otherwise they could be exploited to launder restrictive labels into more permissive ones. Every time that an Intent is created, fetched or accessed for putting or getting extras, the type checker updates the Component objects of the sender and the recipient that are affected by that Intent. If, as an example, a *i1.putExtra()* is performed, then a triple is added to the list of triples of Intent *i1*.

PutExtra

$$\frac{\begin{array}{l}\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash \tilde{\tau} : T \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash \Xi_0[T] : \mathcal{C} \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash i : (r \times (k_1,\tau_1,L_1) .. \times (k_n,\tau_n,L_n)) \quad \in \Xi_0 \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash k : \text{String} \ \rightsquigarrow v_k \\ \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash e : (\tau_e, L_e)\end{array}}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\Xi_0 \vdash i.putExtra(k,e) \rhd \Xi_0[i := i,(v_k,\tau_e,L_e)]}$$

For what concerns retrieving data, the situation is very similar. Java does not allow to overload methods based on their returned type, hence it is not possible to get data from Intents by calling a generic *getExtra()*. Instead they have a plentiful list of methods whose name contains the type (e.g., *getIntExtra()*). It can happen that the type system has yet to type-check the component that sent the corresponding Intent, as such it is not always possible to be able to access to the label of the corresponding put value. However, all these methods require to be given a default value, that is used if the key is not found inside the dictionary. Hence, the *getExtra* is considered to return a value labelled with the label of this default value. To prevent illegal flows, this latter label must be at least as restrictive as the label of the value that as been put in the Intent. To ensure that this is the case, the incoming Intent is given a new triple with this default label, that will be checked later against the corresponding outgoing Intent.

GetIntExtra

$$\frac{\begin{array}{l}\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash \tilde{\tau} : T \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash \Xi_0[T] : \mathcal{C} \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash i : (r \times (k_1, \tau_1, L_1) .. \times (k_n, \tau_n, L_n)) \quad \in \Xi_0 \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash k : \text{String} \ \rightsquigarrow v_k \\ \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash d : (\tau_d, L_d)\end{array}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi_0 \vdash i.getIntExtra(k, d) \rhd \Xi_0[i := i, (v_k, \tau_d, L_d)]}$$

Once the type system has checked the entire application, it has to verify that the label of all default values used in the *getExtras* were at least as restrictive as those used in the corresponding *putExtras*. This is done by verifying all entries of Ξ one at a time. For every Outgoing Intent i_o there must be at least one Incoming Intent i_i such that $i_o \sqsubseteq_i i_i$. Otherwise a type error is reported. Please remember that at the moment we are dealing with Explicit Intent only, so it is always possible to correctly identify the recipient of an Outgoing Request.

Bundle objects can also be used to launder labels, if not monitored accordingly. They are dictionaries that work in a much similar fashion to Intents, but they are supposed to store intra-component data, and as such they do not possess a recipient. They are populated inside the *onSaveInstanceState()* method of the Activity. Overriding the method is optional, so the first thing that AFC does is to verify whether it has been overridden or not. If the answer is negative, then type-checking proceeds as normal, because it means that the developer did not customize the instance state Bundle. Otherwise, it performs the type-checking of the *onSaveInstanceState()* first, after it continues with the usual ordering.

Definition 5.6.6. (Bundle Set) Let b be a Bundle populated inside a *onSaveInstanceState()*. Then, its representation in the formal model, is referred as $\mathcal{B}$ and is the set of triples (k_i, τ_i, L_i) that represent the keys, Java types and labels of data put inside a b.

$$\mathcal{B} = \{p | p : (k_i, \tau_i, L_i) \in b\}$$

When it skims through the statements of the *onSaveInstanceState()* method, the type checker populates $\mathcal{B}$ every time that encounters a *put*. This allows to keep track of all the labels, that can be accessed from every part of the application by

accessing the previously defined set. Should a default value be supplied to the *get* function, then the returning label would be the join of the label found within the set with the label of the default value. $\mathcal{B}$ becomes thus a part of the type-checking context.

PUT

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash k : \text{String} \rightsquigarrow v_k \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash e : (\tau_e, L_e)}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash b.put(k, e) \rhd \mathcal{B} = \mathcal{B} \cup \{(v_k, \tau_e, L_e)\}}$$

GETINT

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash k : \text{String} \rightsquigarrow v_k \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash d : (\tau_d, L_d) \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash (v_k, \tau_k, L_k) \in \mathcal{B}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \Xi; \mathcal{B} \vdash b.get(k, d) : (\tau_k, L_k)}$$

5.6.4 Creating an Activity

The problem described in Section 4.4.1 consists in devising a mechanism that recognizes the unusual Android control flow whenever it has to start new Components. The code to be type-checked against a *startActivity(intent)*, as a matter of fact, is not to be found inside the method obtained with the usual *resolve overloaded* function ($this \Downarrow startActivity(intent)$). The tool, instead, has to type-check a call to the *onCreate()* method of the activity pointed out by the argument Intent. In Section 5.6.2 it has been shown how to discern which Activity is the target of a given Intent, therefore it is enough to define a limited number of rules that recognize a well-known syntactical pattern and treat those method calls as special cases. This behaviour is justified by the fact that the number of methods with such an uncommon behaviour is reasonably low.

STARTACTIVITY

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \tilde{\tau} : T \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \Xi[T] : \mathcal{C} \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash e_{intent} \rightsquigarrow (r \times (k_1,\tau_1,L_1)..\times(k_n,\tau_n,L_n)) \quad \in \Xi \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash r.onCreate()}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash startActivity(e_{intent})}$$

STARTACTIVITYFORRESULT

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \tilde{\tau} : T \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \Xi[T] : \mathcal{C} \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash e_{intent} \rightsquigarrow (r \times (k_1,\tau_1,L_1)..\times(k_n,\tau_n,L_n)) \quad \in \Xi \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash r.onCreate()}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash startActivityForResult(e_{intent},e_{code})}$$

STARTSERVICE

$$\frac{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \tilde{\tau} : T \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash \Xi[T] : \mathcal{C} \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash e_{intent} \rightsquigarrow (r \times (k_1,\tau_1,L_1)..\times(k_n,\tau_n,L_n)) \quad \in \Xi \quad \tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash r.onStartCommand()}{\tilde{\tau};\tilde{pc};\tilde{tl};\mathcal{G};\Sigma;\Gamma;\mathcal{B};\Xi \vdash startService(e_{intent})}$$

The unusual Android control flow is designed to automatically invoke all life-cycle callbacks, once a Component is started. For Information Flow purposes this behaviour can be approximated by adding, at the end of every life-cycle callback, an invocation to the following method. Two consecutive callbacks with different labels can lead to implicit flows, as the program counter label at the end of a method could be higher than the Begin Label of the following. To prevent this from happening, the proposed type checker requires Android life-cycle callbacks to be labelled as the

onCreate(). Moreover the *onCreate()* callback is the only one allowed to be labelled, from the moment that it is the only one supposed to be invoked directly (through a *startActivity()*.

5.6.5 Returning Results

Likewise to every other data communication in Android, results are returned through Intents. The whole process is made up of three steps. In first place, the caller activity builds an Intent and sends it with an invocation to *startActivityForResult()*, with a string that specifies the requested action and a code that is used to discern between other eventual requests. Second, the callee receives the Intent, performs the required computation and builds a new Intent containing the results, which are then sent thanks to a call to the *setResult()* method. At last, the caller is woken on its *onActivityResult()* callback, where it may use the results it asked for. Regrettably, there is no way to understand which is the activity that requested the computation, and as such it is not possible to understand to whom the results are addressed. Hence, it cannot be helped to used the Intent handling mechanism described in Section 5.6.2. For every Incoming Answer $\mathcal{I}_{\mathrm{A}}$, the type checker looks which Activities are recipient for at least one of its Outgoing Requests $\mathcal{O}_{\mathrm{R}}$. Then it tests all Outgoing Answers $\mathcal{O}_{\mathrm{A}}$ of every possible recipient against the original Intent $\mathcal{I}_{\mathrm{A}}$. Amongst all Intents that are subtype of $\mathcal{I}_{\mathrm{A}}$, the closest one is picked as its most likely counterpart. If, otherwise, no subtype Intent is found, then an error is reported.

5.6.6 Terminating an Activity

An Activity that needs to be terminated should invoke the *finish()* method. Despite it does not influence the control flow of the current method, it might prevent subsequent lifecycle callbacks from being executed, as explained in Section 4.4.3. It has been found that, in order to avoid the aforementioned implicit flows, the program counter label at the call-site of a *finish()* invocation, to which we will refer with $\tilde{pc}_{fin}$, must be no more restrictive than the Begin Label of the callback that follows the current method. As a matter of fact, the Begin Label represents a lower bound on all side-effects that a method can generate. So, if a side-effect would be disallowed immediately after the invocation of a *finish()*, it would be even more so inside a method whose lowest possible $\tilde{pc}$ is at least as restrictive as $\tilde{pc}_{fin}$. Remember that in Section 5.6.4 it has been said that all callbacks have the same Begin Label, that is the one of the *onCreate()*. Hence, a *finish()* invocation is considered legal as long as the Begin Label of the *onCreate()* is at least as restrictive as $\tilde{pc}_{fin}$.

FINISH

$$\frac{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \mathcal{B}; \Xi \vdash \tilde{\tau} : T \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \mathcal{B}; \Xi \vdash T.onCreate\langle L_{BG}, L_{RT}\rangle(b) : (Bundle, L_b) \to \text{void} \quad \tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \mathcal{B}; \Xi \vdash \tilde{pc} \sqsubseteq_L L_{BG}}{\tilde{\tau}; \tilde{pc}; \tilde{tl}; \mathcal{G}; \Sigma; \Gamma; \mathcal{B}; \Xi \vdash finish()}$$

5.6.7 Supporting Generics

Every class in Java can be declared with a set of type parameters, that are generic names each used to refer to an unknown Java type. They are a key feature of Java that allows to increase code reusability. Introduced in Java 1.5, they are indeed natively supported by the Android Platform. The typing context described in the previous rules carries along information about the type of *this*, that is the class that is being type-checked. Thus it is possible to access information about the current class from anywhere in the model. It has been shown how a type in AFC is a pair made up of a Java type and a DLM label, therefore each Type Parameter has its own parameter label attached. Parameter labels produce constraints exactly like dynamic labels (Section 5.3.5). However, they differ in the fact that they can be checked statically whenever an object of that type is instanced, instead of performing run-time checks.

5.6.8 Dealing with Undecidable Cases

Alas, there are some cases that cannot be resolved statically. As an example, it is possible for the developer to present the end-user with the choice of which Component should be used as target for its Intent. It is obvious that it is not possible to statically predict user choices by any means. Also Partial Evaluation may fail to compute some keys, as values too can be influenced from user input. To cope with these undecidable situations, two countermeasures are taken. The first is to warn the developer through an Hint message, suggesting him that keys and values essential for inter-component communication should be *final* and *static* values, because any other solution is very likely to introduce bugs. Second, in order to restrict the freedom of the developer as little as possible, they are treated for what they are: undecidable cases for which nothing can be asserted at compile-time. As such, any label that cannot be tracked or inferred due to their undecidability is replaced by the corresponding Default Label. This allows the type system to remain sound, as Default Labels always represent the safest approach possible, preventing any side-effect.

6

Implementation

As previously hinted, AFC has not been implemented as a stand-alone tool, but is part of a larger framework called **Lintent**[39]. Lintent is designed to statically check Android applications, providing developers with a powerful - yet user-friendly - tool that guarantees the quality of their work. In addition to preventing both malicious and accidental information flows, it detects Privilege Escalation attacks[2] and reconstructs Intent types to avoid bugs and run-time errors[5].

Any developer interested in writing applications for the Android Operating System must download and install on its workstation the Android SDK, which provides the necessary API libraries and tools to build and test Android apps. The official *Integrated Development Environment* for Android applications is Eclipse[34], which is a well-known and widely-spread IDE used by millions of code-writers around the whole world. Additionally, developers can install an official plug-in for Eclipse called *ADT Lint*[35], which scans Android project source files for potential bugs and errors. It offers a convenient graphic interface that allows to perform more than a hundred pre-defined checks with a simple mouse click, giving him the possibility to tweak and disable them. Luckily enough, it also gives the possibility to extend the list of checks by writing custom Java code that returns feedbacks to the user after having visited the AST of the application.

The goal of Lintent is to make life easier for Android programmers, helping them to develop better and safer applications, reducing bugs to a minimum. With this idea in mind, we felt it natural to aim our attention to a so widely spread tool in the Android community, believing that the integration of our work in a familiar interface such as ADT Lint would encourage people in adopting and using it. Moreover, the prospect of being able to analyze Java code without the need to write our own parser was a tempting one indeed. At the moment in which this Thesis has been written, work on Lintent was still in progress, but it had already reached a good stage of development, and was available to be downloaded and tested in its core features. The full source code for both Lintent and AFC can be found at [39].

6.1 Lintent Architecture

Despite being an ADT Lint plug-in, the Java code written for Lintent acts only as a front-end interface to communicate with the actual engine, which is an external executable written in F#. We felt that such a complex project would have required too much work and effort, if it were to be developed in an imperative language as Java. On the other hand functional languages offer an higher-level abstraction, lifting programmers from the burden to handle micro-management tasks typical of imperative languages, leaving more time to spend on improving core algorithms. Between the many functional languages available, F# has been chosen because it offers good performances, a rich documentation and one of the best IDE on the market. Figure 6.1 depicts the basic architecture of Lintent, which is composed of two separate blocks that cooperate to produce the desired result.

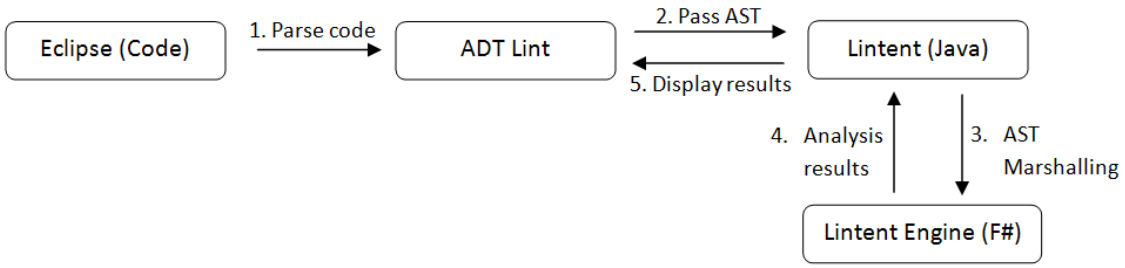

Figure 6.1: Lintent architecture.

Every time that ADT Lint is asked to perform a full code analysis, it starts Lintent after all pre-defined checks are completed. As soon as Lintent starts running, it spawns a new processes for the LintentEngine and opens a *socket* connection with it. For every compilation unit, it is given the root node of the corresponding AST and then begins the visit. Instead of performing checks, though, Lintent marshals the AST and sends it through the socket, where the engine rebuilds it with its own data structure. The F# code is thus ready to start looking for issues, and, every time that it has to report one, it sends a message through the socket, where the Java module is listening. Once the type-checking is completed, the engine shuts down himself, informing its parent process that is now free to return the control to Lint, as soon as all the issues are reported to the programmer.

6.2 Annotations

The majority of Information Flow type systems use a custom programming language to define labels. It is clear that this is not a feasible approach, otherwise it would not be possible to integrate a revised Java language with ADT Lint, because it would detect several syntax errors. Nevertheless, it cannot be helped to have a way to supply labels to the type system, otherwise it would be worthless. The strategy adopted is to exploit Java annotations. Annotations are meta-data that can be attached to class, members and variables declarations. They are meant to provide information for the compiler and do not influence the code behaviour. A typical example is the *@Override* annotation, which tells the compiler that the methods they are attached to are meant to override one of those of the superclass. If this is not the case, the compiler issues an error to the programmer that can immediately solve the problem at compile-time, instead than having to spend much time in debugging a simple, but difficult to find, error.

Java allows to define custom annotations that are ignored by the compiler, yet still parsed in the AST and available to be used by any tool that should be interested in them. Therefore we defined a range of Annotations to supply all the labels needed by Lintent to perform its checks. These, amongst other utilities, are coded inside an auxiliary library that is supplied with Lintent and that must be added to the project of the Android application. Table 6.1 shows the correspondences between DLM labels and the names of our custom annotations. Please notice that for every label there are multiple naming choices: this is done because shorter names (e.g., *L* for *Label*) are indeed faster and more elegant, however they are more likely to have similar names with other custom Annotations, generating confusion. Additionally, longer names allow to immediately understand which DLM label they are referring to, requiring less memorization efforts. We decided to provide the programmer with multiple choices so as to get the best of both worlds.

Label	:=	DlmLabel, Label, L
Begin Label	:=	DlmBeginLabel, DlmBegin, BeginLabel, Begin, B
Return Label	:=	DlmReturnLabel, DlmReturn, ReturnLabel, Return, R
Principal Declaration	:=	DlmPrincipals, Principals, P
Authority Set Declaration	:=	DlmAuthoritySet, DlmAuthority, AuthoritySet, Authority, A
Declassify	:=	DlmDeclassify, Declassify

Table 6.1: Names for AFC annotations.

6.2.1 Grammar

Here it is shown the syntax for AFC programs. It defines both how annotations should be written and where they should be placed. Notice that both terminals and non-terminals inside square brackets are optional, thus replaced by default labels if missing. All constructs followed by an asterisk (*) can be repeated zero or multiple times. The uppercase terminals for labels can have any of the values defined in Table 6.1. For the sake of simplicity the grammar for standard Java constructs is taken for granted, as it would be meaningless for the purposes of this essay.

BRA	:=	(
KET	:=	)
compilationUnit	:=	dlmType
dlmType	:=	dlmClass [dlmType]
	|	dlmInterface [dlmType]
dlmClass	:=	[principalList] [authoritySet] [LABEL] javaClass
dlmInterface	:=	[authoritySet] [LABEL] javaInterface
dlmMethod	:=	[BEGIN] [RETURN] [LABEL] [authoritySet] javaMethod
dlmFieldDeclaration	:=	[LABEL] [DECLASSIFY] javaFieldDeclaration
dlmVariableDeclaration	:=	[LABEL] [DECLASSIFY] javaVariableDeclaration
dlmParameter	:=	[LABEL] javaParameter
principalList	:=	PRINCIPALS BRA principal (SEMICOLON principal)* KET
principal	:=	ID [ACTS-FOR idList]
idList	:=	ID (COMMA ID)*
authoritySet	:=	AUTHORITY BRA idList KET

Table 6.2: Grammar for AFC annotations.

6.3 Programming Style

Most F# constructs and data structures have been used throughout the whole program, spanning from active patterns to usual object oriented classes. Amongst all adopted techniques there is one that has seen a wide usage. It is indeed one of the most interesting features of F#, called *Computation Expressions*. Computation Expressions provide a convenient syntax for sequencing computations, using imperative control flow statements to produce pure functional code. The advantages thus enabled are numerous, such as the ability to write more readable, bug-free code while also being a lot more productive. In Lintent they have been used to implement Monads[26], that allow the computation to implicitly carry along the type checker state in an automated fashion. This is achieved through the redefinition of some special methods that are implicitly invoked by *monadic* code. As an example the *let* binding[1] has been redefined so as to take two additional arguments, that are the state of the monad and a function.

```
Bind: ('s->'d*'e) -> ('d->'e->'s) -> M<'s, 'b>
Bind (e, f) =
    fun s ->
        let (r, s') = e s
        f r s'
```

As shown in the example above, the state of the monad is applied to the expression *e*, which computes the value to be assigned to the given identifier. This expression might affect the state itself, which is referred to as *s'*. The function used as second additional parameter, called *f*, contains nothing more than the following statements; in other words it represents rest of the computation to be performed. The method then applies the results of the first statement to the rest of the program, thus propagating the changes to the state performed by statements. This is particularly convenient, as the programmer only has to write *let!* $x = e$, and the compiler will automatically translate it in the aforementioned code. By using monads we have been able to adopt a peculiar style of programming called *Continuation Passing Style*, which is a common paradigm in functional languages, as it allows highly maintainable code with a minimum amount of effort.

[1] A keyword that associates an identifier with a value, which can also be a function.

6.4 State of the Type-Checker

Monads in Lintent are used to carry along information about the type-checking progresses, all saved within a record that is the state. Inside this state it is possible to find all those environments and information that constitute the type-checking context, already shown while formalizing the type system model in Section 5.4. There are also some additional data that do not regard the formalization but were indeed necessary to implement the type checker in its entirety.

```
type [<NoComparison>] state =
    {
    classes          : TJ.class_env
    var_env          : Env.t<id, var_decl>
    comp_env         : Env.t<fqid, componentt>
    bundle           : (var_decl list) * (var_decl list)
    pc_label         : Dlm.label
    principals       : Dlm.principal Set
    hierarchy        : (Dlm.principal * Dlm.principal) list
    static_authority: Dlm.principal Set
    latest_loop_lb   : Dlm.label option
    goto_targets     : (id * Dlm.label) list
    goto_statements : (J.statement * Dlm.label) list
    }
```

- **classes**. An environment that contains all classes in the application;
- **var_env**. The implementation of the Γ environment;
- **comp_env**. The environment for Components, Ξ;
- **pc_label**. The current program counter label;
- **principals**. The list of all principals that have been defined;
- **hierarchy**. The static hierarchy, containing all acts for relationships;
- **static_authority**. The authority set possessed by the current method;
- **latest_loop_lb**. The pc label before the latest loop, to type-check breaks and continues;
- **goto_targets**. The list of all java labels paired with corresponding pc labels;
- **goto_statements**. The list of al goto statements whose label has not been encountered yet.

6.5 AFC Implementation

Android Flows Checker has been implemented with a modular approach, subdividing it in several logical units, each with a different purpose, but all cooperating to reach the common goal. Also known as *divide et impera*, it is a well-known approach to attack complex problems, reducing them into many smaller tasks that are easier to be dealt with singularly. It also offer the undeniable advantage of increasing code maintainability, because should the need to entirely rewrite a single module arise, it is not necessary to modify the rest of the code, as long as the module does not change its external interface. Here it is possible to find a list of all AFC modules, with a brief description of their task and some eventual interesting implementation highlights.

Report. This module is responsible for reporting errors and warnings to Lintent. It forwards error and warning messages to an instance of the *IssueReporter* type, that is an object containing the API to perform bidirectional communication with the Java extension for ADT Lint. The Java module of Lintent listens for messages on the Socket, and whenever it receives errors and warnings it feeds them to the error reporting API of ADT Lint.

Error. The Error module consists in a list of functions to be invoked whenever AFC encounters a standard type error. Based on the source of the error, they require further information on what generated the type error and which is its location on the source code. All errors reported through this module are considered recoverable. An error is deemed as recoverable if it does not abruptly end the type-checking, allowing AFC to continue to look for other errors.

Warning Similar to the Error module, with the main difference being that it reports Warnings and Hints to the programmers. Differently from errors, they do not strictly compromise the security of the application, but are caused by bad programming practices that could cause troubles if not kept under control.

Defaults This tiny module is a collection of default values and labels. As an example, it contains all the default labels defined in Section 5.3.4

Utilities The Utilities module contains several miscellaneous functions that are used throughout the implementation of the proposed type checker. They are not strictly related to the Information Flow domain or the monadic environment, but are still useful to keep the code simple and readable.

Listing 6.1: Code snippets involving Active Patterns

```
let (|DlmMethod|) (m : J.method_signature) =
        let (DlmBeginLabel begin_lb) = m
        let par_lbs = List.map (function (DlmParamLabel l) -> l) m.paramss
        let (DlmReturnLabel par_lbs begin_lb.Value ret_lb) = m
        let (DlmAuthority auths) = m
        in
            DlmMethod (m, begin_lb, ret_lb, par_lbs, auths)


let rec typecheck_class ctx cl =
    ...
    for m in cl.methods do
        do! trap (typecheck_method_or_constructor ctx) m
    ...


let rec typecheck_method_or_constructor ctx (D.DlmMethod (m, begin_lb, ret_lb, par_lbs, auths) =
    ...
```

Detector The Detector module is indeed a keystone for the whole AFC type checker. It contains all *Active Patterns* used during the entire implementation, an awesome feature that allows to recognize and discern between multiple syntactical patterns with a bunch of code lines. In the Listing 6.1 it is possible to find an example of the expressiveness and power of these constructs, where an Active Pattern called *DlmMethod* is defined in five lines of code that put together other four Active Patterns, for a total of **nineteen** code lines. Then, in the *typecheck_method_or_constructor* function, the DlmMethod pattern is used to automatically attach DLM meta-data to the method, without the need to pass them around as five distinct parameters. It is already an interesting feat if it is considered that this hides a lot of work behind the scenes, such as looking for annotations for the correct name, reporting eventual syntax errors, parsing the contents of the annotations and returning default labels if any of them is not defined. All just by simply passing a *method_signature* object. But what truly makes this mechanism amazing is the fact that all these computations are declared as *lazy*, which means that they are performed only once, and only when they are actually needed. Should be that, for whatsoever reason, one or more of this labels is not required, then all the aforementioned computation is not performed at all.

CustomMonad This module is used to implement all the special methods used to define the monad, as previously explained in Section 6.3. It also declares the state of the monad and several functions used for its manipulation.

TypeChecker The TypeChecker module is the earth of the AFC implementation, and it is also by far the largest and most complex module. It is composed by twelve monadic functions that sift through the whole program and implement all the rules defined in the implementation. Together with the Detector module it is perhaps the one that allows to appreciate most the elegance of functional languages, with their strict resemblance to mathematical logic rules. All these functions heavily rely on the Pattern Matching feature, which allows to produce code that presents astonishing affinities with the formal rules, as in the example below.

```
match s with

    ...

    | J.Decl ((ty, id, D.DlmLabel label), None) ->
        let! label = parse_label label loc
        do! bind in_var_env id (Some label, ty, None)

    | J.Decl ((ty, id, D.DlmLabel label), Some init_expr) ->
        let! label = parse_label label loc
        do! bind in_var_env id (Some label, ty, Some init_expr)
        do! typecheck_statement ctx (J.Assign (J.Var id), e), init_expr.location)

    | J.Decl ((ty, id, _), None) ->
        do! bind in_var_env id (None, ty, None)

    | J.Decl ((ty, id, _), Some init_expr) ->
        let! lb = check_expr init_expr
        let! expr_lb = combine_with_pc_label [lb] loc
        do! bind in_var_env id (Some expr_lb, ty, Some init_expr)

    ...
```

Conclusions

Protecting data secrecy is a long-standing difficult problem. *Discretionary access control* models do limit damages with data disclosure prevention, but do not solve the problem, being unable to stop data propagation. As a matter of fact, once a piece of information flows to an authorized entity, it exits from the grasp of these models, thus becoming vulnerable to human carelessness and unsafe programming practices. Information Flow control mechanisms are intended to monitor data propagation, disallowing all information transfers to untrusted sinks. The first attempts at devising Information Flow models in the literature, however, did lack the ability to cope with many real-world situations, because they failed to properly handle all those scenarios in which interested entities mutually distrust themselves and have different security requirements. The *Decentralized Label Model* proposes to solve the problem by allowing every entity (or *principal*) to express its own security requirements on data. Hence, in an environment protected by the Decentralized Label Model, only those information flows that respect all security requirements from all principals are allowed.

Nowadays, mobile devices such as phones and tablets are spreading with astonishing speed. Such is their diffusion that they have long since overtaken both desktops and laptops in selling volumes. It was just a matter of time before the market would see several high-quality Operating System designed ad-hoc for them. Amongst these competitors, the most adopted one is currently Android OS, distributed by Google. Despite a double layer of protections, that is a sandbox environment for applications and a permission-based API to access system resources, several studies have shown that Android applications are far from being safe, both regarding data secrecy and integrity. Software to be deployed on the Android Operating System needs to be written in the Java programming language. As of now, the only implementations of the Decentralized Label Model for Java are JIF[36] (Java Information Flow) and JLift[37], which is an extension of JIF. Until one month before the publication of this thesis, that is several months after the work herewith described started, JIF compliance with respect to Java was limited to version 1.4, while the Android Operating System is based on a custom implementation that strictly resembles the 1.6 version. Just recently JIF has been updated so as to be used with the latest Java Development Kit release, but it is still not compatible with Android. As if it was not troublesome enough, there are several challenges posed by Android, as for example a completely non-standard control flow, that need a system specifically tailored to tackle them.

The goal of this work is to make a step forward in the security of Android applications, with the design and the implementation of a full-fledged DLM type checker that is explicitly targeted for them. Additionally, it sets the objective of filling a severe lacuna that has troubled all Information Flow type systems up till now, that is their excessive restrictiveness. As a matter of fact, Information Flow has yet to be widely accepted, because of the difficulty to program real-world applications in languages that support it. For this reason we felt that, with some effort in conceiving a more practical system, it could be possible to develop a tool whose usefulness is not limited only to its scientific improvements, but that can also be proposed as a viable tool to any developer concerned about security of its applications. The product of this work, however, did not have to be a stand-alone tool, but needed to be designed to be part of a larger framework, called Lintent. Lintent further increases the run-time security standards by typing Intents and preventing Privilege Escalation attacks. Always with the same goal of achieving the highest usability possible, it is designed as an ADT Lint extension. ADT Lint is an official Eclipse plug-in, distributed and supported by Google, that is heavily suggested to all Android programmers. The reason behind this choice is that it allows Lintent to become part of an environment that is familiar to most developers.

In Chapter 5 a formal model has been proposed, that represents a first attempt at describing how to use Information Flow to improve data secrecy and integrity of Android applications. The proposed type system, called Android Flows Checker (AFC), comes with a set of typing rules that describes how to deal with most Java constructs, supporting all Java releases up to the latest one currently available, that is version 1.7. This allows AFC to be compatible with an eventual newer release of the Android API without the need of further work. Only a few statements and expressions, such as type casts, are not given a typing rule, because they do not provide any interesting notion about Information Flow. Nevertheless, they have been correctly handled in the actual implementation of the type checker, described in Chapter 6, that implements the aforementioned type system. The chapter also describes the syntax for Annotations, that are a standard Java feature used to pass meta-data to the compiler. As such, they have been exploited as a way to supply information to the type checker without altering the its run-time behaviour or performances. Finally it gives an in-depth description of its modular structure and the choices that have been taken, together with their reasons. We believe that the final product of our work gives to the developers of Android applications a powerful tool to greatly improve the security of their products. We also feel to have achieved the goal of improving the usability with respect to previous Information Flow systems, as the reduced amount of required labels, and a better error reporting system, both allow for a smoother learning curve.

C.1 Future Work

Albeit the proposed type system provides a viable way to improve the security of Android applications, there is still much work that can be done to further improve the model. In previous attempts at Java Information Flow systems, exceptions are handled in a way that makes programming too much strenuous. Civitas[23] demonstrates how much burdensome is exception handling in JIF. Developed by the same authors of JIF, they used their language to implement a secure voting system. In a little more than ten thousands lines of code there are 528 empty catch statements[22]. In about the 80% of them, the caught exception is called *imposs*, a clear indicator that the same authors recognized that JIF compels the programmer to stuff code with useless *try-catch* statements. This is a fulgid example of why Information Flow systems are having troubles spreading. We believe that, instead of handling exceptions in such a clumsy way, it is much better to ignore them until a smarter way to cope with them is found, as they only allow the creation of covert channels that are difficult to exploit. Hence, conceiving a good strategy to prevent implicit flows from exceptions, without unloading the encumbrance of managing them to the developer, would surely represent a further step forward in the security of Android applications.

Some features of the proposed type system are still in the design phase, as time was pressing and it could not be possible to perform further studies. Chances are that, with a deeper investigation, it would have been possible to devise smarter and more neat mechanisms to handle *dynamic labels* and *Generics* (i.e. Type Parameters). Moreover, implicit Intents are indeed troublesome. At the moment every statically-undecidable case is treated as if its run-time counterpart corresponds to the worst-case possible. Although this approach guarantees the safety and the soundness of the system, it is likely to be considered as overly-restrictive in some situations. A further set of Annotations would probably improve the situation, but we felt that a sound solution would require a meticulous series of trial-and-error studies that are not possible yet, due to the *alpha* state of the type checker implementation. Likewise to Android Flows Checker , also Lintent is currently under active development. Once completed, it will allow to perform more in-depth studies on real-world Android applications. It is likely that this increased knowledge, gained through empirical facts, will allow to devise new ideas to further increase security warranties and to improve the practicality of Information Flow systems on Android applications.

Bibliography

[1] Myers, A.C. and Liskov, B., *A Decentralized Model for Information Flow Control*. 16th ACM Symposium on Operating Systems Principles, 1997.

[2] Bugliesi, M., Calzavara, S. and Spanò, A., *Taming the Android Permissions System, by Typing*. Università Ca' Foscari di Venezia, 2012.

[3] Myers, A.C. and Liskov, B., *Protecting Privacy using the Decentralized Label Model*. ACM, 2000.

[4] Myers, A.C., *Mostly-Static Decentralized Information Flow*. Massachusetts Institute of Technology, 1999.

[5] Spanò, A., *Information Extraction by Type Analysis*. Università Ca' Foscari di Venezia, 2013.

[6] Austin, T.H. and Flanagan, C., *Efficient Purely Dynamic Information Flow Analysis*. University of California at Santa Cruz, 2009.

[7] Enck, W. et al., *TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones*. 9th USENIX Symposium on Operating Systems Design, 2010.

[8] Biba, K.J., *Integrity Considerations for Secure Computer Systems*. MTR-3153, The Mitre Corporation, 1977.

[9] Enck, W., Octeau, D., McDaniel, P. and Chaudhuri, S., *A Study Of Android Application Security*. USENIX Security Symposium, 2011.

[10] Felt, A.P., Chin, E., Hanna, S., Song, D. and Wagner, D., *Android Permissions Demystified*. University of California, Berkeley, 2011.

[11] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna S. and Chin, E., *Permission re-delegation: Attacks and defenses*. USENIX Security Symposium, 2011.

[12] Ongtang, M., McLaughlin, S., Enck, W. and McDaniel, P., *Semantically Rich Application-Centric Security in Android*. Pennsylvania State University, 2009.

[13] Fuchs, A.P., Chaudhuri, A. and Foster, J.S., *SCanDroid: Automated Security Certification of Android Applications*. University of Maryland, 2009.

[14] Armando, A., Costa, G. and Merlo, A., *Formal Modeling and Verification of the Android Security Framework*. 7th International Symposium on Trustworthy Global Computing, 2012.

[15] Mann, C. and Starostin, A., *A Framework for Static Detection of Privacy Leaks in Android Applications*. 27th Symposium on Applied Computing: Computer Security Track, 2012.

[16] Fenton, J.S., *Memoryless subsystems*. Computing J., 1974.

[17] Chin, E., Felt, A.P. , Greenwood, K. and Wagner, W., *Analyzing inter-application communication in Android*. MobiSys '11, 2011.

[18] Maji, A.K, Arshad, A.F., Bagchi, S. and Rellermeyer, J.S., *An empirical study of the robustness of inter-component communication in Android*. DSN, 2012.

[19] Bartsch, S., Sohr, K., Bunke, M., Hofrichter, O., and Berger, B.J., *The Transitivity of Trust Problem in the Interaction of Android Applications*. CoRR, 2012.

[20] Syme, D., Granicz, A. and Cisternino, A., *Expert F# 2.0*. Apress, 2010.

[21] Cormac, H., *So Long, And No Thanks for the Externalities: The Rational Rejection of Security Advice by Users*. Microsoft Research, 2010.

[22] King, D., Hicks, B., Hicks, M., Jaeger, T., *Implicit Flows: Can't Live With 'Em, Can't Live Without 'Em.*. The Pennsylvania State University, 2008.

[23] Clarkson, M.R, Chong, S., Myers, A.C., *Civitas: Toward a Secure Voting System*. Cornell University, 2007.

[24] Project Lombok, *Lombok AST Repository on Git-Hub*, 2013.
https://github.com/rzwitserloot/lombok.ast

[25] Lippert, E., *Continuation Passing Style Revisited*, 2010.
http://blogs.msdn.com/b/ericlippert/archive/2010/10/21/continuation-passing-style-revisited-part-one.aspx

[26] HaskellWiki, *Monad*, 2013.
http://www.haskell.org/haskellwiki/Monad

[27] Wikipedia, *Apache Harmony*, 2013.
http://en.wikipedia.org/wiki/Apache_Harmony#Use_in _Android_SDK

[28] Canalys, *Google's Android becomes the world's leading smart phone platform*, 2011.
http://www.canalys.com/newsroom/googles-android-becomes-worlds-leading-smart-phone-platform

[29] Oracle Corporation, *Java^{TM}Platform, Standard Edition 7, API Specification*, 2013.
http://docs.oracle.com/javase/7/docs/api/

[30] Microsoft Corporation, *F# Language Reference*, 2013.
http://msdn.microsoft.com/en-us/library/dd233181.aspx

[31] Android Developers Guide, *Keeping Your App Responsive*, 2013.
http://developer.android.com/training/articles/perf-anr.html

[32] Johnson, K., *The urgent need for mobile device security policies*, 2011.
http://www.gsnmagazine.com/node/24983

[33] OODesign.com, *Visitor Pattern*.
http://www.oodesign.com/visitor-pattern.html

[34] Eclipse Foundation, *Eclipse Juno 4.2 Documentation*, 2013.
http://help.eclipse.org/juno/index.jsp

[35] Android Tools Project, *Android SDK Tools & Eclipse plug-in (ADT)*, 2013.
http://tools.android.com/release

[36] Meyers, A.C., *Jif Reference Manual*, 2012.
http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html

[37] King, D., *JLift*, 2008.
http://siis.cse.psu.edu/jlift/jlift.html

[38] Frazza, A., *Android Flows Checker source code*, 2013.
https://github.com/alvisespano/Lintent/blob/master/src/LintentEngine/DlmTyping.fs

[39] Spanò, A., Frazza, A., *Lintent Repository*, 2013.
https://github.com/alvisespano/Lintent.