Crystalline: A Privacy-Aware Middleware for the Android Platform

Eduardo Joaquim Leitão Libânio Gomes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s):  Prof. Dr. João Nuno De Oliveira e Silva  
Prof. Dr. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Dra. Teresa Maria Sá Ferreira Vazão Vasques  
Supervisor: Prof. Dr. Nuno Miguel Carvalho dos Santos  
Member of the Committee: Prof. Dr. Pedro Miguel dos Santos Alves Madeira Adão

November 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
For my parents,
Acknowledgments

First, I want to thank my supervising professors, João Silva and Nuno Santos, for all continuous support, patience, and guidance throughout the development of this thesis. Igor Zavalishyn also played a crucial role in this work, as his input and advice were very incisive. I would also like to thank my mom and dad for their love and for the endless support across all my life. In all fairness, I am grateful for the all encouragement provided by my whole family, with special mentions to my siblings, my niece, my aunt, and of course, my friend Tetê. In the last year, I was lucky enough to be accompanied by my beautiful and supportive girlfriend, which helped me surpass stressful times with her love and affection. Finally, I would like to thank all my friends for the good times and the motivation they gave me through all my academic journey. My friends Mário Vieira and Bonzas also deserve an individual mentions. Mário set me up on the path that led me to this thesis, and Bonzas’s ability to keep track of our household bills was crucial in motivating me to commute to the office space in order to work.
Resumo

Na última década, o mercado de aplicações móveis cresceu exponencialmente. Devido ao potencial que têm para facilitar as nossas rotinas diárias, estas aplicações conseguem rapidamente cativar muitos utilizadores. No entanto, elas também trouxeram problemas de privacidade para os consumidores. Muitas aplicações acedem aos sensores do dispositivo para extrair dados pessoais, que muitas vezes acabam por ser vendidos para outras empresas. O Android aplica medidas para proteger a privacidade dos utilizadores, informando quais os recursos a que cada aplicação tem acesso. No entanto, por si, esta estratégia não é suficiente para que o utilizador consiga perceber como uma aplicação trata os seus dados, visto que informar que eles são acedidos nada diz sobre como são processados pela aplicação. Esta falta de informação impede que os utilizadores consigam compreender as possíveis consequências de usar determinadas aplicações.

Para aumentar o controlo que os utilizadores Android têm sobre a sua privacidade, propomos um mediador entre utilizadores e aplicações. O Crystalline disponibiliza aos desenvolvedores um paradigma de programação orientado à privacidade, enquanto deixa ao encargo do utilizador mecanismos para controlar como os seus dados privados são tratados por cada aplicação. Adotamos algumas ideias introduzidas numa proposta recente para melhorar a privacidade em ambientes de casas inteligentes, e transformamo-las num sistema compatível com Android, que garante o cumprimento das políticas de privacidade de cada utilizador sem alterar o sistema operativo. Graças a uma avaliação experimental, conseguimos afirmar que o Crystalline dá ao utilizador controlo sobre os seus dados sensíveis sem comprometer o bom funcionamento das aplicações.

**Palavras-chave:** Privacidade, Android, Aplicações Móveis, Dados Sensíveis, Programação Android
Abstract

Over the last decade, the mobile applications market reached an enormous size. Due to their friendly user interface and high potential to ease our daily routines, apps can quickly captivate a large heterogeneous consumer base. However, they also brought new privacy concerns for consumers. Many apps access the device's sensors to collect the user's personal data, which they send to cloud servers, and often, sell it to organizations like marketing or insurance companies. Android currently has some measures to protect its users' privacy by informing what resources each app has access to. However, this strategy is not enough for users to understand or control what happens to their sensitive data, as only stating what data is accessed by an app, tells little about how the application processes it internally. This lack of information prevents users from realizing the potential consequences of using certain apps.

To improve users' understanding and control over how apps handle their private data, we propose a trusted middleware between users and Android applications. Crystalline allows developers to program with a privacy-oriented framework while empowering users with the ability to control how their sensitive data flows within each app. We adapted some ideas introduced in a recently proposed solution for privacy in smart homes into an Android compatible framework that does not require any modifications to the OS while still securing the user's privacy policies for each app. An experimental evaluation shows that Crystalline gives users control over their data without compromising the smooth running of applications.

Keywords: Privacy, Android, Mobile Applications, Sensitive Data, Android Development
Contents

Acknowledgments .......................................................... vii
Resumo ................................................................. ix
Abstract ............................................................... xi
List of Tables ........................................................... xv
List of Figures .......................................................... xvii
Glossary ................................................................. xix

1 Introduction .......................................................... 1
  1.1 Motivation ....................................................... 1
  1.2 Contributions .................................................. 3
  1.3 Thesis Outline ............................................... 4

2 Related Work ......................................................... 5
  2.1 Android Security Model ...................................... 5
  2.2 Privacy Problems on Android ................................ 10
  2.3 Android Apps Security Analysis Tools ...................... 14
  2.4 Android Permission Enhancement Mechanisms ............. 18
  2.5 HomePad ....................................................... 22
  2.6 Summary ....................................................... 26

3 Architecture .......................................................... 27
  3.1 Requirements .................................................. 27
  3.2 Threat Model .................................................. 28
  3.3 Overall Architecture ......................................... 29
  3.4 Architectural Challenges ..................................... 30
  3.5 Applications Development Framework ....................... 31
  3.6 The Runtime Environment .................................. 35
  3.7 The App Management System ................................ 37
  3.8 Certification Mechanism .................................... 39
  3.9 Summary ....................................................... 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Development Framework</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Runtime Middleware</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Sandboxing Untrusted Elements</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>The Code Transformer</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>App Certification Service</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>The App Manager</td>
<td>57</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Effectiveness</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>Programming Effort</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Security Assessment</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Future Work</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Use Case Apps</td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>SimpleApp</td>
<td>85</td>
</tr>
<tr>
<td>A.2</td>
<td>PhotoUploader</td>
<td>87</td>
</tr>
<tr>
<td>A.3</td>
<td>HeartRateApp</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Comparison of privacy feedback and control benefits between Android permissions and 
Crystalline’s Privacy Manager, regarding the HeartRateApp (Local) . . . . . . . . . . . . . 62
5.2 Comparison of privacy feedback and control benefits between Android permissions and 
Crystalline’s Privacy Manager, regarding the HeartRateApp (Cloud-Local) . . . . . . . . . 64
5.3 Comparison of privacy feedback and control benefits between Android permissions and 
Crystalline’s Privacy Manager, regarding the HeartRateApp (Cloud-Local) . . . . . . . . . 65
5.4 Comparison between the lines of code written by the application developer, for three dif-
ferent apps, created with and without Crystalline’s framework. . . . . . . . . . . . . . . . . . 67
5.5 Comparison between the time take to build traditional and Crystalline applications. . . 69
5.6 Time taken for the Validator to verify the application’s integrity. . . . . . . . . . . . . . . . 70
5.7 Time taken for the Privacy Manager to inspect applications data-flows. . . . . . . . . . . 71
5.8 Runtime benchmarking tests results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
List of Figures

2.1 Application requesting permission to read and send SMS. 8
2.2 Application workflow when requesting the dangerous SMS permission. 8
2.3 Android permission enforcement scheme (based on [33]). 9
2.4 Relation between control and feedback on privacy treatment, and an application’s installation rate (from [13]). 12
2.5 FlowDroid’s taint analysis algorithm representation (from [14]). 14
2.6 TaintDroid’s runtime architecture (from [18]). 17
2.7 Poly Installation Interface (from [20]). 18
2.8 ipShield example use-case. 19
2.9 Picky app (from [24]). 20
2.10 BinderFilter concept representation (adapted from [24]). 20
2.11 Example HomePad application graph. 23
2.12 HomePad architecture (adapted from [30]). 25

3.1 Overall Crystalline framework architecture. 29
3.2 Structural differences between HomePad, Android, and Crystalline apps. 32
3.3 Some Crystalline modules, and their trusted elements. 34
3.4 Runtime architecture of a Crystalline application. 35
3.5 Normal workflow of launching a Crystalline app. 36
3.6 Process of defining privacy policies for a Crystalline app. 37
3.7 Installation process for certified Crystalline apps. 39
3.8 Certification process for Crystalline apps. 40

4.1 Visual representation of the example graph defined in Listing 4.2. 45
4.2 Simplified UML representation of Crystalline’s Runtime Middleware implementation. 47
4.3 Crystalline permission request at runtime to send data to a cloud server. 48
4.4 Crystalline’s sandboxing technique example. 52
4.5 AOP contribution for Crystalline’s integrity. 53
4.6 Certification process representation. 56
4.7 App Manager windows. 58

5.1 Heart rate flows in a read local, stay local app – HeartRateApp (Local) app. 62
5.2 Heart rate flows in a read from cloud, stay local app – HeartRateApp (Cloud-Local) app  63
5.3 Heart rate flows in a read local, send to cloud app – HeartRateApp (Local-Cloud) app  65
5.4 Memory usage comparison between Crystalline apps and traditional Android ones.  73

A.1 SimpleApp’s Activity 1 screen.  85
A.2 SimpleApp’s Activity 1 flow graph.  86
A.3 PhotoUploader window.  87
A.4 PhotoUploader Main Activity flow graph.  87
A.5 HeartRateApp (Local) MiBand2Monitor Activity flow graph.  90
A.6 MiBand2Monitor screens.  90
A.7 HeartRateApp (Cloud-Local) MiBand2Monitor Activity flow graph.  91
A.8 HeartRateApp (Local-Cloud) MiBand2Monitor Activity flow graph.  91
Glossary

**AOP** Aspect Oriented Programming. 4, 40, 52

**ART** Android Runtime. 8, 17, 49, 50, 54, 74, 78

**GDPR** General Data Protection Regulation. 2, 77

**GID** Group Identifier. 7, 8

**ICC** Inter-Component Communication. 15, 17, 32, 38

**IPC** Inter-Process Communication. 6, 7, 10, 16, 21, 72

**UID** User Identifier. 6–8, 59
Chapter 1

Introduction

Over the last couple of decades, smartphones not only revolutionized the way we communicate with each other but also extended drastically our ability to perform multiple daily tasks thanks to a vast market of mobile apps with different purposes. However, the widespread of mobile applications also brought new concerns for the general population. Many apps collect personal data from the multiple device’s sensors to draw inferences about the user’s social life, health, habits, and preferences. These types of data are highly valuable for advertisement companies. Still, their disclosure presents significant privacy concerns for users [1–3], which witness their private data being exposed to entities they do not know. Often, mobile applications rely on cloud services, to which they send their end-users private data under the pretext to adapt the application’s contents to each user. However, using apps that send private data to the cloud means users have to sacrifice their privacy, as from the moment data leaves the device, they lose control over it. In this work, we present Crystalline, a privacy-aware framework for Android. Crystalline allows developers to transparently convey how their apps handle users’ sensitive data while empowering users with the means to control how applications can actually process it.

1.1 Motivation

While users certainly benefit from the ever-evolving mobile applications and their features that explore the device’s capabilities, privacy concerns also arise from using these apps. Because smartphones can now connect to the Internet from practically anywhere, apps can explore this to transmit local data to the cloud. Many times this data is obtained from the device’s sensors and carries sensitive user information, whose disclosure has privacy implications for users. For example, a common practice between apps is to access the device location and inform a web-server where the user is while using the application – sometimes even in background – which allows companies to trace the user’s routines and habits. Other times, apps send out their users’ photos or even health data generated by Bluetooth fitness trackers. Privacy concerns are not obvious to define, as they depend on the inferences that can be extracted from each type of data, as well as the subjective importance each user attributes to them. Users with different professions, ages, cultures, or habits will most likely have different privacy preferences.
Recently, a study [4] found that a significant number of Android applications aimed at a younger population illegally collected and transmitted personal data of children without attaining parental consent. In even more recent news [5], a controversy fell upon the popular FaceApp application, which allows users to change their appearance in their photos, with age or gender swap filters. Because this app runs its algorithm on the cloud, every time someone uses it, it uploads the private photo to a web server. Despite the authors of FaceApp stating that they do not share the pictures with other companies, many users were uncomfortable upon learning that their photos ended up in the cloud, when the advertised service was to edit a local image. While these examples present relatable privacy concerns when using mobile apps, in some cases, the disclosure of sensitive data can even endanger user safety.

In 2017, a mobile app that allows cyclists and runners to track their fitness activities presented to the public [6] a worldwide data visualization map showing their user-base running courses. While this company made sure to anonymize the data – so that information could not be traced back to individual users – a critical problem was found [7]: The heatmap included locations of oversea US military bases, some within territories with active conflicts. The map was also detailed enough for anyone to see the facilities’ internal layouts, and to distinguish training routes, presenting a danger for military personnel.

However, the privacy problem on mobile devices is not recent news, and Google itself has been putting a continuous effort in improving its measures to protect its users’ privacy. Android is endorsed with security mechanisms that protect users from malicious apps [8, 9], and provide some information and control over what resources – e.g., contact list, location, Bluetooth connectivity, file system – each app accesses, through a permission-based system [10]. However, some studies [3, 11–13] argue that Android permissions are too coarse-grained and that providing information about access to resources is not enough for users to perceive how each application processes personal data. For example, stating that a fitness app has access to Bluetooth and the Internet is not enough for the user to accurately understand if the application collects the user’s heart rate, nor if it sends this type of data to the cloud.

A substantial amount of research has been done to improve privacy in Android-based environments. Some proposed solutions approach the problem by analyzing how sensitive data flows within applications, either with static flow analysis [14–17] or with runtime tainting of variables [18, 19], and then informing the user on potential leakages of sensitive data. Still, these mechanisms can be deceived by ill-intended developers. Other solutions [20–28] aim at giving users finer-grained control mechanics over their personal data by extending Android’s native permission system. However, we found that existing solutions fall short at controlling how personal data is processed inside applications. Furthermore, the most ambitious solutions modify Android’s OS core and consequently create serious compatibility issues that difficult their adoption from the majority of Android users.

While improvements to the way Android handles privacy would mostly benefit users, developers can also gain from new means to address some related issues on their side. With the adoption of the General Data Protection Regulation (GDPR) by the EU, organizations that process the data of EU residents must oblige by strict rules devised to protect the user’s privacy. While this legislation secures European users’ privacy interests on a legal level, the general lack of legal knowledge hinders application developers’ ability to translate what their legal advisors say into programming methodology objectively [29].
1.2 Contributions

The main contributions of this work were the design, implementation, and evaluation of Crystalline, a middleware framework whose goal is to improve Android users’ control over their privacy. We provide a middleware framework that does not change Android’s core but allows the development of applications’ with their data flows transparent to users. With Crystalline, the user can understand what sensitive data applications collect and how each app treats specific types of data - either processing it purely on the device or also sending it to outside services, like cloud servers. Our framework also allows users to define fine-grained privacy policies to restrict applications from handling sensitive data in specific ways – for example, blocking an app from disclosing a particular type of data to one specific web server. We also provide means for developers to create privacy-aware Android apps supported at runtime by our middleware. Crystalline’s development framework allows developers to objectively expose how they planned their apps to handle sensitive data. At runtime, Crystalline apps do not suffer from a high-performance overhead. Our runtime middleware does not compromise user experience but also blocks applications from processing data against the user-defined privacy policies.

Technical challenges and solutions: In the design of Crystalline, we adopt some ideas from a recently proposed privacy-aware middleware for smart homes named HomePad [30]. Whereas traditional smart home apps run on cloud servers, HomePad apps run on the smart home’s hub. Homepad supports applications structured as functionality blocks, connected through explicit interfaces. This way, HomePad can inspect and control how sensitive data flows between these blocks. Users can define privacy policies to restrict some flows of data, and thus limit the application’s ability to treat specific types of sensitive data in particular ways, e.g., blocking the app from sending photos to the cloud. For an application to run on the hub, the developer must code it through a data-flow oriented programming model that respects HomePad’s policy of having apps divided into functionality blocks.

Our work starts by bringing HomePad’s ideas to Android, adapting its programming model into a compatible Android development framework, solving all the incompatibilities between Android’s traditional programming paradigm and HomePad’s. We then provide users with a privacy management system that they can utilize to observe how each app handles sensitive data, as well as define fine-grained, but straightforward, privacy policies that restrict how specific types of data flow within applications.

Crystalline has a runtime middleware that translates this data-flow oriented programming model into instructions that Android understands. Our middleware also safeguards users’ privacy at runtime, by first, only allowing data to flow through each app as conveyed to users, and second, forcing each application to respect the user-defined privacy policies. In practice, Crystalline controls how data flows through the app, and blocks the developer’s code from accessing Android’s API directly. To achieve both these requirements, we need to ensure that our middleware has full authority over the application execution environment. The straightforward way to accomplish this would be deploying the middleware as part of the operating system. However, we do not want to change Android’s core. We, therefore, placed the middleware inside each app, but also without repacking it. Instead, we deliver the middleware as a
library for developers to import into their apps before releasing them to users.

Having the middleware injected into applications by developers themselves presents us with two challenges to solve. First, assure that our middleware has authority over applications’ runtime. Second, guarantee that no ill-intended developer tampered with our middleware library, nor its authority. We address the first one with the help of a code transforming mechanism, which relies on aspect-oriented programming (AOP) to weave the application and inject specific code patterns with instructions that forward control to our middleware while blocking any dangerous code. We also devised an app certification service which guarantees both that submitted applications were correctly weaved, and that they run with an unadulterated Crystalline middleware. We then provide users with an application installer app that ensures that only applications certified by Crystalline are installed through it.

Summary: In summary, the main contributions of this thesis are as follows:

- A runtime middleware that controls how sensitive data flows within supported Android applications.
- A privacy manager for the user to observe how each app handles his sensitive data, and define privacy policies that suit his preferences.
- A development framework for developers to create supported, privacy-aware applications.
- A certification mechanism that vouches for the integrity of apps that passed a verification process.

1.3 Thesis Outline

The rest of this document is structured as follows. Chapter 2 covers both the current security and privacy mechanisms Android offers its users and the most relevant proposed solutions to improve privacy in Android devices. Chapter 3 describes Crystalline’s design requirements, the architecture we developed, and how it solves the challenges we faced. Chapter 4 presents our framework’s implementation. In Chapter 5, we evaluate our system and the extent to which it meets the initial requirements. Finally, Chapter 6 summarizes our work and proposes some directions for future work.
Chapter 2

Related Work

In this chapter, we discuss the current state of the art of protecting users’ privacy on Android devices. In Section 2.1, we present Android’s security model, its permission system, and how it helps users’ understand and control how they disclose sensitive data to apps. Next, in Section 2.2, we discuss privacy issues still present in the Android environment and look into relevant research that attempts to identify them and the reasons for their existence. We then look into proposed solutions that help users’ protect their data, either by analyzing how apps behave (Section 2.3) or by improving Android’s permission system (Section 2.4). We observe that none of the solutions help users’ understand entirely and control how apps process their data. Finally, in Section 2.5, we examine HomePad [30], the way it monitors users’ privacy treatment in smart homes, and why Android devices can benefit from its concepts.

2.1 Android Security Model

Android is a widely adopted, open-source mobile OS, developed by Google. Applications can be developed by anyone, with good or bad intentions. To protect the user from apps’ malicious behavior, Android has security mechanisms that restrict their potential for harming the device or violating the user’s privacy. The OS isolates applications within their own domains [9], only allowing interactions with other apps or system resources through a well-defined set of interfaces. These interfaces are controlled by Android OS, which supervises whether or not an application has the necessary permissions to access some external functionality [10]. Each permission can be granted either by the OS or the user.

Because this work aims to tackle privacy problems currently bestowed upon Android users, in this section we present Android’s security model, which covers several aspects: sandboxing of applications, controlling inter-process communication (IPC) between apps, a permission-based system that determines what resources an app may access, the verification of the application’s authenticity, and finally, Google’s app scanning mechanism, which protects users’ devices from malware.

Applications sandboxing: Because Android was built on top of Linux, it leverages on the latter’s native security mechanisms. Each applications runs within its own isolated process and can not access
each other’s memory [9]. Moreover, Android’s kernel also considers each app running as a distinct user. This is due to the fact the low-level implementation of the Linux user's spaces on Android counts both the device's personal user accounts and the installed applications. An application’s unique user identifier (UID) is associated with the physical user’s ID and the app’s unique signature. This mechanism assures the isolation of applications on a system level. Consequently, they also have their own persistent and isolated storage directories, as well as other resources the system deems unshareable between them. Additionally, Android is able to isolate the same app installed for different users, as the its UID also depends on the user’s. Enck et al. [31] explain in some detail how Android's sandboxes applications.

Nevertheless, apps are allowed to communicate with the outside through some interfaces. They can use system drivers within the system kernel or interact with other apps through a centralized IPC broker.

**IPC control:** Two applications can communicate with each other through Android’s Binder mechanism [32]. Binder is a driver for higher-level IPC communication supported by the OS kernel. It has an essential role in security because it oversees communications and authenticates both interacting processes. This is also an indispensable requirement to defend users’ privacy, as most of their personal data accessible within the device is provided by services or applications running on different processes. For example, the users’ location and contact list are maintained by the native System Server process, while his Facebook profile and social circles information are the responsibility of the Facebook application.

Binder provides a well-defined interface for apps to communicate. It implements a client-server protocol, as the application making the requests is the client of the one serving them. When an app sends a message to another, the Binder driver, after verifying the former’s authenticity, reserves a part of the receiver’s memory, in which it writes the message. When an app makes an IPC call, the Binder driver provides the called process with the UID of the calling one. The former can then check if the sender has the necessary authority by querying Android if the app associated with that UID has the required permissions.

**Permission system:** Android employs a permission system that controls applications’ access to system resources. Android’s development team states on their web page: “The purpose of a permission is to protect the privacy of an Android user” [10]. In practice, permissions determine if an application may access resources like the Internet, the file system or data such as the user’s contact list, or GPS location. The permissions with more sensitive privacy or security implications are only granted to apps with the user’s explicit consent. This way, Android empowers users with the decision to whether or not grant apps with access to sensitive data, as well as helping them perceive if an app has malicious intent. For example, if an offline single-player game asks the user for permission to access the contact list, he or she may wonder whether the application intends to steal information, as contacts are definitely not required to play a solo game.

Android groups permissions into three categories, depending on the privacy risks they carry:

- **Normal** are considered by Android to have, little privacy and security implications [10]. They are automatically granted by the OS to applications that request it, e.g., permission to set an alarm.
• **Signature** permissions were not designed for the user, but to give applications a method of sharing data with others developed by the same creator, e.g., the Facebook app may implement signature permissions to allow Facebook Messenger and Instagram apps to obtain the user’s session.

• **Dangerous** permissions are related to the disclosure of sensitive data or to functionalities that control the device, e.g., access to the user’s location or the ability to send SMS automatically. These require the user to give consent on runtime through a pop-up window, managed by the OS.

Developers need to take into account what permissions their apps require, and design and implement them accordingly. First, they must add an entry to the application manifest for each permission the app needs. For example, as presented in Listing 2.1, if the app communicates with the cloud, then the manifest must refer the Internet permission. If the app only needs normal permissions, Android automatically grants them to the app, and users are not notified. However, if an application requires dangerous permissions, then it must ask the user to grant them at runtime, and act accordingly to his decision. For example, sending SMS automatically from the user’s phone is considered a dangerous functionality with a wide range of possible worrisome consequences. As depicted in Figure 2.1, Android only grants it if the user allows so, through a pop-up window managed by the OS. As presented in Figure 2.2, if the user denies the permission request, then the app is notified, so as to act accordingly. This mechanism allows users to understand and control what resources each app has access to.

As shown in Figure 2.3, permissions are enforced when the application either performs an IPC request to another app or service or attempts to access a kernel driver. As stated in [32], when the user installs an application, Android safely stores a file containing the app’s signature, its UID, the granted permissions, and its multiple group identifiers (GIDs). This file is also updated when the user revokes permissions or grants new ones through the pop-up window or the system settings menu. When a system service receives an IPC message, it verifies the sender’s permission by asking the OS if the UID of the requesting process was granted with the required permissions. For example, the system server only allows another process to send SMSs if its UID was granted with the SEND_SMS permission.

Additionally to IPC communication, many native Linux functionalities may also come as sources and sinks for sensitive data. For instance, there can be personal information stored on the file system, which can be leaked by making Internet calls. The file system and Internet sockets are two resources controlled by Linux’s kernel. While the system also takes into account an app’s permissions when it attempts to access a system driver, the verification process is different from the one done in IPC communications. The

Listing 2.1: Example app manifest file that declares the need to access the Internet (from [10]).
permissions that control access to these resources are transformed, when the application is installed, or the user changes its settings, into multiple GIDs the system grants it with. This allows Android to leverage the group-based access-control natively supported by Linux. For example, the system tests if the application has permissions to access Internet sockets by verifying if it was assigned with the AID_INET GID, which is given to an app granted with the ACCESS_Internet permission [32].

Applications authenticity verification: Another important feature of Android’s security model is the way it verifies applications’ authenticity. A malicious developer may implement a fake app that impersonates a legitimate one by assigning it the same title, icon, and package name. The objective when doing this may be to leverage on the permissions granted to the original application in order to obtain sensitive information. Android prevents this attack by giving each application a user identifier (UID) at install time, and associating it with the application signature [32]. Assuming that the signature is proof of origin, no fake app can access the original’s space to steal user’s data or pretend to have its permissions. To understand how applications’ signatures work as their authenticity identifier, we should first look at how Android apps are composed.

Applications are shipped in the APK format, which very is similar to JAR archives, but in addition, it contains application manifest and the classes compiled into DEX files. A DEX file constitutes the binary representation of classes, the same as JVM’s CLASS files, but for Android’s ART engine. Because APKs are based on JARS, they inherit the same signing mechanism [34]. The META-INF directory contains the files that guarantee the application authenticity. Inside it, we can find a manifest file stating hash values of the multiple DEX files, a signature file that also holds hashed values, but generated from the contents of manifest itself, and finally, a certificate file vouching for the app’s signature.
An app published to the Google Play Store must be signed by its developer's private key and contain a certificate stating the developer's public key, which can be used to verify the app's authenticity. The certificate itself is a document signed by a trusted authority, which ensures that the public key within the package belongs to the developer. While trusted certificate authorities can issue certificates, Google also lets developers self-sign their applications' certificates. This is allowed because, for a developer to publish an app to Google Play, he or she must be signed in Google's online platform, which guarantees one's authenticity, and consequently, the certificate's signature.

When the application is installed, Android computes digest hashes for its contents and compares them against the ones on the manifest file. This step guarantees that the application was not tampered with after the manifest was generated (at build time). Next, Android uses the developer's public key included in the certificate file to read the signature file and see how the hash of the manifest should look like. Then it calculates a digest of the manifest and obtains the corresponding hash. If the latter matches the one included in the signature file, then the application was not tampered with. This mechanism allows Android to be sure that the application came from the developer vouched by the certificate file. However, an ill-intended developer could corrupt the certificate in order to alter the developer's public key it vouches for and thus deceiving the verification. To prevent this from happening, Google encourages users only to install apps from its Play Store, since it authenticates the developer's signature in a cloud-based service and assures the developers duly signed their applications with valid certificates.

This procedure allows Android to be sure an app came from the developer who originally published it, and it is not a fake app developed by an ill-intended developer.

**Google's app scanning mechanism:** Google's Play Protect [8] system comes natively with Android. It offers users with two safety walls against malicious apps: one in the cloud and one on the users' devices. A cloud-based service scans applications before they enter Google Play and re-scans them at a periodic rate. If an app is labeled as malicious by the scan, then Google removes it from the store. These scans are based on a self-improving machine-learning algorithm, which allows Google to detect dangerous patterns in applications' code. Additionally, Google tracks the application developer's record in search of any suspicious history. Both these tasks determine if the app is deemed trustworthy or categorized...
as a PHA – Potentially Harmful Application. Locally, Play Protect scans the user’s device at a daily rate in search of a PHAs and maintains communication with the cloud both to get new information about malicious code patterns and to provide feedback to improve the online machine-learning algorithm. An application also undergoes a scan at install time, or when the user manually requests it.

**Discussion:** In this section, we saw that Android employs a variety of techniques to protect its users’ privacy. Users can control how apps access the device’s functionalities by granting or denying permissions [10]. Play Protect [8] defends users from malicious apps by categorizing code patterns on the cloud and scanning both the apps available in the Play Store and the ones installed on Android devices. The signature and its developer’s online authentication assure that not fake applications can obtain privileges given to the original. By sandboxing apps, Android makes sure that they can not access each-other domains to collect sensitive data. Android’s control over the IPC calls guarantees both interacting processes authenticity. Finally, enforcing permissions on a kernel-level and on system services assures that the applications conform to the restrictions users’ imposed on them. However, there are still some privacy problems that Android does not solve completely, which we discuss next.

### 2.2 Privacy Problems on Android

Despite the security measures employed by Android, a number of threats can still compromise the user’s privacy. Some risks are caused by limitations of Google’s Play Protect service on categorizing malicious apps [35, 36], others linked to the lack of transparency from applications when treating their end-users data [37, 38], and others even related to the permission system exposed to users [11, 13, 39]. In particular, the native Android’s permission system, while giving users some knowledge and control about the disclosure of sensitive data, also adds ambiguity to the perception users have on the way apps treat their data. Some benign applications need access to an unjustified large set of resources to work while others ask to access them with straight-on malicious intents in order to steal the user’s private data. In this section, we identify the most prominent privacy risk untackled by Android and discuss some relevant work that explores these issues.

**Google’s scans are not failproof:** Today, Play Protect [8] protects Android users from malware apps by scanning applications when they enter both its Play Store and users’ devices. However, its scanning algorithms still overlook some malware [38]. More importantly, a considerable number of apps available within Google Play, while delivering the functionalities they promote, also do some dubious processing of users’ data. In a survey conducted in 2011 to investigate malware patterns in mobile environments, Felt et al. [40] differentiate malware from grayware. On the one hand, malware are only designed to either steal user’s data or exploit the device’s resources to its gain – the most common example of these is farming cryptocurrencies with the device’s processor or sending premium-rate SMS in the user’s behalf. These applications are blocked by Google as soon as they attempt to enter the Play Store. Grayware, on the other hand, get this designation because they can be categorized both as malicious and non-
malicious applications. They usually fulfill the purpose for which the user installed them. However, they also collect one's data to sell it to third party companies. Users often neglect these apps' privacy agreements, which tend to be craftly written to be overlooked, annoying to read, and ambiguous. This pattern is typical among many free applications available on the Google’s App Store.

The most common practices used by malicious applications involve procedures to steal users’ private data or even their credentials. Furthermore, access to the Internet was proven to have little to no correlation with an application being malicious. Other studies confirm these findings [35, 41, 42]. Since access to the Internet is a requirement for the majority of applications, this criteria does not come as an effective approach to identify malicious apps. However, cloud calls are indeed the main channels for most private data leakages, as applications can easily perform one to send data to their servers. This led to Google concluding that simply looking at the permissions of an application is not enough to categorize it as malicious or benign. Instead, the current scans were designed to detect code patterns, within apps, categorized as harmful. This is also not a failproof approach to identify malicious apps, as new and unknown malicious patterns will continually arise to trick the scanner.

For instance, in recent news, Kaspersky revealed [36] that the CamScanner app, listed in Google Play Store with over 100 million downloads, included an advertisement library with a trojan malware inside. The goal of this popular application is to scan document photos taken with the phone’s camera and convert them into PDF files. However, the advertisement library injected it with a module that dynamically loaded encrypted Java classes present in its resources folder, which in turn downloaded and executed malicious code to either present the user with impertinent adds or sign him in third-party, unsolicited paid subscriptions. The fact that this malware has successfully deceived Google Play Protect scans demonstrates some of its limitations in protecting users from malicious applications.

**Overpriviledged apps:** Alenezi and Almomani [37] present research results that show that around 80% of applications intended for education purposes request more permissions than they actually need. They also make the case that this kind of practice opens the door for malware exploitation. This becomes particularly relevant when applications contain third-party advertisement libraries, which may maliciously exploit the application’s permissions to leak sensitive data or perform other dangerous operations.

Overpriviledged apps can also become a target for cloned, malicious fake versions of them. Some users disable Google’s safety measures on their devices so that they can download paid apps for free from alternative markets and distributors. When doing so, they are dismissing Google’s online mechanism of verifying developers’ authenticity, and thus exposing themselves to clone applications with malware implanted (again, most of them actually do work, but steal sensitive data), something Li et al. [38] refer as piggybacked apps. The creators of such applications target mostly games, likely due to their extensive premium market, which lures unwilling-to-pay users into piracy. The typical behavior of these apps is to inject some method calls into the normal flow of the original app, and leverage the user’s trust in the latter to get permissions and steal personal data. Since these applications do not come from Google Play, and because Android allow apps to be self-signed, the attacker can trick the system by generating a fraudulent certificate of its own, stating that the app is from a trusted source.
Permissions are too coarse:  Another article [11] claims that, while overprivileged apps are indeed a problem, the main culprit is often not the developer, but the inherent nature of Android's permission system. The authors argue that permissions are too coarse-grained, and some of them can grant a single app multiple privileges, with a range of different privacy implications. As an example, they discuss the READ_PHONE_STATE permission, which allows an app to read the phone number, the device's IMEI, and in the case of an active call, the peer's phone number as well. Indeed, the fact that this permission grants access to multiple different types of data may prevent the user, and Android, from identifying what the application does, and consequently, if it has malicious intentions.

Another example of the coarseness of the permission system is the INTERNET permission. An application having access to the Internet can have numerous privacy implications, mainly because this resource allows applications to leak data to remote third parties. However, in 2014, Google downgraded this permission from the dangerous to the normal category, which means that Android no longer warns users when apps access the Internet. If a user wants to check if an app has this permission, he needs to go to the device settings and read the details of the application. Google justified this decision by stating that almost all apps nowadays access the Internet [43]. Meaning this permission was not a good marker to distinguish dangerous behavior from typical.

Back in 2004, Lederer et al. [39] published a widely cited article that aimed at helping computer developers with how they handled users' privacy. In it, they identify prevalent malpractices developers bring to their software design when addressing end-users' privacy. We should emphasize the importance they gave to software informing users on the possible extent of their data processing. They state, "Designs should not obscure the nature and extent of a system’s potential for disclosure. Users can make informed use of a system only when they understand the scope of its privacy implications." and "Designs should not forgo an obvious, top-level mechanism for halting and resuming disclosure." While Android's permission system can somewhat tackle the latter, it still lacks ways to address the former. Permissions do not inform the user of how an application process his data. Instead, they present information about the resources it has access to, e.g., the Internet or the SD card.
A more recent survey, authored by Eling et al. [13], backs Lederer et al. [39] arguments with statistical data. This study shows that apps that complement Android’s permission system with additional mechanisms to provide users with more control and feedback on how their sensitive data is treated are more likely to be adopted. For example, the ability to prevent the disclosure of personal information to third-party advertising companies. By contrast, apps that only inform users that they sell data to advertisers were less popular among the studied group. Figure 2.4 displays some statistics obtained from this survey results, correlating the awareness that users have about apps’ ad policies (X) with their installation rate (Y). Colors distinguish the user’s control over data disclosure to advertisers. These results suggest that with a finer-grained permission model, users would be more willing to explore the applications market, knowing that Android protects their data according to their personal privacy policies.

Felt et al. [12] performed two studies that aimed at understanding how sufficient Android’s permissions are at presenting comprehensive information to users on how applications treat their privacy. They found that, while Android gave participants some awareness of the potential behavior of applications, most of the users did not pay attention or understand what the permissions warnings meant at all. This article also argues that permissions should convey information about the potential risks involved, instead of listing the resources they grant access to. The authors state that the current way the Android permission system is designed does not bring enough clarity to privacy risks and may create misconceptions for users when they attempt link an application accessing a resource with perils for their privacy.

**Misconceptions about data-treatment:** Balebako et al. [3] conducted a survey on 19 participants in the US to study awareness and misconceptions smartphone users have about how apps treat sensitive data. They used popular free games available on the Google Play, which acquire and yield personal information about the player for target advertisement purposes. They found that the majority of the group did not even understand that these games would share their data. After being informed, an even bigger fraction revealed uneasiness with whom their information ended up with. Another study [44] shows that users have more concerns regarding their privacy when using a smartphone, rather than a computer. These fears are aggravated when using free apps. Furthermore, it suggests that cases, where personal data is gathered and sold are more common between users who are not aware of it.

The statistics presented in these papers support the claim that the mobile platforms still lack the means to convey to users what explicitly happens to their personal information. Furthermore, while application developers may be legally exonerated by showing their privacy policies, users may still have to deal with the consequences of having their personal data being exposed unknowingly.

**Summary:** The research discussed in this section demonstrates that there are still some open significant privacy problems in the Android environment. First, Play Protect [8] is not a failproof mechanism at detecting malicious applications, which leads to users unknowingly installing them. Second, over-privileged apps open the door for malware exploitation, either by third-party libraries or cloned apps. Last, the current permission mechanism is too coarse, and is not enough for users to make informed decisions about how applications actually treat their data. In the sections that follow, we present some
proposed solutions that aim at improving users’ perception of their privacy and allow them to exert more control over how their applications access their sensitive data.

2.3 Android Apps Security Analysis Tools

Several proposals try to mitigate the current problems regarding privacy in Android devices, by analyzing applications, how they access data, and if they leak it. There are several tools developed for Android that aim at improving user's knowledge about their privacy treatment by applications by inspecting their behavior either statically [14–17] or at runtime [18, 19]. Although these tools do not grant users additional control of their data, they do improve their understanding of how sensitive data is treated, and why an app requires specific permissions.

Susi: Susi [45] is a tool that is able to identify with high precision all data sources and sinks the Android API grants to applications. Because Android itself has numerous and ever changing functionalities, parsing its API to classify its methods into sources and sinks comes as a too burdensome approach. What Susi does, instead, is to run a machine-learning trained algorithm that sweeps the Android source code, categorizing its methods into data sinks, sources, or neither, based on the names of the methods, arguments and return variables. Despite not relating its findings with privacy concerns, Susi allows other tools to identify applications’ dangerous behaviors. Many of the mechanisms discussed below, use Susi’s categorization of native Android methods to identify potential data leakage.

FlowDroid: FlowDroid [14] aims at providing users with information about potential data leakages performed by applications. Its runs an algorithm that performs static taint analysis on Android apps. It obtains a list of Android's API data sinks and sources methods from Susi. Flowdroid starts by parsing the application bytecode and modeling its instructions into call graphs. Then it attempts to identify possible data leaks by running a procedure based on the Interprocedural Finite Distributive Subset (IFDS) framework [46]. IFDS relies on an algorithm that implements interprocedural dataflow analysis, which is able to model how data flows between functions, variables, and recursive calls.

Consider Figure 2.5 as an example that displays how FlowDroid adapts the IFDS algorithm to run its own taint analysis. First, it sweeps the application in search of a call to a data source method. When...
found, it memorizes $w$ as the variable in which the data originated from the source was stored (1). The application then assigns $w$'s value to a field of one of its existing variables, i.e., $x.f$ (2). Since $x$'s already existed in function’s scope, the algorithm triggers a backward search to find its origin (3). Because $x$ was not originated in the function itself, but rather assigned from a field of a variable passed as an argument, i.e., $z.g$ (4), the algorithm performs a backward analysis of the call graph. This analysis consists of searching for previous operations involving the variable $z.g.f$ (4), in the caller function. Notice that the variable $z$, in this call graph, is referenced in main as $a$. Therefore, the algorithm continues moving backward, tracking operations that affect $a.g.f$. While doing it, it notices that $a.g$ is assigned to $b$, and thus also adds $b.f$ to the list of variables of interest. Because the algorithm runs into $a$'s creation, without finding any of its fields having values assigned from other variables, it must mean that $a.g$, and $a.g.f$ were also created along with $a$, meaning that there are no more references to $a.g.f$ before this statement, and thus the forward analysis resumes (7). Next, the algorithm identifies a sink method called with $b.f$, which will trigger a data leak warning, as the variable carries data obtained from a source of sensitive data.

Android apps do not have a main method. Moreover, their components’ execution order is dependent on user interaction. This hinders the data-flows analysis procedure, as the standard IFDS framework was designed to analyze programs that always behave the same way, with the same order of function calls. The fact that Activities – a type of component with UI elements – internal behavior and methods calls are also reliant on the user operations, only accentuates this problem. FlowDroid addresses this issue by assuming all possible combinations of sets of user-independent execution flows within a simulated main method. After examining an app, FlowDroid outputs a list of possible data leaks the applications may have, which the user can read and then decide to keep or uninstall the application.

While being a high precision tool for identifying dangerous operations, FlowDroid only taints data as direct assignments between chunks of memory. It falls short in dealing with implicit flows, for instance, in cases where the user’s data is not leaked directly through the original data bytes, but by new information constructed from inferences from the data through control flows. A good example would be an application retrieving the user’s location to leak a boolean stating whether he is or is not within a commercial area. FlowDroid will fail to taint this boolean, and consequently analyze how it flows through the app, as it is not a direct assignment from the location variable.

**IccTa:** Li et al. [15] propose another taint analysis tool, which extends FlowDroid’s approach. Its novelty is in being able to identify Inter-Component Communication (ICC) operations to construct a more faithfull model of the application data flow. Note that while FlowDroid considers the communication between components, it does by considering all the possible sequences of interactions. By computing the flow paths for all individual components, and then merging them in all possible combinations, it generates many false-positives. Contrarily, IccTa’s approach takes into consideration how the app’s components interact both with each other and other apps. It then recreates a graph containing all possible flows conditioned by inter-component interactions. Despite presenting higher accuracy at detecting leakages of sensitive data, IccTa still can not detect leaks through control-flow operations.
**AppScalpel:** AppScalpel [16] is another static application analysis tool. It can inform users about potential concerns regarding their personal data treatment by combining FlowDroid reports with statistical information it gathers from both trustworthy and suspicious applications. The former category is composed of the top 300 free apps, while the latter is composed of 4000 random malicious applications identified as such by malware database services. Users can analyze the application using AppScalpel’s Android app. This app creates a report carrying some information about how their data might be treated. Additionally, AppScalpel shows users how inspected apps rank in terms of good practices.

Since AppScalpel uses statistic analysis as its base to rank applications, new mischievous practices may go undetected, and thus a database needs to be kept updated in order to produce good results, much like standard antivirus software. Nevertheless, because it shows users qualitative rank information on the analyzed app, AppScalpel is useful for users to understand the privacy risks of using the app.

**MutaFlow:** Mathis et al. [17] present another leakage static analysis tool for Android applications. It detects data leaks by running multiple tests, mutating the possible values of sensitive data delivered to the application, and observing how the data sent to the outside change. For example, it detects if an application leaks the user’s location by first running multiple tests, each changing the city names provided to the application by simulating Android’s location service. Then, it verifies if the data that flows into the sink methods invoked by the app change between the tests made. If so, then there is indeed leakage of the user’s location. Similarly to FlowDroid and IccTa approaches, MutaFlow gets the list of possible sink or source methods from the outputs provided by Susi.

While this approach can identify control-flow data leaks, its detection methods can be deceived. For instance, an app may act inoffensively during the first few hours of usage after being installed. During this period, a user runs MutaFlow to scan the app, which does not detect leakages. Later on, the same application may send out sensitive data without the user being aware.

**TaintDroid:** [18] is a widely cited Android extension that helps users inspect how applications treat their data. In contrast to other tainting mechanisms already referred to in this section, TaintDroid performs dynamic taint analysis. By doing so, it can alert users when applications perform some dangerous behaviors, like sending their data to a cloud server or another Android application.

Figure 2.6 shows TaintDroid’s architecture. Its approach consists of modifying both the system’s native Binder IPC library, which applications use to perform IPC communications (as discussed in Section 2.1), and the Dalvik VM interpreter that implements Android’s Java runtime environment. The Binder library was changed to track how data travels between processes. In turn, the VM interpreter taints the data that passes through native system libraries’ sources and sinks of sensitive information. By combining both methods, TaintDroid is able to control how data flows through the application and warns users when the app attempts to disclose sensitive data outside its process.

Contrarily to previous discusses taint-analysis tools, TaintDroid does not need any statistical information nor to perform tests to detect if an app leaks sensitive data. It can, in fact, detect leakages while they happen. However, while having a fair performance overhead considering its benefits, TaintDroid
has its own shortcomings. As many previous discussed approaches, it still does not track implicit flows. For example, the information contained on the user’s phone number may be indirectly leaked through a set of cascaded if, elses that match each digit against a possible value. This method would allow the app to reconstruct the phone number without directly assign it from a data variable. This sensitive data could then be sent to a cloud server without TaintDroid knowing. Another significant restriction with its approach is the need to modify an already shipped and widely used OS, which difficulties its generalized adoption by users. The solution we propose does not suffer from this problem.

As of the time of this writing, TaintDroid is considered to be obsolete. In 2014, Android swapped its runtime engine from the Dalvik VM to Android Runtime (ART). Dalvik VM compiles apps into machine code at runtime, imposing a significant performance overhead. ART, on the other hand, employs Ahead-of-Time compilation (AOT) of applications, which means that apps are compiled into machine code when installed. This resulted in increased performance and lower battery consumption. However, TaintDroid was developed by changing Dalvik’s core system, which made it have incompatibilities issues with newer Android versions. TaintART [19] adapted TaintDroid for modern Android devices, leveraging from ART improvements over Dalvik to enhance the performance of its algorithms.

**Discussion:** The research work presented in this section presents the state-of-the-art for application data flow analysis in Android. The main goal of the proposed solutions is to inform users about possible leaks or dangerous behavior by applications when treating sensitive data. Most of these approaches [14–16, 18] are based on taint-analysis of code. These provided high accuracy results on their testing cases, but still let control-flow leaks go undetected. MutaFlow [17] detects this last type of leakages, but its approach consists of running tests against each app, which can be tricked by a malicious developer.

However, some [14–16] rely on static taint-analysis of applications’ code, which is not always effective at detecting runtime data leakages. A study carried in 2018 [47] shows that FlowDroid and IccTA fail to track flows of sensitive data when they exchanged through ICC calls in the form of complex Strings.
TaintDroid [18] and TaintART [19] achieve better results at detecting leakages, but require changing the Android OS. Furthermore, these tools rely on taint-analysis at a variable level to track how data flows within applications, which can be deceived by a malicious developer [48].

Moreover, the systems discussed in this section only aim at informing the user on privacy leaks and do not provide one with actual methods to enforce privacy policies. Some other solutions give users more control over their data by extending Android’s permission system, which we discuss next.

### 2.4 Android Permission Enhancement Mechanisms

In this section, we present some proposed solutions that give users more control over their data by extending Android’s permission system and how they can be utilized to preserve privacy on Android.

**Apex:** Nauman et al. [20] present Apex, a policy enforcement framework which aims at improving Android’s permission system by adding the concepts of contextual permissions. Apex allows users to change what application can access at run time, with a fine-grained set of constraints they may use when defining a privacy policy. For example, a user can decide that a certain app can only access his location during the daytime, or even restrict the number of SMS messages the app may send. Apex modifies Android’s core to change the default Android app installer to a new system application named Poly, with access to Android’s permission system internals. Figure 2.7 shows Poly’s interface. Users can access Poly either when installing a new application or by opening it directly from the device’s home screen to change the permissions given to an existing app.

While Apex gives users a more fine-grained control mechanism over their sensitive data, users are not informed about how applications threat their data. Furthermore, because it demands modifications to Android’s code, the widespread adoption of Apex is severely limited on non-rooted devices.
Mockdroid: Beresford et al. [21] modified the Android OS in order to make its permission system support mocking of sensitive data. Differently from Apex, it departs from contextual permissions and implements data obfuscation. Mockdroid includes a trusted system app in which users can state, for instance, specific GPS coordinates, which will always be passed to a certain app, regardless of one’s actual position. SemaDroid [22] pushes this concept a little further by also letting users control the quality of sensor data acquired by apps. For example, it can lower the sample rate of an audio recording.

Mockdroid’s approach brought an interesting concept for obfuscating data. However, users can not specify under which circumstances they want their data altered, and the mocking values are always the same, statically defined on a settings window. Moreover, it also requires users to root their phones.

ipShield: Chakraborty et al. [23] present a framework for Android devices that allows users to enforce real-time privacy policies over what sensitive data applications may collect. These policies are formed from a set of changeable rules that define what sensors’ data an application may access, how, and under what circumstances. Similarly to Apex and Mockdroid, it was conceptualized to improve Android’s permission system by introducing some innovations at controlling access to sensitive data.

Whereas the permission system fails at conveying privacy implications of granting apps with access to specific resources, ipShield does present to users the risks associated with each sensitive data type an application attempts to obtain. With ipShield, users can define context-based rules to not only control the circumstances in which apps can reach users’ data but also obfuscate the data itself. For example, Alice can decide that a specific application may access her location at any time of the day, except if she is in a particular pub. In this situation, ipShield provides a fake position to the app, e.g., her friend’s house, or her college campus. In this sense, ipShield’s policy constraints are tailored for each data type.

Figure 2.8 shows a purely illustrative example to give an idea of how ipShield works. This framework is implemented by changing Android’s native system services to include obfuscation capabilities while
controlling accesses from part of applications with context-aware rules defined on a trusted system app. This app also presents users with policy recommendations for each kind of sensitive data. It does it by maintaining a database relating the category of the data with the possible inferences the app may obtain from it. For instance, location data can be used to detect if the user is at a club, at home, or a hotel. These possible inferences are then presented to users, so they can be aware of potential privacy consequences before deciding if the app should access that data type. If an inference is black-listed by the user, the user is advised with possible obfuscation rules, e.g., announcing that the user is at home when, in fact, he is at the club. Users may leverage these recommended rules or write their own when defining the data disclosing policy to each app. The Firewall service notifies other system services of these rules, which then control the data they disclose to the app accordingly. Thanks to this way of conveying information and control, ipShield allows users to perceive privacy implications, and better address them. However, this system still has some limitations. First, it loses track of what happens to data once applications obtain it. Second, it also requires the modification of the Android OS.

BinderFilter: Wu and Bratus [24] developed a framework that tackles privacy concerns similarly as ipShield. However, while the ipShield enforces users’ privacy rules on the system server process, BinderFilter resides at the kernel level and modifies Android’s IPC Binder to enforce security policies when exchanging messages between processes. Users may define these policies as a set of contextual rules in a system app called Picky, as shown in Figure 2.9. For instance, in the example shown in Figure 2.10, whenever an application issues a request to access the user’s location, internally, BinderFilter iterates over the current user policies, and alters the data accordingly. If the user wants to obfuscate his location, then BinderFilter changes the string to some other value. If the user blocked the app from accessing his data altogether, then the string’s bytes are set to zeros. This system, however, suffers from the same limitations as ipShield – it just controls how apps access data and requires changing Android’s core.
**Aurasium:** Xu et al. [25] propose another privacy monitor solution that gives users a more fine-grained control than the native permission system. Contrarily to the previous approaches we discussed, Aurasium does not modify Android’s core. Instead, it relies on intervening between the developer’s code and both the Android Framework and native system libraries by injecting the application with their own proxy code. This is achieved by displacing the installation flow from Google’s Play Store and forwarding it to Aurasium’s cloud service that decomposes the application, injects their code and then repackages and re-sign it. Then the new app is downloaded and installed into the user’s device.

Once the user runs this modified application, this tool gains control over the native app’s behavior by making use of Android’s API Application class. The onCreate method of this class is called before any component life-cycle starts, thus giving Aurasium’s code a chance to control the application’s context. Then it can force their native proxy libraries to replace the original ones which Android linked with the application runtime. These libraries are entangled with privacy mechanisms that attempt to give users more control over the application behavior. As an example of fine-granularity control, Aurasium can prompt users when the application attempts to perform a cloud call to an unknown domain.

The main limitation of Aurasium is that it relies on repackaging applications, breaking their original author’s signature. This system also does not track how data flows within applications. Furthermore, Aurasium’s strategy changes the dynamic-linking process at runtime, which is compatible with newer Android versions, due to new security policies enforced by Android’s development team [49].

**AppGuard:** AppGuard [26], similarly to Aurasium, also repackages applications to inject them with a privacy monitor, which enforces defined policies when they attempt to access some system call. However, additionally to the problems created by repacking apps, AppGuard does not monitor native code calls, which can be exploited to disable its security classes by a malicious application.

**AppFence:** AppFence [27] uses TaintDroid dynamic taint analysis mechanism to impose additional control on sensitive data disclosure to environments outside the device. It categorizes some data types as sensitive, and allow users to define restrictions to each app when operating with these data types. When TaintDroid’s system detects a leakage, either through the Internet, the file system, or IPC mechanisms, AppFence blocks the communication. Additionally, this system can employ data shadowing of some data types to prevent applications from accessing the actual user information. In practice, shadowing a value means replacing it with a false one, which can be based on the original. For instance, AppFence can replace the user’s contact list with one containing just entries that the user considers safe to disclose, e.g., his work contact list.

**Kynoid:** Kynoid [28] also uses TaintDroid [18] to identify data leakages, and lets users define fine-grained privacy policies associated with each data type. These can specify, to whom the app may disclose the data it obtains — for instance, a list of IP addresses which the location data may be shared with. Kynoid enforces these policies when tainted data attempts to leave the app’s domain through native code calls, e.g., by performing cloud calls or writing to the file system.
**Discussion:** In this section, we looked into research made to improve Android’s permission system. Almost all the studied solutions [23–26] only control how applications access data and do not consider how it flows within the app. Therefore, users can not determine, for instance, what web servers their sensitive is sent to. The systems that control how data flows within applications [27, 28], rely on Taint-Droid [18] to detect leakages of data, which means they inherit its limitations, e.g., being vulnerable to motivated malicious developers, and having incompatibility issues with newer Android versions.

Furthermore, we found that all but two solutions [20–24, 27, 28] involve changes to Android’s core. Aurasium [25] and AppGuard [26] do not modify the OS, but instead, rely on dynamic instrumentation of applications. Both of these techniques interfere with Android’s ecosystem and raise compatibility problems, which makes it harder for the general population to adopt these systems. Another prevalent problem with the studied solutions is that they all control access to data at a very low abstraction level, such that it becomes hard for users to understand how applications use their data and for what purposes.

We envision a system that aims to address both these problems. To achieve this goal, next, we explore some ideas proposed to protect users’ privacy in smart home environments.

### 2.5 HomePad

Zavalishyn et al. [30] propose a new way of handling privacy in smart homes. Most modern smart apps run most of their computation on cloud servers. For example, the popular SmartThings platform [50], allows developers to create apps to interact with smart home sensors and actuators, e.g., lights, fridges, cameras, and thermostats. These apps run on a cloud server, and users can interact with them with a client on their mobile phones. For an application to operate, all data collected from the smart home sensors need to be sent to the cloud, which means users need to give away their data into an untrusted environment. In HomePad, on the other hand, applications run on the smart home’s hub.

What is most relevant for our work is that HomePad applications are developed following a privacy-aware programming model that makes it explicit to users what functionalities are processing their data. This allows HomePad to present the user with incisive information regarding each application’s functionalities, such that the user may perceive what applications do to their data, and if they threaten one’s privacy concerns. The user can then make an informed decision to whether or not disable some of the app’s behaviors according to his privacy preferences. Naturally, the ability to define privacy policies for the app’s internal functionalities, allows the user to have much finer-grained control over his sensitive data than only determining what data the whole app has access to.

HomePad apps must be structured as a directed graph of multiple connected elements. Elements are functions that implement some well-defined and specific behavior. The edges that connect them express the possible data paths, as HomePad only allows data to flow between connected elements. Consider the example in Figure 2.11, which shows an application installed on a HomePad hub as part of a surveillance system. The app uploads camera frames to a cloud server when it identifies a person at the front door of the house. The application is implemented by three interconnected elements. The FromCamera, FaceBlurrer and ToCloud elements are part of HomePad’s native API, and thus deemed
trusted. However, the developer needs to make the app run his own proprietary algorithm that detects a human shape in a camera frame. He is able to achieve this by creating an application-specific element – Person Recognizer – in charge of this task, and making it run between receiving data from the camera and sending it to an Internet server. HomePad considers the Person Recognizer element untrusted. However, thanks to the way the graph is connected, HomePad knows that the camera frames can only be sent to the cloud through the ToCloud element. Because this element is trusted, HomePad can inform the user which cloud servers the app is able to send camera frames to, and let him decide to block the app from communicating with some of them. Furthermore, because the application uses the FaceBlurrer element connected to the FromCamera one, HomePad can also inform the user that the camera's frames are only sent to the cloud after being anonymized.

**Fine-grained privacy policies:** HomePad supports a variety of privacy policies that the user can use to control how his sensitive data can flow through applications. In the example mentioned above, one could make a privacy policy to assure that no installed application can leak camera frames to the cloud without them being blurred first. While the application in question satisfies this privacy policy, other installed apps that do not use the FaceBlurrer element will be blocked from running, or possibly altered to meet the policy criteria. Additionally, the user could define specific cloud URLs, which no app may send photos to. HomePad would then configure the ToCloud element to not interact with those addresses.

In comparison with the studied proposals to improve Android's permission system that only allow the definition of access-control privacy policies [23–26], HomePad enable the user to regulate what apps can do with his data after they collect it. While AppFence [27] and Kynoid [28] allow the user to define URLs where sensitive data can not be sent to, they are vulnerable to implicit flows of data. They also do not inform the user if his data is anonymized before being sent to the outside. HomePad is immune to leaks by implicit flows since it tracks data-flows between blocks of functionalities and not variable tainting. It also informs the user if the app anonymizes his data before disclosing it.
Developing apps for HomePad: Developers need to explicitly declare, within a manifest file, how the application graph is structured, i.e., all the untrusted and trusted elements it is composed of and the edges that connect them. HomePad's programming model makes developers design applications around data flows, and thus helps them identifying potential privacy violations when treating users' data.

Developers may access a smart house resource's API by importing its respective module on their applications' manifests. A module is a package containing multiple trusted elements, each to control a device's functionality through its specific element driver. Trusted elements can be used to receive data from sources, send data to sinks, and transform data in a specific and trusted way. Developers need to connect these trusted elements in a way that reflects the sequence of functionalities they want their apps to run. Developers can implement new (untrusted) elements to run application-specific algorithms and connect them to other elements. While the implementation of untrusted elements gives developers the flexibility to customize the application's functionalities, only trusted ones might receive data from sensors, or send it to the outside. This implies that, no matter what graph the developer designs, ultimately, he needs to send the data with the help of a trusted element, which only sends data to other allowed parties that HomePad configured according to the user-defined privacy policies. As an example, these configurations can be web servers URLs or Bluetooth device's mac addresses.

Architecture: HomePad is designed as a middleware (presented in Figure 2.12) implemented between apps and the hub's operating system. It has a runtime environment in which applications are executed, a model checker that verifies compliance of the app's data flow against user-defined privacy policies, a configuration manager where users can manage the installed apps, and an extension manager to maintain the Hubs extensions – i.e., other trusted elements not shipped natively with HomePad.

The configuration manager also loads applications into the runtime environment, instantiating their trusted and untrusted elements according to each enabled app's manifest, and the user's privacy policies. Trusted elements are loaded from HomePad's internal safe repositories, and can not be altered by application developers. The runtime engine has an event bus that processes messages between elements (layout events) and between trusted elements and their respective drivers (global events). The trusted elements drivers are the only entities that have direct access to system drivers. This way, HomePad assures exclusivity to all data sources and sinks, like sensors and cloud communications.

Implementation: HomePad was implemented in Java, to run on a dedicated computer. Each supported app must also be written in Java, both its untrusted elements implementations and its graph manifest. When HomePad launches, it loads all configured and enabled applications through the configuration manager, which was implemented as a Java class with the framework's main method. When starting, it initializes the runtime environment and loads all necessary element's classes into it. Then it instantiates all required elements, the untrusted ones within a sandboxed environment with the help of Java's Security Manager [51]. The event bus runs on a dedicated thread to process events, meaning listening to newly dispatched ones and forwarding them to the appropriate receivers.

Users can define privacy policies through an Android app that communicates with the computer
running HomePad. The Android app sends the policies as Prolog rules to HomePad, which, based on these, restricts how its trusted elements disclose data to untrusted ones or, in some cases, disable apps altogether. HomePad uses SWI-Prolog [52] as its prolog engine and presents to the user, graphical representations of the multiple application graphs with the help of the Graphviz tool [53].

**Discussion:** Within the context of smart homes, HomePad provides a safe hub to run applications. This system displays novelty on the fine-grained information it presents to users about how apps threat their data once they collect it, and can even disable specific functionalities that go against user-defined privacy policies. Furthermore, because HomePad applications are structured as a directed graph of elements, developers are also able to observe how their end-users’ data flow through their apps, which helps them understand the privacy implications of their design choices. HomePad was also implemented without changes to the computer’s OS, which facilitates its adoption by users.

These benefits brought by HomePad could help with the current privacy problem in Android devices. For instance, informing a user that a certain application will obtain one’s camera photos and send them to specific cloud servers carries much more valuable information than just saying it accesses the device’s storage and the network. Furthermore, HomePad’s privacy policies are not aimed at applications as a whole, but at their individual functionalities. These policies are naturally more fine-grained control measures than Android’s native permissions and could give the user more control over how apps process sensitive data. For instance, a user could disable an app from disclosing his photos to particular Internet addresses, which could be enforced by HomePad disabling the corresponding data flow, while leaving the rest of the application, including other flows that communicate with the Internet, operational.

Due to the large control options, Homepad provides users with. Our work tries to solve some of the current privacy problems in Android by adopting some of HomePad’s ideas, in order to give users fine-grained control mechanisms over their privacy, without modifying the Android OS.
2.6 Summary

In this chapter, we described how Android’s security mechanisms work and how they protect the user from malicious apps. Then, we presented privacy problems still relevant to the Android environment, some because due to the permission system coarseness, others due to malicious apps still accessible to users, and others due to users’ misconceptions of privacy. Then, we presented existing proposals that aim to solve some privacy concerns in Android. Some by analyzing and reporting dangerous application behaviors, others by extending the Android’s permission system itself to become more fine-grained. Finally, we discussed HomePad and how its privacy-aware programming model can enhance applications’ design and give users more control over their privacy. Next, we introduce Crystalline, our system that grabs HomePad’s concepts and molds them into an Android compatible framework.
Chapter 3

Architecture

In this chapter, we present Crystalline, a framework for the development of privacy-aware Android applications. We describe our we designed our system, the challenges identified while doing so, and the solutions we found to address them. First, we present our design requirements in Section 3.1. We specify the threat model under which Crystalline operates in Section 3.2. Then, Section 3.3 presents the overall platform architecture. The challenges that our architecture solves are stated in Section 3.4, followed by an explanation of how our framework allows developers to create Android applications (Section 3.5). In Section 3.6, we present the runtime architecture of our platform, and how the user’s privacy policies are enforced upon applications. Section 3.7 describes how users can use the Crystalline App Manager to install applications and define their personal privacy policies. Section 3.8 elaborates on how Crystalline changes the way apps are built and published to the app store, and explains the certification mechanism employed by Crystalline to verify the integrity and authenticity of applications.

3.1 Requirements

Our main requirements for the design of Crystalline are as follows:

Provide a fine-grained privacy control mechanism: We want to provide users with more feedback and control over their sensitive data. Crystalline must present the information regarding how each app handles sensitive data in a way that users understand and may make informed decisions upon. We also want to provide the means for users to define fine-grained privacy policies specific to each app. These policies must determine how an application may process the user’s sensitive data, without restricting more functionalities than they need to.

Provide an accessible development framework: We want to allow Android developers to create Crystalline apps as close as possible to the traditional Android programming environment. We also want to maintain the additional programming effort required at a minimal level. An over-complex development framework would be a hard obstacle for the generalized adoption of Crystalline by developers.
Achieve a fair performance impact: Crystalline must not impose a high impact on applications’ performance. The improvements we want to bring to the current privacy management mechanisms in Android should not be burdened by a high-performance overhead, as this would severely limit the general adoption of our framework by both users and application developers.

Guarantee applications’ compliance with privacy policies: Our platform needs to uphold against malicious developers, which may attempt to circumvent the data-oriented programming model rules in order to perform unauthorized operations with the user’s sensitive data. When users define their privacy policies, Crystalline must make sure that applications comply with them. This demands Crystalline to enforce security measures against possible attacks from motivated malicious developers.

Not change the Android OS: Most proposals to improve privacy in Android environments rely on changing Android’s core. This hinders their adoption by the wide Android user-base, as it requires rooting the phone and installing a modified OS, which is a complicated process, voids the device’s warranty, and likely brings compatibility issues with the specific hardware for each device. Crystalline must design without changes to Android so that users can install and use without these concerns.

3.2 Threat Model

We assume that both the hardware and the operating system are trusted, the mobile devices run with an official version of Android, and users only install applications from Google Play, guaranteeing that their apps come from authenticated sources. However, Crystalline must be secure against potentially malicious application developers. Their intentions may range from creating upright applications with data treatment perceivable by their users to attempting to find and exploit vulnerabilities in our platform so they can deceive users by hiding malicious data flows from their knowledge.

Developers may try multiple methods to leak data from the user’s mobile device. An attacker may also attempt to fool the user into installing an application that fakes being backed by our framework or even develop an app that impersonates a well-known legitimate one. Additionally, a developer may publish an upright application, but later add malicious code through an update.

We do not, however, concern ourselves with mischevious behavior applications may have, as long they transparently expose how they handle sensitive data, and respect Crystalline’s middleware authority. It is not the scope of this work to ensure that apps do not present unfriendly content like violent or sexual images, as well as repetitive and irritating advertisement, poorly designed UI, slow performance of tasks, or even application crashes. Developers themselves should consider all these topics, and take responsibility for the way their applications are designed and implemented. Furthermore, our framework does not aim at protecting the user’s privacy after sensitive data leaves the device. Our work is only focused on Android environments, and does not cover cloud-based services, and if they oblige by their privacy agreements or maliciously sell the user’s data to others after obtaining it.
3.3 Overall Architecture

Figure 3.1 presents the overall architecture of our framework. Crystalline provides a trusted medium between users and application developers. Developers are responsible for implementing their applications using Crystalline's development framework. The resulting application package is then published on both Google Play and registered with Crystalline's app certification service. Users can then install submitted applications on their devices using the Crystalline App Manager, which will explicitly inform the user about how the apps will treat his privacy while also allowing them to define personal privacy policies for each app. Then, when the user launches the application, our runtime middleware guarantees that these policies are enforced at runtime. Crystalline is divided into four blocks: A development framework, a certification mechanism, a trusted app within the user's Android devices, and a runtime middleware within each Crystalline supported app.

**Development Framework:** Developers can create Crystalline applications with the help of the Proxy Library. It contains a modularized and extensive Java API that allows them to build Android applications following HomePad's programming model and thus leveraging from inherent transparency regarding personal data treatment. The API supports the native Android functionalities but acts as a proxy in the sense that it does not directly access Android resources, but interacts with Crystalline's runtime middleware, which translates its commands into Android API calls, while enforcing user's personal privacy policies.

**Certification Mechanism:** The cloud-based Certification service runs within a trusted environment, and verifies if apps are correctly sanitized. It also serves as repository in which users can find trustworthy apps, which passed the verification process. We designed a certification mechanism which itself is
divided into two parts: The Code Transformer runs on the developer's computer, and sanitizes code in order to ensure it complies with our framework rules.

**Runtime Environment:** Each Crystalline app runs on top of our middleware, which controls its execution and enforces the user-defined privacy-policies. The App Manager module audits applications when launching, providing the runtime middleware with these policies, and notifies the user when a trustworthy app is running, so one can distinguish between the apps which are supported and the ones that are not. Each app's runtime middleware also controls the app's graphs execution by translating graph events into Android instructions and Android events into graph events.

**App Management System:** Crystalline has a component that runs directly on Android devices, in which users can trust to 1) install honest apps on their devices, and 2) enforce their privacy policies to these apps. The App Installer provides users with a safe gateway to install trustworthy apps by being in synchrony with the Certification Service's apps repository. Users can open the Privacy Manager to inspect the installed apps and enforce personal privacy policies to each of them.

### 3.4 Architectural Challenges

Android users (and developers) can benefit from HomePad's ability to expose applications' internal data flows in a transparent way. However, there are several incompatibilities between HomePad's design and the nature of Android itself, which complicates its adoption in Android-based mobile devices. Throughout the remaining of this section, we explain the challenges we face that lead us to devise the architecture presented in Figure 3.1.

**The architecture of Android apps is different:** We can think of HomePad apps as simple handler functions that operate when sensors’ data are available, and their execution order does not depend directly on human intervention. In contrast, Android apps are usually more elaborate than smart home applications: they implement more complex functionalities and have user interaction at its core. Therefore mobile apps need to change their internal operations according to user behavior. Android’s programming model was designed to divide applications into components, each handing a single responsibility, whether it being user-interactive operations (handled by Activities), or background tasks (handled by Services). An Android component must be abstracted out from the rest of the application, such that the application may launch its components in different orders, and replaceable between one-another at runtime. For example, a real-time messenger application must have an Activity to authenticate the user, another to display his contacts, and another to show the proper chat history shared with a contact. In this case, the user may change the active Activity as it navigates through the app. HomePad's model of having the multiple application's elements all within a graph can not be directly applied to this programming paradigm. Therefore Crystalline should adapt HomePad's programming model into an Android-compatible development framework.
Controlling applications at runtime: From the OS perspective, HomePad is an application running in its process. HomePad installs and launches apps under its own process environment. However, Android apps must have a manifest file that states the components that comprise the application. This file is immutable after the app is installed, which means a trusted Crystalline Android app can not install new applications, that require their specific components, under its own domain. This restriction forces us to deviate from HomePad’s runtime architecture.

Since Android requires apps to run on directly top of its runtime engine, and we do not want to modify Android’s core, Crystalline Runtime Middleware must be placed within applications, yet act as a mediator between Android and the app’s graphs. Furthermore, to ensure that the user-defined privacy policies are being followed, Crystalline’s Runtime Middleware needs exclusive access to Android’s API. This means gaining control over the developer’s code while being part of the app itself.

Integrity of applications: As stated in Section 3.2, applications are untrusted. Therefore Crystalline can not allow any direct interaction between their code and the Android API, as these could be sources of undetected data leakages. All interactions between applications and the system resources should be visible in the graphs structures and performed by the appropriated native (trusted) elements so that graphs structure transparently convey how the app treats sensitive data. However, since Crystalline’s Runtime Middleware is packaged inside apps, a motivated malicious developer may deploy apps with the middleware code adulterated. Hence, we need a mechanism that lets users install only apps that are certified to run with our middleware uncorrupted. Because we do not want to give up the default security mechanisms of Android, this mechanism must rely on Google Play to install applications.

Making privacy policies user-friendly: As stated in Section 3.1, we want Crystalline to provide an understandable fine-grained privacy control mechanism. HomePad gives smart-home owners a variety of ways to enforce fine-grained control over their sensitive data. However, it requires somewhat technical knowledge to do so. We can not expect mobile users to have the patience to read nor understand all the configuration options they have at their disposal to control how data can flow within each app. Crystalline’s Application Manager needs to transform each app’s dataflows into accessible information and allow users to define privacy policies in a straightforward manner.

3.5 Applications Development Framework

To allow developers to write privacy-aware android applications with our framework, Crystalline’s Development Framework provides them with a set of tools that bring the same functionalities of the Android Framework while forcing them to follow HomePad’s programming model. However, Android forces applications to follow specific rules, which defy some characteristics of HomePad’s programming model. In this section, we explain how Crystalline conciliates HomePad’s paradigm with Android development.
Applications’ structure: Android’s programming model was designed to group the multiple application’s functionalities within components. A component, by design, must be singled out from the rest of the application and handle a single responsibility. This allows applications to treat them as replaceable and interchangeable. In practice, a component might be either: An Activity, to control the application window, and interact with the user; A Service, which handles a well-defined set of application-wide tasks that are accessed by multiple components, e.g., network transactions, I/O procedures, or Bluetooth operations; A Broadcast Receiver, which listens to system events; Or even a Content Provider to allow other apps to share some set of persistent data within our application’s domain. To respect the way that Android are split into these building blocks, we designed Crystalline to support one flow graph per component, contrarily to HomePad’s approach of one per application.

Figure 3.2 displays the structural differences between Android, HomePad, and Crystalline applications. In this example, the HomePad app is composed of multiple elements, each in charge of an operation, and exchange information with each other using layout events passed through the edges that connect them. In this example, the traditional Android app is composed of two Activities and one Service. These components communicate between themselves through ICC interfaces. In this case, the first Activity launches the second Activity, with an argument carrying some data. Depending on user-interactions, the second Activity asks the Service to run some tasks and subscribes to the results.

Crystalline apps respect both HomePad’s programming model and Android’s compartmentalization of the app into components. Each component implements its multiple operations through elements. Crystalline supports ICC through specific trusted elements that can communicate with others residing within different component’s graphs. The Crystalline app represented in the figure has the first Activity’s graph launching the second one through a trusted element. The second Activity’s graph has an element to represent its creation, which forwards the argument passed by the first Activity to the rest of the graph. The second Activity can also access the Service’s functionalities with the help of another trusted
element, configured with the sole purpose of interfacing with that Service. When this Service receives a request, a trusted element is notified, which forwards the parameters to an untrusted one to run its algorithm. The result is handed to another trusted element, which sends it back to the Activity.

**Developing applications:** When developing traditional Android applications, developers need to write a manifest stating the multiple permissions the app requires, as well as the various components that compose it. For each Activity of the application, developers also need to compose an XML file stating the layout, i.e., the design of the window shown to the user. All these steps are also required when creating Crystalline apps. However, with our framework, developers do not write code on a component's class, as usually. Instead, they should write a file for each component, stating how its graph is structured. This is a paramount rule when developing Crystalline apps, as their workflow logic should be all represented by its graphs. This does not necessarily mean they have to write the file manually, as there can be some IDE support feature that provides a graphical panel to drag, drop and connect elements, and generate the file accordingly.

HomePad refers to the file describing the app’s graph as Application Manifest. Since Android already uses this term for the XML file containing the app’s permissions and components declarations, we will refer to graph description files as Graph Descriptors. When writing a graph descriptor, developers may import a variety of modules, each providing multiple trusted elements comprising the API to a resource supported by Android. After importing one, a developer may use any of its included trusted elements to compose the graph. A trusted element represents a specific operation that involves communication with a resource outside the graph’s environment, like loading an image from the gallery or sending an HTTP POST request to a cloud server. Like HomePad, only elements that are trusted (native to Crystalline or provided by certified modules) may exchange information with outside the graph. A Graph Descriptor should explicitly state the following information:

1. The native (or certified) modules which the graph depends on.
2. The modules’ trusted elements that will actually participate on the graph’s composition.
3. Which application-specific (untrusted) elements are also part of the graph, and reference their implementation classes.
4. Enumerate the connections between all elements, defining the internal structure of the graph.

One thing developers do not need to worry about when creating Crystalline apps is requesting dangerous permissions at runtime. Crystalline Runtime Middleware automatically dispatches Android permission requests to the user when a trusted element attempts to run a functionality that requires some permission not yet given to the application. If the user does not grant permission, then the respective element outputs an error message, which other elements may handle.

**Modules and trusted elements:** Crystalline packs support for resources provided either by the Android Framework or some OS libraries in modules within a support library. A module contains multiple
trusted elements that together form a complete interface to access a resource. Each element will be tied up to one single functionality. When importing a module, developers can use any of its contained elements to build their application graphs. Figure 3.3 displays some modules, and the way trusted elements are grouped within them. While not representing the entirety of the modules or elements, the support library provides, it shows some important ones. The View Module has trusted elements to manage UI operations. The Activity Life Cycle Module provides an API to handle Activity life-cycle events. The ICC Module allows the application’s graphs to communicate with each other. The HTTP Module lets applications interact with well-defined cloud server endpoints.

Note that Crystalline’s concept of a trusted element is slightly different from HomePad’s. HomePad defines one as an interface to access either a smart home’s sensor or actuator, or even to run perform some nativity supported algorithms. This system knows well the available devices and the data types they generate or consume, so trusted elements that interact with one of them always handle the same types of data. However, this approach does not cover functionalities that depend on the application’s characteristics. For example, Android apps have different windows layouts, each with specific views that the user may interact with. They can also deploy interactions between different components and communicate with multiple web servers. To address these requirements, Crystalline adds the concept of configurable trusted elements, which still have trusted and well-known behavior, but can also be configured by developers (in each Graph Descriptor file) on some parameters. For instance, a graph may need a way of running an algorithm when a certain View on the screen is clicked. A developer may achieve this by configuring a View Clicked element to attach it to that specific view’s ID. Figure 3.3 represents configurable elements with "< >" around their configurable parameters.

Untrusted elements: Similarly to HomePad, Crystalline deems all elements implemented by application developers, or un-certified third-party libraries, as untrusted. We assume these elements may contain malicious code, and thus Crystalline needs to sandbox each of them to protect users’ privacy. Furthermore, developers can implement whatever classes they want, creating relationships between

---

1 An Android View is an object on which the user can interact with, defined on Activity’s layout XML

Figure 3.3: Some Crystalline modules, and their trusted elements.

<table>
<thead>
<tr>
<th>View Module</th>
<th>View Clicked</th>
<th>Image View SetImage</th>
<th>TextView Text Watcher</th>
<th>TextView Update Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity Life-Cycle Module</td>
<td>Activity Created</td>
<td>Activity Started</td>
<td>Activity Resumed</td>
<td>Activity Stopped</td>
</tr>
<tr>
<td>ICC Module</td>
<td>Launch Activity</td>
<td>Bind to Service Class</td>
<td>Run Service Functionality name</td>
<td>Activity Result</td>
</tr>
<tr>
<td>HTTP Module</td>
<td>GET &lt;endpoint&gt;</td>
<td>POST &lt;endpoint&gt;</td>
<td>PUT &lt;endpoint&gt;</td>
<td>DELETE &lt;endpoint&gt;</td>
</tr>
</tbody>
</table>
them as needed to achieve the functionalities internal to each untrusted element. However, to ensure data only propagates through the graph edges, two untrusted elements are not allowed to share memory. Therefore two elements resolve the same class into different memory references. The mechanism employed by Crystalline to impose this rule differs from HomePad's. Section 4.4 explains our approach.

3.6 The Runtime Environment

At runtime, Crystalline needs to make sure that applications are running exclusively code associated with the elements declared in their respective graphs, following the flow-oriented paradigm and thus respecting user-defined privacy policies. Naturally, if we placed the Runtime Middleware inside Android, this task would be enforced both at a kernel-level and within the system services. However, as stated before, we do not want to change Android, as this strategy would break compatibility with existing devices. The approach we chose consists of placing the middleware code packaged within each app. This way, we allow applications to run directly over the Android OS without modifying it. However, as stated in Section 3.3, this also means we lose control over the middleware and its integrity. Nonetheless, as discussed in Section 3.8, we employ security measures to make sure Crystalline only acknowledges applications verified to both run on top of an unadulterated Crystalline Runtime Middleware, and respect its exclusivity to the underlying Android API.

Runtime Middleware: Crystalline’s Runtime Middleware (depicted in Figure 3.4) is in charge of controlling the application’s graphs. It shares some similarities with HomePad’s, since both have an event bus, trusted elements, and a pool of drivers. Elements communicate with each other through graph layout events, while drivers and trusted elements do it though global events. Each Crystalline app has its own middleware instance, which enforces the privacy policies obtained from the App Launcher service. The Privacy Enforcer does this at launch time, by configuring the trusted elements with rules associated
with policies. For example, it might disable some elements from letting sensitive data enter the graph, or add new ones that transform the data right before it is handed to untrusted elements.

**Crystalline Drivers:** Crystalline drivers are in charge of translating global events dispatched from trusted elements into Android API calls, and Android API events into global events to notify the appropriate trusted elements. Some drivers are in charge of controlling critical native Android functionalities – e.g., Activity Life-Cycle Driver, Service Life-Cycle Driver, ICC Driver, View (UI) Driver – while others control optional features- e.g., Bluetooth Driver, HTTP Driver, File system driver.

**Launching task:** Figure 3.5 displays the launching workflow of a Crystalline app. When the user launches the app, Crystalline’s Runtime Middleware asks the App Launcher for the user-defined privacy-policies so that it may enforce them. When receiving a startup request, the App Launcher checks if the application maintained the same version as when the user defined his or her privacy policies specific to the app. If so, then the App Launcher delivers the policies to the middleware and notifies the user that Crystalline vouches for the app’s execution, meaning that the launching app will respect the privacy policies. In case the app changed its version, then the user is forwarded to the App Manager, so the application’s validity and data-flows can be re-verified.

To gain control of the application when launched, the middleware needs to have authority over the application life-cycle methods invoked by Android. We assure this with the help of the Code Transformer, discussed in Section 3.8. Essentially, Crystalline changes the application bytecode before being packaged, to inject proxy code directly on the multiple app’s component classes. This code forwards Android events to the middleware classes. These events include component’s life-cycle check-points, UI actions by the user, or even system events. For example, when an Activity is created, Android API calls its respective class’s onCreate method. However, thanks to the proxy code injected around this method, the event is handed to Crystalline’s middleware. If this is the first component in the application’s lifetime, then the middleware starts the launching task. If not, then the event is simply forward to the Activity Life-Cycle driver, so the appropriate graph’s trusted elements are notified.
As soon as the App Launcher gives the current user-defined privacy policies for the app to the middleware, the elements are loaded, and the graphs constructed. Every time an untrusted element is loaded, Crystalline sandboxes it so that it may only communicate with other graph elements, and only through layout events. Our sandboxing mechanism relies on changing Java’s internal class loading model, and we detail it in Section 4.3. Every time a trusted element is loaded, the Privacy Enforcer determines its status – enabled, disabled, or “ask first.” For instance, if the user determines a policy that prevents the app from sending gallery photos to http://xyz.com/store_data, then the HTTP Post element in charge of executing the request is disabled. When a trusted element is disabled, it may still receive events, but does not act upon them. The Privacy Enforcer can also add new elements to the graph, specifically elements to transform data before they arrive at untrusted elements.

### 3.7 The App Management System

Crystalline provides two interactive apps for users’ Android devices. The Privacy Manager allows users to check and control how apps treat sensitive data. The App Installer is a safe gateway to install certified apps. In this section, we cover these two apps and how they help users protecting their privacy.

**The Privacy Manager:** Users can inspect how installed applications treat their privacy by opening the Privacy Manager. It examines how the app’s graphs are structured and how data flows through them, by running an algorithm based on HomePad’s. However, we do not display all the application’s data-flows to the user. This is because too much information makes the actual meaningful part hard to perceive. Instead, Crystalline only shows to the user what sensitive data the app collects and where it is sent to.

Some of the research studied in Section 2.4 [20, 23, 24], as well as HomePad itself, support contextual-policies. These allow the user to define conditions upon which apps may access data – like the time of the day, or the WiFi connection status. While the Privacy Enforcer could implement them with some if-else logic, they also make the whole process of defining privacy policies more complex to users. After installing a new app, users may not want to take some minutes to write a set of conditions for each data
type, before actually getting to use the application. Because we want to keep Crystalline simple to use and appealing for users, we consider four privacy policies:

- **Sink-blocking policies**, in which the user stops the data from being exposed to a particular sink. E.g., the user sees that his photos end up being sent to an advertisement server. He may decide to block the app from performing this action.

- **Source-blocking policies**, in which the user blocks the app from obtaining a data type. E.g., the user sees that the app obtains his photos. He may block the app from obtaining this data.

- **"Ask first" policies**, in which the user forces the application to request permission every time it attempts to obtain, or disclose, a particular data type. E.g., the user may decide that an app must ask permission every time it wants to send gallery images to the cloud.

- **Data transformation policies**, in which the user only lets the app send some types of data to the cloud if they are transformed first. E.g., the user may decide that gallery images must be blurred before being sent to a specific web server.

**Inspecting Inter-Component Communication:** One of the aspects Crystalline has to consider when analyzing the multiple app's data-flows is Inter-Component Communication (ICC). Crystalline apps are divided into multiple graphs, each corresponding to an Android component. Crystalline inspects the app's graphs individually, obtaining conclusions about sensitive data processing according to each graph's internal data flows. However, graphs may communicate across each other with elements provided by the ICC module. When graphs perform ICC communications, they may trade-sensitive data between them, extending their internal data-flows outside each graph's scope. To truly capture how data flows within the whole application, Crystalline changes some aspects of HomePad's algorithm, in order to perform a recursive inspection over the application. While inspecting a graph's data flows, if the Privacy Manager finds an ICC element that receives data from a second graph, it places the first graph's inspection on hold and starts inspecting the second one. If there are no circular-dependencies of data-flows, then the data that arrives at the first graph can be determined by purely inspecting the second graph. With the type of data received by the other graph determined, the inspection of the first graph may resume. The explain this algorithm further in Section 4.6.

**The App Installer:** Google Play provides a safe gateway to install applications. When the user installs an app from Google Play, it most likely has no malware. However, in our case, this does not mean the application is trustworthy. Within Google Play, there might be apps that describe themselves as certified by Crystalline, despite not being accepted by our Certification Service. Apps may claim this to convey a false sense of transparency regarding privacy treatment. Therefore, users should not install Crystalline apps by directly exploring Google's Play. Instead, they should use Crystalline's App Installer.

Figure 3.7 displays the installation process of a Crystalline supported app. The App Installer is a client that communicates with the Certified Apps Store – which belongs to the online Crystalline’ Certification Service – to provide the users with certified apps through a graphical interface. We do not
Figure 3.7: Installation process for certified Crystalline apps.

want to give up the security mechanisms Google natively provides Android with. Therefore, Crystalline
does not install apps directly on the device but forwards the user to their installation page on Google
Play. Since updates can introduce malicious code to applications, the Certified Apps Store verifies if the
app requested maintains the same version in Google Play as the one within the database of certified
Crystalline apps. If so, then the installation proceeds, and the App Installer is informed of the app’s
location (ID or URL) in Google Play. Next, the App Installer opens the application’s installation page in
the Google Play app. From here onwards, the install process continues as for normal apps.

3.8 Certification Mechanism

As stated in Section 3.5, Crystalline apps should restrict their workflow to their graphs structure. How-
ever, because they are also Android applications, they may try to run malicious code outside graphs
directly on the components classes. This goes against the data-flow programming model and thus ren-
ders Crystalline useless. Furthermore, Crystalline packs its middleware within applications, in the form
of a library, which means a motivated malicious developer may tamper with the middleware’s code to
bypass the enforcement of privacy policies. The malicious developer may also write code to directly
access Android API classes, which would allow his app to process sensitive data outside graphs.

To guarantee its middleware classes were not tampered with and have exclusivity to direct interac-
tions with the Android API, Crystalline only acknowledges applications that went under a code trans-
forming process – performed by the Code Transformer -, and subsequent verification of its correctness –
performed by the Validator. In this section, we present how Crystalline guarantees that applications both
do not run code outside their graphs and run with the unadulterated Crystalline Runtime Middleware.

**Code Transformer:** The process of transforming an application’s code to assure Crystalline’s runtime
integrity would be enforceable in an online service, by changing the app, repacking it, and then delivering
it to the user. However, this strategy would break the app’s developer’s signature. Instead, our approach sanitizes the app’s bytecode before developer packages, signs it, and publishes it to Google Play. As shown in Figure 3.8, the process of assuring applications’ integrity begins in the developer’s computer. During the app-building process, Crystalline’s Code Transformer tool reads the compiled bytecode and outputs a sanitized version of it. It is in charge of two tasks. First, it needs to assure that no code, besides the one belonging to Crystalline’s middleware, can run outside untrusted elements. Second, it must inject the app’s component classes with code to forward their life-cycle notifications to Crystalline’s middleware. The code transformation process must happen after the application is compiled into bytecode, so it covers all the imported and already compiled third-party libraries. However, it must occur before the developer signs the app, as changing an already signed app breaks its signature validity. This process was envisioned with Aspect-Oriented Programming (AOP) in mind, which allows us to inject security code across all the application. We explain how we implemented this mechanism in Section 4.4.

**App Certification Process:** We can only be sure that Crystalline supported apps follow the user’s privacy policies if they run with our middleware unadulterated, and the Code Transformer correctly processed their code. These two requirements form the integrity criteria applications must meet to become available for users to install, through the App Installer.

After publishing the app to Google Play, developers must register the application with Crystalline’s online Certification Service. The App Registration Client sends the registration form to the App Registration Server. This form must contain enough data to validate the integrity of the application published to Google Play. The Validator performs the algorithmic integrity test of the app, by verifying if the application published to Google Play meets our integrity criteria. It is in charge of checking if the Code Transformer correctly sanitized the app, and that our middleware’s classes were not adulterated. The Certification Service maintains an updated database with all the applications that passed this test. These apps are the ones available for users to install through the App Installer. When a developer updates an application on Google Play, one must also re-register the updated app with the Certification Service, so that Crystalline can be sure that the update did not introduce malicious code to the application.
3.9 Summary

In this chapter, we presented Crystalline’s design architecture. We first established the requirements of our architecture. Then we defined what entities we assume as trustworthy and the ones which we needed to devise protection measures against. Following this, we presented Crystalline’s overall architecture and the challenges of adapting HomePad to Android, which leads us to design Crystalline that way. We carried on by describing how Crystalline apps would deviate from normal Android ones, structurally wise, and how we would provide developers with a modularized API to build compelling apps. Then we presented Crystalline’s runtime environment and how it was devised to enforce the user-defined privacy policies. We then showed how one could safely install certified apps through the App Installer and how the Privacy Manager allows users both to observe how these apps treat their privacy and to define application-specific fine-grained privacy policies. Finally, we presented our certification mechanism. The Code Transformer that sanitizes apps in the developer’s computer and the cloud-based certification service that verifies the integrity of applications before they are listed for users in the App Installer. Next, we will present how Crystalline was implemented to fulfill the requirements imposed by our architecture.
In this chapter, we present the main features of our Crystalline implementation. First, in Section 4.1, we explain how a developer can use the implemented API to program Android apps on Crystalline. Next, Section 4.2 shows how Crystalline’s Runtime Middleware works overall. In Section 4.3, we display how sandboxes are implemented for untrusted elements, preventing untrusted code from leaking data. Section 4.4 covers the implementation of the Code Transformer and how its AOP-based mechanisms help Crystalline secure applications’ integrity. Then, in Section 4.5, we explain the cloud-based App Certification Service implementation and how we created a secure certification authority for apps. Finally, in Section 4.6, we present Crystalline’s App Manager, which users can interact with in order to obtain certified applications and define privacy policies for the apps already installed.

4.1 Development Framework

Section 3.5 covered Crystalline’s Development Framework and how it allows developers to create privacy aware Android apps. Now we present its implementation and how to develop applications with it.

Application developers neither have to nor shall, write code directly for the components classes, as these are injected with wrapper code to deviate their methods invocations to the middleware’s classes. They should, however, write a graph descriptor file for each component, which must state the elements that comprise the graph and how they are interconnected. The graph descriptor files should be implemented as Java subclasses of Crystalline’s API ActivityGraphDescriptor or ServiceGraphDescriptor classes, according to the type of component each one represents. We now present an example of how to develop a simple Crystalline app.

Developing a simple app: To understand the differences between developing Crystalline applications and standard Android apps, let us assume that the developer wants to make an application with a button and a text on the screen that displays the number of times the button was clicked. Both with traditional Android programming and Crystalline, the app must have an Activity associated and an XML file stating the window layout to be presented to the user. Then there must be code to implement the app’s behavior.
public class Activity1 extends Activity {
    int counter = 0;
    @Override
    protected void onCreate(...) {
        final TextView textView = findViewById(R.id.textView);
        final Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                textView.setText("Click count: " + (++counter));
            }
        });
    }
}

Listing 4.1: Example code for making a button click change a TextView's text with normal Android programming.

//In HandleClickClass.java
@CustomElement(name="HandleClick")
public class HandleClickClass extends Element{
    int counter = 0;
    @EventReceiver
    public void onEvent(...) {
        sendEvent(new Event<String>("Click count: " + (++counter)));
    }
}

//In Activity1GraphDescriptor.java
public class Activity1GraphDescriptor extends ActivityGraphDescriptor {
    public Activity1GraphDescriptor() {
        setActivityClass(Activity1.class);
        requireModule(ViewModule.class);
    }
    @Override
    public GraphStructure defineGraphStructure() {
        return new GraphStructBuilder()
            .defineElement("HandleClick",HandleClickClass.class)
            .defineElement("buttonClick",ViewModule.Click(R.id.button))
            .defineElement("textUpdate",ViewModule.TextUpdater(R.id.textView))
            .connect("buttonClick",0,"HandleClick",0)
            .connect("HandleClick",0,"textUpdate",0)
            .build();
    }
}

Listing 4.2: Example code for making a button click change a TextView's text with Crystalline.

Listings 4.1 and 4.2 display the code required to define the application's behavior in traditional Android programming, and with Crystalline, respectively.

As depicted in Listing 4.1, to program this app, usually the developer would write code in the respective Activity's `onCreate` method. This code consists mainly of passing a callback to the Android API, to listen to click events on a View with a specific ID. When the View is clicked, Android invokes the callback function, which directly changes the text of the TextView by calling the `setText` method.

Listing 4.2 shows how to program the same app with Crystalline. A developer should first implement an (untrusted) element – through the HandleClickClass – that outputs the updated text when receiving a graph event. Then, on the Activity's graph descriptor – implemented by the class Activ-
Figure 4.1: Visual representation of the example graph defined in Listing 4.2.

ity1GraphDescriptor –, one must import the View module – line requireModule(ViewModule.class) –, since the graph uses UI functionalities. The implemented element should be referenced in this file, and linked to its implementation class – defineElement("HandleClick",HandleClickClass.class. The developer should also configure the required View elements to reflect UI objects on the screen. In this case, the View Click element must be attached to the button's ID – ViewModule.Click(R.id.button) – and the TextView Update Text to the TextView's – ViewModule.TextUpdater(R.id.textview). Finally, the developer should connect the graph in a way that reflects the intended behavior. Figure 4.1 displays a visual representation of this graph. The View Click element sends a graph event to the Handle Click when the user clicks on its corresponding button. Then the Handle Click dispatches an event with the new text. The TextView Update Text receives this event and updates the TextView accordingly. Next, we explain in more detail the aspects related to the graph descriptor and trusted and untrusted elements.

**Graph descriptor:** In practice, developers are responsible for implementing two methods in their component's graph descriptor class. The constructor should state the class of the component which the graph controls and the necessary modules to build the graph. The graph structure method must return a GraphStructure object, which contains, in strings, the elements used by the graph, and a map representing their connections. To ease the developer’s task of declaring the graph’s structure, we provide them with the GraphBuilder class, which builds a GraphStructure object by subsequent calls to its methods (see method defineGraphStructure in Listing 4.2).

On the one hand, the GraphBuilder defineElement method can be used to define custom (untrusted) elements’ names and link them to their implementation classes, or define aliases for elements provided by the imported modules (trusted) – e.g, in Listing 4.2, the line defineElement("buttonClick", ViewModule.Click(R.id.button)) defines the alias "buttonClick" for a View Click element associated with R.id.button.ID View ID. Yet, aliases are not necessary as they only exist as a convenience tool for developers to keep track of multiple graph elements names. On the other hand, the connect method defines a connection (edge) between two elements. We also let developers specify the ports for each connection, as an element may have multiple ports for different types of input or output events. For example, in Listing 4.2, the line connect("buttonClick",0,"HandleClick",0) connects port 0 of the View Click element, associated with the button, to port 0 of the Handle Click, as represented in Figure 4.1.

**Using Trusted Elements:** All module classes have static methods that return unique strings identifying declared trusted elements. Some modules have configurable elements, meaning they need to
be configured by the developer to perform their functionalities. This is the case for the View module, which allows developers to use multiple elements for UI purposes, but requires them to link each one to the specific View on the screen. For elements of this module, their configuration arguments must be IDs of Views declared in the app's XML window layouts. For example, in Listing 4.2, the developer imported a trusted View Click element, and configured it with the ID of the button by writing `ViewModule.Click(R.id.button)`. This method accepts the ID of a View and returns a string identifier for the element, unique to that View's click events.

Implementing Untrusted Elements: To implement untrusted elements classes, developers must use the "CustomElement" annotation, with a parameter stating the element name, as defined in the graph descriptor. Furthermore, they must annotate the methods to be called when new events arrive with "EventReceiver." This annotation also accepts the input port which the method listens to. If omitted, then it will be notified of all events, arriving at any port. An element may have multiple handler methods for the same port, in which case all these methods are notified when an event arrives at that port.

Implementation Remarks: We should state that the current implementation of Crystalline does not cover all the functionalities natively provided by the Android API. However, the modules and elements developed are enough to evaluate Crystalline in some of the most common Android features.

4.2 Runtime Middleware

As seen in Section 3.6, Crystalline's Runtime Middleware is divided into four parts: First has an event bus in charge of delivering events between elements and between trusted elements and drivers. Second, it has multiple drivers that translate global events into actual system calls, and notifications received by the system into global events. Third, it has trusted elements to which application-specific ones (untrusted) can rely on to perform multiple functionalities. Finally, it has the Privacy Enforcer, which re-configures the graphs according to user-defined privacy policies. In this section, we present the most relevant characteristics of the Runtime Middleware implementation.

The Runtime Backbone: Figure 4.2 presents a UML representation of the Runtime Middleware implementation. Note that this image is not complete, as it omits multiple driver classes, and does not include the developed trusted elements, nor the multiple sub-types of graph events and global events. It does, however, present the core classes of our middleware and how they relate to each other.

The App Controller is in charge of booting the middleware when the app launches. The Activity Aspect and the Service Aspect notify this class when the first component of the app is launched. Then, the App Controller loads the middleware by two parallel tasks: As for the first task, it asks the App Launcher Client to communicate with the App Launcher (within Crystalline's App Manager application) to obtain the latest user-defined privacy policies. Meanwhile, the second task reads the app's Android manifest file, so it can identify what components compose the application, and their respective graph descriptor.
classes. Then it passes this information to the Graph Manager, which instantiates and connects all elements for each Graph instance. The Graph Manager returns the App Controller the list of drivers that each graph requires, so that the latter may initialize them in advance (completing the second task), and enable or disable them during the app’s lifetime according to the components active at each moment. Once the App Launcher Client has received the user-defined privacy policies (completing the first task), it forwards them to the Graph Manager, which implements the Privacy Enforcer’s functionalities by reconfiguring the Graphs in accord with the restrictions imposed by the policies. Graphs can be reconfigured with the following options:

- Disable trusted elements that are either sources or sinks of sensitive data – source-blocking or sink-blocking policies.

- Make trusted elements ask the user for permission to run their operations – ask first policies.

- Add trusted elements to the graph, to transform sensitive data right before it is handed to specific untrusted elements – data transformation policies.

Once the graphs are fully configured, then the first Graph is booted, and the app is ready. When an event happens in the underlying Android API, either callback interfaces or the Aspects notify the appropriate drivers, which then translate the system events into global graph events, sent to the Event Bus, and handled by trusted elements, which may dispatch layout events to notify other connected trusted or untrusted elements in the graph. On the other hand, when a trusted element receives a graph
layout event that tells it to run an operation, a global event is dispatched to a driver, also through the Event Bus, which then performs the appropriate Android API method calls.

**Handling Components’ Life-cycle:** When a component is created, all the drivers required by its graph are enabled. When Android invokes a component life-cycle method, the corresponding Advice code intercepts the call and notifies interested drivers. All drivers have direct access to the resources they control, either through direct calls to Android’s API or through AspectJ Advices. Advices and their role on supporting Crystalline’s Runtime Middleware are explained in more detail in Section 4.4.

**Handling Android Permissions:** When a driver detects that it does not have the necessary Android permissions to operate, it notifies the Permission driver, if there is an active Activity, then it uses it to request permissions to the user. However, if the app is only running as a Service, the driver will be notified that the permission request failed, since Android only allows Activities to request dangerous permissions at runtime. When the user grants or denies an Activity with a permission, the driver will be notified of the result, and may either perform the intended operation or inform the appropriate trusted elements of the permission denial event.

**Enforcing Ask-first permissions at runtime:** The Permission Driver does not handle only Android permissions requests. It is also in charge of asking the user for consent every time a trusted element configured with the "ASK_FIRST" constraint attempts to execute its functionality. For example, if the user decided that an application must ask him before sending data to a specific cloud server, then the HTTP Post/Get element in charge of that task is configured with an "ASK_FIRST" parameter. When receiving a graph event that requests it to execute the cloud call, the element dispatches a "REQUEST_PERMISSION" global event, which the Permission Driver listens to. As shown in Figure 4.3, the
Permission Driver then uses the current Activity to create an alert dialog asking the user for permission. If the user gives his consent, then a "PERMISSION_GRANTED" event is returned to the element. If not, then a "PERMISSION_DENIED" event is dispatched. When the HTTP Post/Get element receives one of these events, it either executes the functionality or disables itself, respectively. If the app has no Activity currently active, then the permission request will be automatically denied, as we can not ask the user for consent without the UI context exclusive to Activities.

4.3 Sandboxing Untrusted Elements

HomePad uses the Java Security Manager to sandbox untrusted elements instances. Originally, the Security Manager was designed to restrict multiple Java applets running within the same JVM, thus sharing the same address space, from interfering with each other. However, Android’s security model only considers isolation bounds for applications at the process and kernel levels and does not provide support for Java Security Managers. Therefore we can not use this mechanism to sandbox untrusted java code running within the same ART application environment.

To sandbox untrusted elements in Crystalline applications, we altered the internal mechanics of Java’s class loading model to create separated class namespaces for each untrusted element instance. Java loads classes with the help of class loaders. A class loader is itself an object that attempts to resolve class definitions from their fully qualified name. Every Android app has at least two class loader instances on launching: the internal bootstrap class loader, which loads native Java classes, and the application class loader. The latter is in charge of loading all the application-specific classes as well as the ones belonging to third party dependencies, like Android API’s libraries. While the application is running, the ART engine resolves classes when finding new references to them. When class X references Y – either in a method, a static block, or its constructor –, the engine looks up Y's name in a pool of classes loaded by X’s class loader. If it does not find it, then it asks the X’s class loader to load the referenced class.

The bootstrap and application class loaders have a parent-child relationship. Traditional Java class loaders employ a parent-delegation loading model for classes. In our example, the application class loader was the one to load class X. When requested to load Y, it will first ask its parent – the bootstrap class loader – to do it. The child class loader only attempts to find a class in its search path ¹, only if the parent fails to load that class. This is the case for our example, as Y is an application-specific class, and thus does not belong to the bootstrap class loader’s search path. However, X’s class loader is able to find this class within the application binary files, and consequently transform its bytecode into a runtime class definition. As the class Y is resolved, it becomes visible to X and associated with the same class loader. Note that, if it were a native java class – like java.lang.Integer – referring Y, the loading would fail with a ClassNotFoundException error. This happens because the class Integer is loaded by the bootstrap class loader, which is the parent of the application one. In this model, only children may ask their parents to help them load classes, not the other way around. Because Y is not a native java class,

¹ A class loader’s search path is the list of binary files from which it loads classes
By exploring this mechanic, we developed a Sandbox Classloader in charge of resolving classes within an isolated namespace, which we refer to as a bubble. Listing 4.3 shows the most relevant parts of the Sandbox Classloader’s implementation. We omitted the constructor due to its simplicity. It takes two arguments, the binary file names that compose the search path to consider in the findClass method and the parent class loader. The findClass method is implemented in the super-super-class, Android’s BaseDexClassLoader. This method iterates over all opened binary files within the search path in an attempt to find the class’s bytecode. The implemented loadClass method is the one called by the ART engine to load a class the first time it is referenced in the bubble. It identifies the class as white-listed, black-listed or untrusted by matching its name against multiple regex patterns, and acts accordingly.

The need for this categorization of classes arises from the solution we found to isolate untrusted elements. These must be sandboxed at two levels: first, they should not be able to access directly some functionalities of Android’s API – we defined a set of black-listed classes that untrusted code should not reference. Second, elements should not be able to share memory between them – we can translate this into the set of all classes that can store data in a static context, which should not be referenced by two different elements (unsharable classes). When asked to load a class, Crystalline’s Sandbox Classloader deviates from the traditional delegate-parent-first model, and instead considers the following:
1. If the class name is black-listed, then it throws ClassNotFoundException.

2. If the class name is white-listed, then it delegates the loading to its parent class loader.

3. If none of the above are true, then the class is deemed unsharable, and the Sandbox Classloader attempts to (re)load it from its binary search path, creating an exclusive definition only visible to other classes inside the bubble.

**Black-listed** classes should not be accessible by untrusted classes, as they may come as channels for harmful behavior. For example, classes belonging to the java.net package can be used to leak data to the Internet, and the ones within java.lang.reflect grant dangerous reflection capabilities. A **white-listed** class is considered as harmless by Crystalline, and the Sandbox Classloader delegates its loading to its parent, the application class loader. Therefore classes of this type are resolved in the broader application’s namespace, and their definition will be shared not only between the bubble and the outside but also with the other multiple existing bubbles. Some examples of these classes include most java native utility classes like java.lang.Object, and Java .util.ArrayList, as well as and the ones provided by Crystalline’s development API. Finally, **unsharable classes** are, in practice, the ones created by the application developers or even brought in by third-party-libraries. Crystalline does not trust these classes, as they may attempt to perform dangerous behavior. However, all the means they have to leak or obtain sensitive data illegally involve at some point referencing black-listed classes, which we already block. Nonetheless, two bubbles may attempt to access the same class as a means to trade data between them. Crystalline prevents this by making each Sandbox Classloader instance re-define an unsharable class, which then its added and visible only to the rest of the respective bubble. Because different bubbles resolve the same unsharable class name into different memory definitions, they can not use it to share information between them.

Take as an example Figure 4.4. In it, one can identify two bubbles, each associated with a Sandbox Class Loader Crystalline instantiated to load an untrusted element’s implementation class. In this scenario, assume that the developer has some malicious intentions. He implemented the SensitiveData-Handler, an untrusted element that receives some sensitive data from other graph’s elements, to leak the user’s data by making an HTTP request to a web server with the help of the java.net.URLConnection class. Additionally, he also made this element write the sensitive data it receives on a static field of another class he developed – SensitiveDataStorage. Another malicious element named CloudCallPreparer – which does not receive any sensitive data, but is in charge of preparing a cloud call request to be processed by the trusted HTTP Get element – tries to access the data stored in the SensitiveDataStorage class, and then include it in the request parameters, thus leaking it. To stop the first attack, when the runtime engine asks the Sandbox ClassLoader to load the java.net.URLConnection, the class loader verifies that the name is black-listed and aborts the operation. In the second case, each untrusted element is loaded by a different instance of the Sandbox ClassLoader, and the SensitiveDataStorage is considered an unsharable class. Therefore the CloudCallPreparer always reads a null value, as it is not accessing the same class definition, and consequently, the field which SensitiveDataHandler wrote to.
4.4 The Code Transformer

While sandboxing untrusted elements guarantees that their code does not perform dangerous behavior, a malicious developer may attempt to write code outside the graph's scope in order to circumvent the sandbox’s isolation mechanisms. This would introduce a severe vulnerability that could be leveraged for data leakage or illegal processing in general. In particular, the following cases should not be allowed to happen:

- Code within the app’s life-cycle methods. These are invoked by Android, and have full access to the application context and resources.
- Calls to the Android Framework API methods which are not performed by Crystalline’s middleware.
- Execution of native (C/C++) code, which may inject the app’s runtime engine with malicious code.

The Code Transformer sanitizes the compiled application by neutralizing the threats mentioned above. We consider the necessity of blocking these code patterns from running as a cross-cutting concern. As depicted in Figure 4.5, with the help of Aspect-Oriented Programming (AOP), it is possible to define a set of execution points patterns within applications that should only be executed by the middleware. By weaving the app in search of execution points that meet these patterns, and injecting them with safety-guard code, we can assure that no developer’s code has access to Android’s API.

Our implementation uses AspectJ [54], as it is the current most recognized and transversely supported AOP framework, and it is also widely supported on the Android platform. AspectJ allows devel-
opers to write code within Advices to be injected in certain Join-points by declaring Pointcuts. Join-points are specific simple runtime events, like method calls, method executions, and object instantiations. A Pointcut is a set of conditions that may match multiple Join-points. Finally, an Advice is a method that contains code to be executed before, after, or even around the multiple Join-points defined by the Pointcut conditions. By programming code to be executed in these Advices, developers can easily employ cross-cutting concerns like logging, exception management, or even security. We defined multiple AspectJ’s Pointcuts to forward the notification of Android events to middleware classes, and to stop malicious from running outside untrusted elements.

Aspects to intercept life-cycle methods’ invocation: In order to intercept the invocation of dangerous code inside the application’s components life-cycle methods, the ActivityAspect and ServiceAspect aspects wrap their respective component’s life-cycle methods and stop the originals from being called. In turn, they dispatch notifications to Crystalline’s middleware’s App Controller and to their respective drivers.

Listing 4.4 displays how the ActivityAspect was implemented. The Around annotations, which tell AspectJ weaver to make the following Advice wrap the original method. This allows us to determine if the latter will even be executed at all by invoking jp.proceed(), which we do not. This way, we stop malicious code from running in the Activities life-cycle methods. The Pointcut defined matches the onCreate method, invoked with a Bundle argument, of any class (*) that extends (+) an Activity, provided in Android API. The execution parameter tells AspectJ that the code is to be injected within the called method. If we used the call parameter instead, then it would be injected precisely on the line which the method was called from, which in this case, belongs to an internal Android API class that should not be tampered with. Similarly as for the onCreate method, we defined Advices to wrap all Activities and Services life-cycle methods.
Aspects to intercept dangerous method calls: On the one hand, in Section 4.3, we explain how Crystalline forbids untrusted elements from accessing harmful functionalities. On the other hand, the ActivityAspect and ServiceAspect classes ensure that no developers’ code runs in the application entrypoints life-cycle methods. However, a malicious developer may also write static code blocks to obtain the application’s context or even run native code to tamper with the App’s internal runtime engine. AspectJ does provide the Pointcut parameter staticinitialization to intercept static code. However, we do not want to block all static initialization of classes, as this is an important Java feature that application developers may need while creating classes. Instead, as depicted in Listing 4.5, we declared multiple Advices that together completely block the developer’s code capability from calling dangerous methods, including the ones which can be used to launch native (C/C++) code.

Weaving the Code: To inject each Advice in the corresponding Join-points, AspectJ supports weaving the application in either compile-time – before JAVA files are compiled to CLASS files –, binary weaving – before CLASS files are packaged into the app –, or even load-time-weaving, by changing the runtime application class loader. However, this last option is not supported by ART for security reasons. Compile-time weaving does not suffice in our case, as it does not weave already compiled third-party libraries, which may also be malicious. For these reasons, we chose binary weaving.

Crystalline’s Code Transformer was implemented as a Gradle plugin, which developers should use...
to weave their apps automatically. We adopted GradleAspectJ-Android [55], a useful tool created for Android developers to weave applications with AspectJ code. The Code Transformer passes the Aspects developed to the GradleAspectJ-Android plugin, which proceeds to weave all app's CLASS files, even the ones imported by third-party libraries. However, it skips Android API libraries in order to avoid tampering with the underlying Android functionalities. Developers can add the Code Transformer plugin to their applications' Gradle Build script by importing Crystalline's repository and appending the line "the apply plugin: crystalline." When doing so, they are also automatically importing Crystalline's development and middleware libraries. During the building process, the Code Transformer runs its tasks after the JAVA files are compiled into CLASS files, but before the CLASS files are merges into DEX ones.

4.5 App Certification Service

Weaving the app's bytecode is not sufficient to guarantee it was not tampered with after the fact. Because the code weaving takes place in the developer's computer, we have no guarantees that the app he or she sent to Google Play is properly weaved. Therefore, we need a trusted machine to confirm if the AspectJ's Advices were injected correctly. This section explains how we implemented a web-server to be the Crystalline's App Certification Service and how it verifies applications' integrity. The App Certification Service was designed as a cloud-based service in charge of certifying apps published by developers, and making them available to Crystalline users. Figure 4.6 depicts the certification process. It relies on four major components: The App Registration Server, which handles the registration of new apps, or updates to the ones already recognized. The Validator, which is in charge of determining if published apps are trustworthy, certifying the ones which are. A database that persistently stores information about apps that passes the verification task. And finally, the Certified Apps Store, which is a server that provides certified certified Crystalline applications information for the App Installer.

App Registration Client and Server: As seen in Section 3.8, developers must register their apps with Crystalline App Certification Service, so they can be validated and certified by Crystalline, and consequently appear on the store for users to install. To do this developers can use the App Registration Client to send the required information to the Validator in order to verify their apps. We implemented the client as a simple browser page, as our focus was directed to the service itself. The Registration Server was implemented as HTTP server, running on top of nodeJS with the Express middleware, as these two together provide an easy environment to develop web servers.

When registering an application, developers must send some information that allows the Validator to verify the app's integrity. In practice, we ask the developer to send the app's unique id on Google Play, the app's source code, the buildscript, and the imported third-party libraries. After the validation task finishes, the developer is notified if the app was successfully certified or not.

Validation Task: Once the app is uploaded, the App Registration Server asks the Validator to verify if the application does not contain adulterated versions of Crystalline's Runtime Middleware classes and if
the Code Transformer correctly weaved it. The first verification is vital because developers may adulterate the Crystalline's library in order to disable the middleware authority. The second one guarantees that the Advices injected by the Code Transformer are active, and grant Crystalline's middleware exclusivity to Android's API classes. The Validator was written in nodeJs, to match the rest of the Certification Service’s implementation. The first thing it does is to build the app from the source code, the build script, and the libraries passed in the registration request. While doing so, it makes sure to import clean versions of Crystalline's library and Android API ones, as well as to replicate the weaving task as should have been done by the Code Transformer. This re-created app is inherently benign, as the weaving was done in Crystalline trusted environment, and the build process used clean libraries from a trusted repository. In fact, if the developer is not ill-intended, then the app he submitted to Google Play should be identical to this one, except for the fact that it is signed by the developer.

Having re-built the application as it should be, the Validator downloads the app’s APK file that the developer published to Google Play. After the download is complete, the Validator extracts (unzips) its binary DEX files, and compares these files against the ones from the re-built app. If they match, then the app is trustworthy and thus certified. In practice, certifying an app means adding its metadata (title, package name, its version, a short description, and the icon image) to the database of trusted apps.

The binary comparison was implemented similarly to JAR (and APK) files authentication mechanism. The Validator compares the digest entries (hashes) within both APKs’ MANIFEST.MF files. If all entries have the same value, then both apps have the same binary data. This means the APK uploaded to Google Play by the developer contains a clean Crystalline Runtime Middleware library and was adequately built and weaved by the Code Transformer.
Certified Apps Store: The Certified Apps Store is the server from which the App Installer module, residing within users’ Android devices, obtains information about certified apps. For the same reasons as the App Registration Server, we also implemented it with nodeJS and Express. It has three endpoints which the App Installer may query. One to get the front-page apps, which is comprised of the ten most popular Crystalline apps; Another that returns a list of apps that match the search string sent in the request; And another that receives an app's package name and version, and return a boolean declaring if that app's version is certified. In all cases, the server will obtain the results by querying the database where the validator stored apps metadata. However, the last request makes this server query Google Play to check if the app version did not change since its previous certification.

Implementation Remarks: The fact that the verification procedure requires developers disclosing their applications’ source code to the cloud-based certification service could be seen as a disadvantage of Crystalline. However, it is possible to implement the Validator in a way that does not need the source code, but only the compiled application sent to Google Play. A possible implementation following this approach could rely on using Apktool [56], a tool which can decompile the APK bytecode into an assembly language (Smali). Then, an inspection mechanism must parse the decompiled classes to check if the Code Transformer correctly injected the supposed code in the appropriate lines of code. However, this procedure would require more logic than the one we implemented.

Another thing we should mention is that in order to verify that this implemented service worked, the Validator was configured to not download apps from Google Play, but rather from a mock-up repository we made, containing the use-case apps. We did so because, through the development of our work, we did not want to publish these apps to Google Play, as they are test case applications that should be open to the public. For the same reason, we added an extra endpoint to the Certified Apps Store, which provides the use-case APKs, so the App Installer may download and install them directly.

4.6 The App Manager

The App Installer, App Launcher, and the Privacy Manager were all implemented as part of the App Manager. Figure 4.7 presents three screens of the App Manager user interface. The first one belongs to the App Installer. The second one displays the Privacy Manager's report for a use-case app. The third one, presents the Privacy Manager's policies definition user interface. When launched, the App Manager's default behavior is to open the App Installer. It also has a menu where users can switch between two Activities – the App Installer Activity, and the Privacy Manager Activity. The App Launcher was not meant to be accessible to users, but to provide the user-defined privacy policies to Crystalline’s Runtime Middleware within each app. As it requires no user-interaction, we implemented it as a Service.

The App Installer: As presented in Figure 4.7a, when opening the App Installer, users are presented with the top most popular certified apps and can search for specific applications in the top bar. When searching for an app, the Certified Apps Store server will be queried to return all apps which name
matches the searched string. When the user selects one app, a page is presented with the description of the app, and the choice to download it. If the user decides to install the app, the App Installer asks the Crystalline App Store to verify if the app version did not change on Google Play. This is an important step, as malicious developers may update a certified app on Google Play, without notifying Crystalline, thus skipping the certification process. If the app in Google Play has the same version number as when last certified, then the user is forwarded to the download page on the Google Play App.

The Privacy Manager: Once an app is installed, the user may open the Crystalline App Manager and select the Privacy Manager on the menu. This Activity scans the device in search of installed apps by checking for each one if its information (UID, package name, version) is in the App Manager’s local database. If it finds an app that declares using Crystalline (on its Android manifest XML file), but it is not in the local database, then it queries the Certified Apps Store server, asking it if the app’s package name and version are certified. If so, then it adds it to the local database. As shown in Figure 4.7b, once the scan finishes, it presents to the user all current installed and certified Crystalline apps.

When the user clicks on an app’s entry, the Homepad-based data flow inspection algorithm runs, generating Prolog facts describing each graph’s connections, and querying these facts for how sensitive data flows through the graph’s elements. Because Crystalline supports inter-component communication, when inspecting a component’s graph, if the algorithm finds an ICC element that obtains sensitive data from a second graph, then it parses some of the second graph’s data flows in order to track the data types that are disclosed to the first graph. If the data type has a circular flow between graphs, then that flow is ignored, as it does not lead to the original source of the data, which must be trusted element

---

58

---

2 The internal graph inspection/policy checker algorithm was adopted from HomePad, and thus it is not the scope of this work to detail it. We only state the changes implemented by Crystalline. The original algorithm can be found in [30].
that connects to Android resources, and not other graphs. Crystalline adopts tuProlog [57] as its Prolog engine, as it provides a Java API compatible with Android. When the algorithm finishes, the Privacy Manager transforms the output results into readable information. Users are presented with the types of sensitive data an app accesses and to where it discloses them. We do

As presented in Figure 4.7c, if the user is not comfortable with the app using a specific type of sensitive data, or does not want it to be shared to a particular party, he or she can impose rules to disable these behaviors. Additionally, one may force the app to ask permission every time it attempts to obtain or send out a data type. For some data types – e.g., gallery photos, GPS location – the user can also decide to transform the data before it leaves the device. These constitute the implemented types of privacy policies, which are enforced by Crystalline’s Runtime Middleware. When the user defines a privacy policy, the Privacy Manager attributes rules to the trusted element that it affects. For instance, blocking sensitive data from going to a web-server disables the HTTP elements through which that type of data flows. Crystalline stores these rules in the App Manager’s local database, which is implemented with SQLite, since Android natively supports for this technology.

**The App Launcher:** As seen in Section 4.2, when the user launches an app, Crystalline’s Runtime Middleware asks the App Launcher for the user-defined privacy policies. Once the App Launcher receives this request, it verifies the UID of the launching process and checks the local database for information about the app, namely its version and package name. If the database returns a match, it means the app is certified, and it displays to the user a notification assuring him of that. If the app is not in the database, then either the app is maliciously pretending to be supported by Crystalline, or the user did not open the Privacy Manager after installing it. In both cases, the launching is aborted, and the user forwarded to the Privacy Manager, so it can construct the privacy report similar of the app, similar to the one presented in Figure 4.7c. If the app is certified, the App Launcher fetches its user-defined policies from the database and returns to the app’s Crystalline middleware a JSON String stating these. In Listing 4.6, we can see the JSON string which the App launcher passes to the Runtime Middleware, to notify it to disable the HTTP Post element in charge of communicating with xyz.com/1g3AhR6 , and make the other Post element that sends data to healthbuddy.org/stressmeasure re-as the user every time for consent before executing the call.


**Implementation Remarks:** As stated in Section 4.5, the developed use-case apps are not available in Google’s Play. Therefore, we added a debug button to the App Installer Activity that installs applications directly from Crystalline’s Certified Apps Store. We also had to change the testing device default settings to allow installations from sources other than Google Play. However, this was just to evaluate our framework, as Crystalline was designed to rely on Google Play to provide the apps themselves.

### 4.7 Summary

In this chapter, we presented the implementation of Crystalline’s development framework and exemplified how developers can build a simple Android app with it. We then covered Crystalline’s Runtime Middleware and how it acts as a mediator between the application’s graphs and the Android API. Then we explained how Crystalline sandboxes untrusted elements, preventing them from leaking data. We also stated how the Code Transformer sanitizes the apps bytecode, assuring that our middleware has full authority over the graph, and no developer’s code runs outside untrusted elements. We then proceeded to show our implementation of the App Certification Service and the methods we use to validate and certify applications. Finally, we described the App Manager and the three modules that compose it. The App Installer that lets users install certified Crystalline apps, the Privacy Manager, which can trust to take care of their privacy concerns, and the App Launcher, that applies the user-defined privacy policies to each application upon launch. Next, we present an evaluation of our system.
Chapter 5

Evaluation

This chapter presents the evaluation of our system. We are primarily interested in addressing the following questions: How effective is Crystalline at giving applications transparency over their data flows (Section 5.1)? What is the impact for application developers in terms of programming effort when writing Crystalline-supported applications (Section 5.2)? What is the performance overhead endured by both applications and the overall system when running the Crystalline platform (Section 5.3)? What is the overall security assessment of our system regarding in particular its potential security weaknesses (Section 5.4)? Next, we address each of these questions in its turn.

5.1 Effectiveness

We have seen that application development with the native Android framework does not offer guarantees on how sensitive data is processed, but provides instead a coarse set of information about what resources an application might access during its life-time. In this section, we evaluate if apps developed with Crystalline are transparent regarding their sensitive data flows, and if our framework allows users to understand and have fine-grained control over how installed apps treat sensitive data.

We consider three different scenarios of applications interacting with sensitive data. The first one covers apps that only process data locally, and thus have little privacy implications (e.g., the data is never supposed to be exfiltrated from the device). With the second one, we analyze the benefits of Crystalline in cases where applications do not disclose data to the cloud but require network connectivity for other reasons. In the third and final scenario, we evaluate Crystalline in instances where applications obtain sensitive data and send it to the cloud. To evaluate our framework’s effectiveness in these scenarios, we developed three apps that collect the user’s heart-rate, and process it in different ways, each representing a scenario with its concerns regarding privacy. Our implementations of these use-case apps are presented in Appendix A.

Case 1: Read local, stay local. With this test, we want to evaluate how Crystalline compares to the native Android permission system when the application does not need Internet connectivity and only...
Figure 5.1: Heart rate flows in a read local, stay local app – HeartRateApp (Local) app.

<table>
<thead>
<tr>
<th>System Benefit</th>
<th>Android permissions</th>
<th>Crystalline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feedback</strong></td>
<td>Reads the device location; Can pair with Bluetooth devices; Accesses Bluetooth settings;</td>
<td>Gets user’s heart rate from a Mi Band 2 device;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not disclose it to the outside.</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Deny access to location, which blocks Bluetooth functionalities aswell.</td>
<td>Deny access to user’s heart rate; Require user’s permission before obtaining an heart rate value.</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of privacy feedback and control benefits between Android permissions and Crystalline’s Privacy Manager, regarding the HeartRateApp (Local) processes sensitive data locally. Our evaluation method consisted of observing what privacy information and control measures both these systems offered about the HeartRateApp (Local), which processes the user’s heart-rate values, obtained from a Mi Band 2 wristband, through a Bluetooth connection.

The HeartRateApp (Local) has two graphs, each corresponding to an Activity. The first one uses the device’s Bluetooth capabilities to scan the nearby area for a Mi Band 2 device. When found, it launches the second Activity. The first Activity’s graph does not process sensitive data, so we do not detail it here. The second graph sensitive flows are presented in Figure 5.1. The MiBand2 Heart Rate element sends heart rates to the Text Updater, which shows their values on the screen – $F_2$. Additionally, the Compare With Reference element is also notified of new heart rate values and compares them to a reference value. If the user’s heart is beating faster than the reference, then an event is sent to the AlertDialog Show, which creates an alert on the screen – $F_1$.

A comparison between Android’s permission system and Crystalline’s Privacy Manager on handling users’ privacy concerns for this app is presented in Table 5.1. When running the app for the first time, Android prompts the user to decide whether to allow the app to access the device location. This is a good example of how this system is somewhat responsible for users’ misconceptions about how apps treat their data. This happens due to a security measure Google implemented on Android 6.0, which requires apps to have the ACCESS_COARSE_LOCATION permission when accessing the device’s Bluetooth adapter [58]. Additionally, when opening the HeartRateApp permissions window on the device settings, users can prevent the app from accessing the location, and consequently, Bluetooth functionalities.
Now with Crystalline, the Privacy Manager identifies, when inspecting the app’s graphs, two sensitive flows of data that do not end up on an element capable of disclosing data to the outside. This information is enough to inform the user that the application only processes heart-rate values locally. Users are also offered the opportunity to block the app from collecting this type of data or make it ask for permission before collecting the data.

To summarize, we confirmed that Crystalline informs the user that the app would collect heart-rate values. The user is also offered with fine-grained control over this data type, specifically to determine if the app could obtain it directly or would require us granting permission first. It is also possible to disable access to this data type permanently. In contrast, the Android permission system told us that the app required access to the device’s location, which is not practically correct. The user is also not offered with any means to understand what data the app obtains through Bluetooth connections, and can only revoke the application’s access to the device’s location.

Case 2: Read from cloud, process locally. Most Android apps need Internet communication. Some indeed use it to disclose their users’ private data to the cloud. However, many apps require access to the network for other reasons, e.g., to authenticate the user or obtain information from the cloud. Normally, Android users could not be sure an app does not leak data obtained locally to the cloud, as the permission system only informs if the application has the power to access the Internet. With this test, we evaluate the benefits that Crystalline brings to the apps which do not send sensitive data to the cloud despite needing network connectivity none the less. We observed what information the Privacy Manager presents to users in this scenario, and how it correlates with one’s privacy concerns, compared to Android’s permission system.

The use-case app developed to represent this scenario is also based on the HeartRateApp. However, this version does not have the reference value for the heart rate hard-coded within an untrusted element, but instead gets it from the cloud. As depicted in Figure 5.2, the application uses the GET element provided by the HTTP Module. The latter was duly configured to interact with the web-server’s endpoint in question by the developer. Once the server responds with the reference value for a healthy heart rate, the application is notified of its value in the Compare With Reference element, which saves it in its private state. While receiving heart rate values from the MiBand2 Heart Rate element, the Compare With Reference element checks if they are within the range according to the reference. If not, a warning is generated by the AlertDialog Show element – $F_1$. The other flow of data – $F_2$ – is the same as in the
Table 5.2: Comparison of privacy feedback and control benefits between Android permissions and Crystalline’s Privacy Manager, regarding the HeartRateApp (Cloud-Local)

<table>
<thead>
<tr>
<th>System Benefit</th>
<th>Android permissions</th>
<th>Crystalline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback</td>
<td>Reads the device location;</td>
<td>Gets user’s heart rate from a Mi Band 2 device;</td>
</tr>
<tr>
<td></td>
<td>Can pair with Bluetooth devices;</td>
<td>Does not disclose it to the outside.</td>
</tr>
<tr>
<td></td>
<td>Accesses Bluetooth settings;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Has full network access.</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>Deny access to location, which blocks Bluetooth functionalities as well.</td>
<td>Deny access to user’s heart rate;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Require user’s permission before obtaining an heart rate value;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Require user’s permission every X minutes to get heart rate values.</td>
</tr>
</tbody>
</table>

HeartRateApp (Local) app.

Table 5.2 presents a comparison between how Crystalline and Android’s permission system handle this app privacy concerns. When checking the app’s permissions, the user can verify that it is able to 1) access the device’s approximate location, 2) pair with other Bluetooth devices, 3) access Bluetooth settings, and 4) have full network access. These four statements comprise the entire assessment which Android grants its users with. Naturally, with just this information, the user can not perceive the app’s actual behavior. That is, only accessing the Internet to get a reference value and not even processing the device’s location. More likely, one would conclude that it may obtain one’s GPS position and leak it to the cloud, which is not what truly happens, and has different privacy concerns involved.

When Crystalline’s Privacy Manager analyzes the application, it detects that the application uses an HTTP element, but no flow of sensitive data enters it. The flows of sensitive data are similar to the HeartRateApp (Local) ones — $F_1$ corresponds to heart-rate values that end up in an the Alert Dialog element, and $F_2$ for the ones that are sent to TextUpdater element, which displays them on the screen.

Regarding the displayed information, Crystalline’s Privacy Manager informs users that no sensitive data types are disclosed outside the app. Because no flow of sensitive data ends up involved in cloud communications, Crystalline does not even inform the user of the fact the app uses the Internet entirely. On the one hand, the statement that no sensitive data is exfiltrated outside the device exposes how the app treats the user’s sensitive data. On the other hand, Crystalline omits the details regarding the flows that interact with the cloud server, as they do not involve sensitive data, and thus presenting the user with information about them has no privacy benefits and might create unjustified concerns.

This example presents a case where using Crystalline has a significant benefit for the application developer: he can transparently prove to users that his app does not use the Internet permission to send out sensitive data, but for other – harmless – reasons.
Figure 5.3: Heart rate flows in a read local, send to cloud app – HeartRateApp (Local-Cloud) app

<table>
<thead>
<tr>
<th>System</th>
<th>Android permissions</th>
<th>Crystalline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feedback</strong></td>
<td>Reads the device location; Can pair with Bluetooth devices; Has full network access.</td>
<td>Gets user’s heart rate from a Mi Band 2 device; Sends it to healthbuddy.org/stressmeasure; Sends it to xyz.com/1g3AhR6.</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Deny access to location, which blocks Bluetooth functionalities aswell.</td>
<td>Deny access to heart rate values; Require asking for permission before obtaining an heart rate value; Deny sending them to healthbuddy.org, or have the app ask for permission first; Deny sending them to xyz.com, or have the app ask for permission first.</td>
</tr>
</tbody>
</table>

Table 5.3: Comparison of privacy feedback and control benefits between Android permissions and Crystalline’s Privacy Manager, regarding the HeartRateApp (Cloud-Local)

**Case 3: Read local, send to the cloud.** Privacy risks are the highest when an application discloses the user’s personal data to outside the device. However, as seen in the previous example, an app accessing the network does not mean it will send sensitive data to the cloud. Moreover, Android permissions neither warn the user if data is leaked nor inform to where it is being disclosed. One of Crystalline’s goals is to let users determine where applications can send the data they collect. In many cases, this means controlling to what cloud servers apps may send sensitive data to.

To evaluate if our work achieves this goal, we developed yet another version of the HeartRateApp. The HeartRateApp (Local-Cloud) intended behavior is to send the user’s heart-rates to two distinct cloud servers. The relevant data-flows are presented in Figure 5.3. Similarly to the first use case, the heart-rate values are retrieved from a Mi Band 2 device and delivered to the graph through the Mi Band 2 Heart Rate trusted element. This time, however, the application-specific Cloud Call Prepare element sends its values to healthbuddy.org (flow $F_1$) and to xyz.com ($F_2$), though elements provided by the HTTP Module.

We observed what Crystalline’s privacy manager showed us when analyzing this app, and compared with the information obtained from. The results are presented in Table 5.3. On the one hand, when verifying the permissions granted by Android – in the device settings menu – we are granted with the...
same information as in the previous case. The app may read the device’s location and has full network access. We could also only forbid the app from accessing the location, not the network. On the other hand, Crystalline informs the user with three statements: 1) the app obtains heart-rate values from a Mi Band 2 device, 2) sends them to healthbuddy.org, and 3) sends them to xyz.com. For each of these three statements, the user is presented with options to either disable the procedure or make the app ask for permission before doing it. This means the user can disable the application from sending heart rate values from a particular web server, while allowing the app to continue receiving heart rates from the device and sending them to the other server.

This use-case experiment shows that, in scenarios where apps send sensitive data to the cloud, Crystalline’s reports are more fine-grained than the information provided by the Android permission system. Moreover, while Android only allows users to control the resources apps may access, Crystalline lets users set fine-grained privacy policies that control where applications can send sensitive data to.

Discussion In this section evaluated how effective Crystalline is at making applications transparent regarding how they process the user’s private data. For all use-case scenarios, we were able to determine that our framework provides users with fine-grained information and control over their data, in contrast to Android’s permission system. Like the previously studied flow tracking mechanisms [14–19], Crystalline is able to track how applications process sensitive data once they collect it. However, thanks to the adoption of HomePad’s programming model, our framework does not let implicit (control) flows go undetected. Moreover, similarly to the studied permission system enchantment solutions [20–28], Crystalline provides users with a fine-grained control mechanism to enforce their privacy policies to applications. However, whereas most of these systems only let the user determine how apps access data, ours allows one to explicitly define where apps may disclose sensitive data once they collect it.

Furthermore, the studied proposed solutions to improve Android’s permission system rely on changes to the OS core or repacking applications, which difficult their adoptions by the general Android user-base. Crystalline does not change Android. In fact, it even encourages users to download apps from Google Play, which are directly published by developers.

However, Crystalline forces developers to adapt to its programming model, which has an impact on the effort needed to create applications. We evaluate the magnitude of this impact in Section 5.2.

5.2 Programming Effort

In this section, we evaluate the impact Crystalline has in the programming effort performed both by application developers and by Crystalline’s trusted maintenance and development team.

Effort for application developers Android apps are typically programmed under an asynchronous programming model, meaning that most calls to the Android API methods do not complete their intended operations before returning, but rather notify the caller when they finish, through callback interfaces. The main entry points for the developer’s code are with the multiple components’ classes, which
developers need to implement and write code within their life-cycle methods to define their applications' behaviors. However, programming with Crystalline requires developers to use a data-flow based programming model, which forces them to adapt to a different programming methodology. Developers need to consider the following things when creating Crystalline supported apps:

1. Developers need to write a graph descriptor for each application component, defining the graph structure, and the modules required by it.

2. The application-specific code cannot be written in the components’ classes, but rather in each untrusted element implementation class.

3. Developers do not have direct access to the native Android API, but instead can use Crystalline’s to make their untrusted elements interact with the rest of the graph.

4. The Code Transformer plugin must be applied in the application’s Gradle script.

5. Additionally to publishing the app on Google Play, developers must submit it to the App Certification Service, pending verification of Crystalline’s framework integrity.

Using the Code Transformer only requires adding a couple of lines to the app’s Gradle script. Publishing the app to Crystalline’s Certification Service involves uploading a zip file every time one wants to update the app’s production version available to users, which happens typically with long-time intervals that may range from days to months. With respect to assessing the development effort of Crystalline apps, while a research survey would be needed to understand the learning curve of programming with Crystalline’s framework, an objective comparison between the code complexity of similar use-case apps, created with and without Crystalline, can help us deduce how much harder, if any, it is for knowledgeable developers to program a Crystalline apps. To do this, we checked the number of lines of code (LOC) for three different use-case apps. Table 5.4 presents these values. We already presented the PhotoUploader, and the HeartRateApp (Local-Cloud) as test-cases in Section 5.1. The additional SimpleApp only contains a button that, when clicked, changes the text on the screen. It is similar to the example presented in Section 4.1.

We can observe that for very simple applications, Crystalline versions require more lines of code. However, as the number of resources an app accesses, its traditional Android version needs more code.

<table>
<thead>
<tr>
<th>App</th>
<th>Resources accessed</th>
<th>LOC (Traditional)</th>
<th>LOC (Crystalline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleApp</td>
<td>UI</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>PhotoUploader</td>
<td>UI, Filesystem, Internet</td>
<td>98</td>
<td>64</td>
</tr>
<tr>
<td>HeartRateApp (Local-Cloud)</td>
<td>UI, Dialog windows, Internet, Bluetooth, Mi Band 2 services</td>
<td>480</td>
<td>174</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison between the lines of code written by the application developer, for three different apps, created with and without Crystalline’s framework.
to interact with them, while the Crystalline’s version does not. This is reflected in the relatively few lines of code written for the PhotoUploader and HeartRateApp (Local-Cloud) apps developed with our framework, and it is thanks to the modularized, fine-grained Crystalline development API they leverage to interact with the multiple device’s resources. In order to control what data enters and leaves the app, Crystalline forces developers to only interact with resources from outside the graph through the trusted elements provided by native modules. The trusted elements interact with resources through Crystalline drivers, which handle all the direct communications, and then notify other appropriate trusted elements according to the information received. This leaves the application developer responsible for only defining the graphs’ structures and write the application-specific code in untrusted elements. On the other hand, it makes Crystalline’s team create and maintain modules and drivers to support multiple possible resources apps may need to interact with.

**Effort for Crystalline’s development team:** Crystalline’s framework only allows apps to communicate with the devices it supports. For example, applications can only interact with fitness tracker accessories for which Crystalline has an available API (module and trusted elements). This is the case of the Mi Band 2 Module, which provides an element to authenticate the connection with a Bluetooth connection with the device and another that provides heart rate values from this wristband. These elements understand the protocol of this specific device and thus are able to control it by sending messages within the appropriate Global Events to the Bluetooth Driver. They can also interpret the data received from the device, which may carry sensitive information. At a lower level, the Bluetooth driver maintains the connection with the wristband, sends control messages, and receives notifications.

Similarly to Crystalline’s Mi Band 2 API, trusted developers are responsible for developing support modules for contemporary gadgets that may communicate with the device, which include fitness trackers, IoT sensors and controllers, and Bluetooth speakers. Therefore, a continuous effort to expand and update Crystalline’s trusted code base is necessary to maintain it as a viable and modernized development framework. Furthermore, each module developed must respect the privacy rules enforced at runtime by Crystalline’s middleware. Like any other platform, Crystalline’s integrity relies on the uprightness of its trusted development team.

**Discussion** Crystalline’s programming differs significantly from the traditional way Android apps are programmed but also provides pre-defined trusted libraries that handle standard functionalities apps require, thus relieving developers from implementing common logic between applications. Currently, the graphs must be stated in graph descriptor files manually written by developers. However, because these files represent a description of the graphs’ structure, our framework can be easily supported by visual programming tools. For example, OutSystems [59] provides one of these tools, which supports mobile development with little code, and a visual interface to connect multiple blocks of functionalities.

We also saw that Crystalline shifts a big part of the programming effort from application developers to its trusted development team, which needs to implement multiple modules to support commonly used resources. Nonetheless, we argue that this requirement does not necessarily presents a problem. Simi-
<table>
<thead>
<tr>
<th>App</th>
<th>Benchmark method</th>
<th>Traditional (ms)</th>
<th>Crystalline (ms)</th>
<th>Overhead</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleApp</td>
<td>A</td>
<td>1556</td>
<td>2164</td>
<td>608</td>
<td>39</td>
</tr>
<tr>
<td>PhotoUploader</td>
<td>A</td>
<td>1938</td>
<td>2772</td>
<td>834</td>
<td>43</td>
</tr>
<tr>
<td>HeartRateApp (Local-Cloud)</td>
<td>A</td>
<td>1692</td>
<td>2374</td>
<td>682</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison between the time take to build traditional and Crystalline applications.

Namely to HomePad, we believe that open-source repositories, maintained by a peer-approving community, would be an efficient way of expanding Crystalline's trusted codebase while protecting it against potential attacks from ill-intended developers, or even mistakes in the code that compromise users' privacy.

5.3 Performance

In this section, we present an evaluation of Crystalline’s performance. We performed benchmark tests comparing apps developed with and without Crystalline at build-time and run-time tasks. We also measured the times the Validator took to verify the integrity of three applications, and appraise how an application’s complexity correlates with the time Crystalline Privacy Manager takes to inspect and present users with options to control their private treatment. At runtime, we compared how fast Crystalline apps are at performing multiple tasks, compared to traditional Android ones. We also present an evaluation of the impact our framework has on applications’ memory usage and battery consumption.

Methodology

All implemented build-time and validation benchmarks were executed on a computer with an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz processor. The tests we carried to evaluate the Privacy Manager’s performance, Crystalline’s impact on applications’ battery and memory consumption, and the applications runtime benchmarks, all ran on a Neffos C5A smartphone. This device has a 1 GB RAM, a 1.30 GHz quad-core processor, a 2300 mAh battery, and Android 7.0 as its operating system.

Almost all benchmarks were implemented by obtaining system clock value before and after the operation, repeating the test ten more times, discarding the first result – due to possible initialization overheads –, and calculating a representative average – method A. However, some runtime operations required a different approach. The faster the operation executes, the bigger the noise ratio in the measurement. This noise can be attributed to the network connection state, background system processes running on the device, or even the overhead of reading the clock itself. To overcome this issue, we devised a different test method for the fastest operations – method B. This approach consists of executing each operation 1000 times and registered the system clock value before the first and after the last execution. We repeated this process ten times in each test, and obtained the average time from the ten samples. Then we divided this value by 1000 to get the expected time window of one execution.
Validation benchmarks

<table>
<thead>
<tr>
<th>App</th>
<th>Benchmark method</th>
<th>Time to validate (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleApp</td>
<td>A</td>
<td>7039</td>
</tr>
<tr>
<td>PhotoUploader</td>
<td>A</td>
<td>8475</td>
</tr>
<tr>
<td>HeartRateApp (Local-Cloud)</td>
<td>A</td>
<td>7810</td>
</tr>
</tbody>
</table>

Table 5.6: Time taken for the Validator to verify the application’s integrity.

Impact on building applications  To evaluate how much time Crystalline adds to application compilation and packaging, we used an open-source Gradle plugin [60] that displays benchmark metrics of the building process. We compared build times between Crystalline and traditional versions of the SimpleApp, the Heart-rate App (Local-Cloud) and the PhotoUploader app. All benchmark tests were executed on the same computer and only took into consideration building the release type variant of each app. Each build operation was repeated ten times, and the average time calculated. As shown in Table 5.8, Crystalline applications are somewhat more slow to build. We noticed that in all cases, the overhead was mostly due to Code Transformer’s weaving process. Our weaving plugin was configured to parse all classes created by the developer, as well as the ones included by third-party libraries. However, we also noted that the overhead did not increase proportionally to the number of classes the app had, which leads us to believe that the weaving process wastes most of the time at constant initialization and configuration tasks. Nonetheless, relatively, Crystalline apps were between 39% and 48% slower to build, but taking into account we are talking about delays in the order of seconds, and it only happens when the developer changes the code, we state that this impact does not present a problem.

Performance of the Validation Process  The most noticeable delay is on registering apps on Crystalline Store Web Service, as it depends on how long the connection takes to upload the files and the time it takes to the store to verify the app’s integrity. Because the first metric does not depend on our work implementation, we do not address its impact. We do, however, evaluate how the integrity verification time varies for three use case apps. To do this, we calculated the difference between the system’s clock when the verification started and finishes. The results are presented in Table 5.6. We noticed that there for all apps, the validator took less than ten seconds to perform their integrity verification. Naturally, this process correlates which each app’s build time, as the heaviest task is to re-build the app. Unzipping the file sent by the developer with the app’s build script, source code, and third-party libraries also took a considerable amount of time. Both these observations lead us to conclude that the number of seconds required to validate app scales with its total size. We argue that having some additional seconds when publishing an app to the public or even updating an existing one, does not present a hurdle for developers, as these tasks do not happen with frequency.

Performance of the Privacy Manager  We evaluated how long Crystalline’s Privacy Manager took to inspect and present to the user the information about how the SimpleApp, the Heart-rate App (Local-Cloud), and the PhotoUploader treat one’s data. The goal of this test was to determine how the appli-
Privacy Manager benchmarks

<table>
<thead>
<tr>
<th>App</th>
<th>Benchmark method</th>
<th>Total number of elements</th>
<th>ICC</th>
<th>Inspection time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleApp</td>
<td>A</td>
<td>3</td>
<td>No</td>
<td>1923</td>
</tr>
<tr>
<td>PhotoUploader</td>
<td>A</td>
<td>6</td>
<td>No</td>
<td>2432</td>
</tr>
<tr>
<td>HeartRateApp (Local-Cloud)</td>
<td>A</td>
<td>22</td>
<td>Yes</td>
<td>6343</td>
</tr>
</tbody>
</table>

Table 5.7: Time taken for the Privacy Manager to inspect applications data-flows.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Benchmark method</th>
<th>Traditional (ms)</th>
<th>Crystalline (ms)</th>
<th>Overhead (ms)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launching SimpleApp</td>
<td>A</td>
<td>48.3</td>
<td>239.5</td>
<td>191.2</td>
<td>395.9</td>
</tr>
<tr>
<td>Launching HeartRateApp (Local-Cloud)</td>
<td>A</td>
<td>55.1</td>
<td>261.3</td>
<td>206.2</td>
<td>374.1</td>
</tr>
<tr>
<td>Launching PhotoUploader</td>
<td>A</td>
<td>51.2</td>
<td>255.5</td>
<td>204.3</td>
<td>399.0</td>
</tr>
<tr>
<td>Switching Activities</td>
<td>A</td>
<td>40.9</td>
<td>62.2</td>
<td>21.3</td>
<td>52.1</td>
</tr>
<tr>
<td>Changing text on screen upon click</td>
<td>B</td>
<td>16.6</td>
<td>16.8</td>
<td>0.2</td>
<td>1.2</td>
</tr>
<tr>
<td>Uploading 1KB file</td>
<td>B</td>
<td>11.3</td>
<td>11.8</td>
<td>0.5</td>
<td>4.4</td>
</tr>
<tr>
<td>Downloading 1KB file</td>
<td>B</td>
<td>8.3</td>
<td>9.1</td>
<td>0.8</td>
<td>9.4</td>
</tr>
</tbody>
</table>

Table 5.8: Runtime benchmarking tests results.

Application launch time overhead

Crystalline’s executes most of the necessary graph-loading logic when the app launches. It needs to parse all components graph descriptors, instantiate all elements, sandboxe the untrusted ones, and communicate with the App Launcher in order to obtain the user-defined privacy policies. We placed all these operations at launch, rather than when a component is created, because depending on the app’s complexity, this logic may have a significant time overhead. Doing it while the user navigates between the app’s Activities would impact the fluidity of the application.

To evaluate the overhead imposed by Crystalline on applications at launch time, we compared three use-case apps, each having a version developed with Crystalline, and one with the traditional Android programming model. We measured the difference between the clock values when the application is created and when the first Activity launches its onCreate method.

Table 5.8 presents the results. One can notice that Crystalline apps are considerably slower to launch.
when compared to their traditional counterparts. However, we observed that the startup overhead has little correlation to the app’s complexity (number of graphs and elements). In fact, in all cases, the cost was around 200 ms and was mostly due to the IPC interaction with the App Launcher to obtain the user-defined privacy policies. Just this task takes about 150 ms and is done in parallel with the middleware’s graph initialization operations, which takes far less time. The other reason for the startup overhead endured by Crystalline apps is due to the fact that our middleware stops the application’s first component from running, and relaunches it when the graphs are ready. Launching components are expensive operations.

While Crystalline apps suffer from a relatively high startup impact, by considering it is mostly due to operations not related to the app’s complexity, we can expect that more composite applications will not suffer from higher startup times. A survey conducted in 2015 [61] found that around 80% of users were willing to wait at least two seconds for an app to respond, and only 7% expected applications to become interactive in less than a second. The values we observed gave us the confidence to state that Crystalline does not slow apps’ launch time enough to make them unattractive to users.

**Impact on switching Activities** To grasp how much Crystalline impacts an application when switching Activities, we developed a test to verify how the HeartRateApp (Local-Cloud) developed with Crystalline compared against its traditionally programmed counterpart. The test consisted of measuring the difference between the clock values when the first Activity launched the second, and when the second Activity’s oncreate method returned. The results, presented in Table 5.8, suggest that switching Activities in Crystalline apps takes 50% more time. However, we could not observe, by simply looking, a difference between Crystalline and normal apps, when performing this task. It is hard for the human eye to notice a difference between tasks completed in 40ms and 60ms. These results are expected, as we shifted the graphs instantiation task to launch time, which is performed while the library waits for the App Launcher to obtain the user’s privacy policies. The results also allow us the state that Crystalline, while bestowing a significant relative impact on the transactions between the multiple app’s components, it does not present a usability problem.

**Impact on UI operations** Because Android apps revolve around user-interaction, evaluating Crystalline’s impact on the app’s UI operations is important. We designed a simple workflow for this test. The user clicks on a button, which changes the text of another view. We implemented the testing cases both on Crystalline and on native android programming. For the first case, we developed a simple Activity graph with the View Clicked element connected to an untrusted element, which, when notified of a click, sends a hard-coded text to a TextView TextUpdater element. The traditionally developed app consisted just of subscribing a callback for when the button was clicked and then directly invoking the TextView.setText method. This use-case is based on the use-case application SimpleApp. The results, also shown in Table 5.8, suggest that Crystalline does not impose a significant impact on UI operations.
Figure 5.4: Memory usage comparison between Crystalline apps and traditional Android ones.

Cloud communication overhead  Most applications that perform sensitive data processing involve some sort of cloud communication. We performed two benchmark tests to understand Crystalline’s overhead in both sending and receiving data to a web server, through HTTP calls. In both cases, we developed applications with and without Crystalline, and tested them on similar conditions. In all tests, the device was connected to the server’s host computer through a WiFi hotspot connection set up by the latter. The distance between these devices was also constant, as they were apart by only a few centimeters in both cases. The first test consisted of 1000 sequential downloads of a 1KB image from the server, while the second was to upload the same image to it the same amount of times. The results obtained suggest Crystalline adds approximately, in average, 785 $\mu$s on one individual 1KB download and 494 $\mu$s on uploads of the same size. Both these overheads, by themselves, do not come perceivable by the user, nor impact one’s experience. Note that the overhead Crystalline apps have on downloading is bigger than on uploading data. This is due to the fact that converting raw data from an HTTP response into graph layout event objects is more complex than the reverse process.

Memory footprint  To measure the impact Crystalline has on applications’ memory usage, we compared the HeartRateApp (Cloud) and the PictureUploader implemented with Crystalline against their respective versions developed with Android’s native programming model. These two cover different memory usage scenarios as the first app operates with small data types (heart rates), while the second handles images from the device’s gallery. The tool we used to obtain memory measurements was Android’s Memory Profiler [62], which lets developers visualize their apps’ memory usage in real-time. For each run, we launched the app, waited one minute for it to stabilize, and took a memory snapshot.

By looking at the charts presented in Figure 5.4, we can notice that Crystalline does impose an additional memory footprint on applications. While in the PhotoUploader case, this overhead is not significative, Crystalline’s version of the HeartRateApp (Local-Cloud) uses considerably more memory.
than its traditional counterpart. By inspecting in more detail, it is noticeable that total memory usage discrepancies are mainly due to code memory overheads. In the first case, the app’s version developed with Crystalline adds 2.1 MB to this memory category. In the second one, this impact got attenuated to 0.6 MB. These values are a result of Crystalline’s technique of sandboxing of untrusted elements.

As seen in Section 4.3, the sandbox feature employed by Crystalline places each untrusted element instance within a bubble, in which references to unresolved, untrusted, classes make the sandbox class loader reload them from the app’s binary files into a new definition (or copy), which is then added to the bubble. Each time a Sandbox ClassLoader is instantiated, it tells the internal Android Runtime (ART) engine to open the application’s DEX files, in which the classes bytecode is stored. When opening a DEX file, ART automatically loads into cache some of its bytecode, to optimize the runtime loading of classes. This understanding lets us deduce that Crystalline’s overhead on an app’s code memory usage scales linearly with the total number of untrusted elements specific to the application. The HeartRateApp (Local-Cloud) has a higher code memory overhead because it has five untrusted elements, while the PhotoUploader only has one. Furthermore, loading DEX files requires, at a lower level, creating buffers to read from each of these, which reflects on a slightly higher native memory usage for Crystalline apps.

Weight on Battery consumption To evaluate the impact Crystalline had on the battery drainage of supported apps, we made a direct comparison between the HeartRateApp (Local-Cloud) developed with Crystalline and its traditionally programmed counterpart, both running for one hour straight. Both apps spent the whole hour receiving heart rates from a Mi Band 2 device, updating the screen, and sending periodic requests to the cloud. To gather the necessary evaluation data, we made use of the BetterBatteryStats [63] application, which details how much battery each app consumed since the device was last charged. The results obtained suggest that both applications use around 2% of the device battery in one hour. These results are expected, as our middleware requires little processing power, since, in the long run, it purely acts as an intermediary agent between Android and the developer’s code, blocking the event bus thread when there are no messages to deliver. Therefore we can at least state that Crystalline’s middleware does not affect an application’s battery drainage in a significant amount.

Discussion In this section, we evaluated Crystalline’s performance. First, we presented results which show that the Code Transformer and the Validator make the whole process of compiling and publishing apps more time-consuming. However, causing relatively infrequent events to take some extra seconds does not seem like an issue. We observed that the time the Privacy Manager takes to inspect an application’s flows, and show users with information and control options regarding their sensitive data treatment, can be annoying for the users. Regarding runtime performance, we noted that the only significant overheads Crystalline adds to apps occurs at launch time, and when the user changes Activities. However, we have confidence that these do not affect user experience. We also observed that our framework makes applications utilize more memory due to its security measures. These results allow us to determine that, while Crystalline does not affect apps’ performance at executing their tasks, it does cause some bothersome effects. Further optimization in these aspects may be beneficial for the future.
5.4 Security Assessment

One of the requirements of this work is to guarantee the applications comply with the user's privacy policies. Our framework must uphold against the possible attacks a malicious developer can carry to process data undetected by the Privacy Manager inspection procedure and unrestricted by the user-defined privacy policies. In this section, we evaluate how Crystalline protects the user against these attacks. We consider the following scenarios:

- Untrusted elements directly interacting with the device resources – Direct access attack.
- Unconnected untrusted elements sharing data with each other – Data sharing attack.
- Malicious code set to run outside untrusted elements – Middleware bypass attack.
- Altering the app’s bytecode after the Code Transformer sanitizes it – Weaving disable attack.

**Direct Access Attack:** An untrusted element directly accessing data means obtaining it through direct calls to the Android API methods. It is paramount that Crystalline successfully prevents these attacks, as the Privacy Manager only tracks how data flows through graphs. A developer that successfully lands this attack could obtain and send out sensitive data with the user ever knowing.

Crystalline prevents untrusted elements’ code from interacting directly with Android’s API classes, thanks to the sandboxing mechanism explained in Section 4.3. When an untrusted element’s implementation class or any other class within its bubble attempts to reference a dangerous API class, the application throws a `ClassNotFoundException`. For example, if an untrusted element code tries to leak sensitive data to a webserver by directly creating a `java.net.URLConnection` client, the Sandbox Class-Loader will fail to load this class, thus preventing the leakage. Therefore, we can state that Crystalline protects users against malicious code within untrusted elements that attempt to obtain or disclose data to outside through other means than the explicit connections between the graph’s elements.

**Data sharing attack:** Two untrusted elements can not, under any circumstance, share data through other means than events carried by the explicit edges of their graph. If this were to happen, two unconnected elements could pass sensitive data between each other without this interaction being detected by the Privacy Manager. Therefore this attack can be a source of undetected leakages.

Crystalline prevents this thanks to its SandBox ClassLoader. This mechanism sandboxes different untrusted elements in separate bubbles. All classes that can be used to store state data values are resolved into different memory definitions, meaning that untrusted elements are not able to access memory shared with each other. For example, if an untrusted element receives a sensitive data value and stores it in the class X’s field, other untrusted elements may reference class X, but will always read a nil value from the field which the first element stored data in.

**Middleware bypass attack:** Crystalline’s programming model works under the assumption that the application-specific code is all executed from the untrusted elements’ event receiving methods. Cryst-
talline can not allow any developers’ code to run outside these. For instance, having code operating on an Activity’s onCreate method gives the developer direct access to Android’s API, which he or she can use to collect or send out sensitive data, without it ever passing through the graph, and thus invisible to the data-flow inspection and invulnerable to enforcement of the user-defined privacy policies.

The Code Transformer weaves all the application classes. By leveraging from AOP capabilities, this tool injects safety guard instructions across the application classes that make sure that no developer’s code is directly invoked by Android. Therefore, Crystalline only allows applications’ to process sensitive data within their graphs’ environments, which enables our framework to transparently convey to users how their private data flows within the app, and to where it is sent to.

**Weaving disable attack:** Because the Code Transformer weaves the app on the developer’s computer. Some motivated malicious developers may disable the AOP Advices injected by the Code Transformer in order to execute code outside untrusted elements, or even tamper with Crystalline’s Runtime Middleware classes to disable the Sandboxing of untrusted elements. However, Crystalline’s App Certification Service only certifies apps that are correctly weaved by the Code Transformer, and use a clean middleware library. While developers can publish adulterated applications to Google Play, they will not appear available for users in the App Installer. This means Crystalline does not protect users’ privacy from apps that falsely state being supported by our framework on Google Play, but are not available on the App Installer. Nonetheless, the App Installer is a fundamental part of our architectural design, and we assume users’ are aware to only install Crystalline apps from it.

**Discussion** This section covered the possible attacks from within the applications’ code. However, some attacks may come from other channels than malicious apps. To guarantee that users only install trustworthy apps, the online Crystalline App Certification Service must maintain its integrity at all times, and thus requires a constant effort by a security team to protect it against attackers. Furthermore, as stated in Section 3.2, our framework assumes that the device runs with an original distribution of Android, as altering the underlying Android Framework could make Crystalline lose control over applications. Nonetheless, our work protects the user against malicious developers that try to circumvent our framework rules to perform undetected leakages of sensitive data or bypass the user-defined privacy policies configured in Crystalline’s Privacy Manager and enforced by its runtime middleware.

5.5 Summary

In this chapter we evaluated Crystalline’s effectiveness at making applications’ data-flows explicit to users, and at providing a fine-grained privacy control mechanism. We also addressed the effort expected from developers to create apps, and the responsibilities bestowed on Crystalline’s development team. We proceeded to cover the performance impact our middleware has on applications, and finished with an assessment of how Crystalline upholds against attacks by malicious developers. The next chapter finishes this dissertation by summarizing the document and presenting directions for future work.
Chapter 6

Conclusions

While the widespread adoption of smartphones allowed society to interconnect in a way never seen before, the interactions with these devices continue to bring new privacy concerns for users. The multiple functionalities offered by mobile applications currently on the market many times involve different processing types of sensitive data, which results in users losing track of who obtains their data and what specific information is disclosed. Furthermore, with the adoption of the GDPR by the EU, companies that develop mobile applications are now legally liable for how they treat their end-users private data. This means they also require means to inform users on how exactly they process sensitive data and must oblige by the contractual statements disclosing what data they collect and how it is processed.

The current solutions for Android environments either do not correctly notify users which data each application collects, or lack the means to inform and control what happens to the data after the app obtains it. Moreover, with the current programming model for Android applications, developers can overlook some violations in their code when handling their end-users sensitive data, which could make them liable for not following the privacy agreements presented to users.

In this work, we proposed Crystalline, a privacy-aware middleware that helps both users and developers safeguard the former's privacy. Crystalline adopts a programming model proposed for IoT applications in smart home environments, which allows developers to transparently expose how they treat user’s sensitive data by developing apps with a data-flow oriented programming model. For users, Crystalline enables them to understand how their private data flows through each application, and where it ends up at. With our framework, users do not have to assume that granting an app access to a sensitive data type means losing ownership over it. Crystalline allows users to define fine-grained privacy policies for each app, that control what data apps may access, and where they can send that data to. Additionally, our framework can support data obfuscation or anonymization functionalities that allow users to make applications transform their sensitive data before disclosing it to others.

Our evaluation of the framework shows that Crystalline successfully informs the user on how Android applications treat sensitive data and supports fine-grained control mechanics to determine what the app can, and can not do, with his or her private data. A comparison between the Android programming model and Crystalline’s shows that developers can create apps with our framework with relatively little
effort. It can also make application development more accessible, as it can be easily integrated with a visual programming tool. We also noted that our middleware impacts applications mostly at launch time, but still does not present a significant usability problem. Furthermore, a security assessment of potential attacks shows that Crystalline successfully protects the user’s privacy when using supported applications developed with malicious intents.

6.1 Future Work

Currently, we identified two possible directions for future work. First, both Crystalline’s Runtime Middleware and the Privacy Manager’s Prolog engine were developed with Android’s SDK, to run on top of the ART Java VM. While this approach does not compromise the overall performance, migrating the graphs’ startup logic to run in native code (C/C++) could allow us to change Crystalline’s SandBox Classloader to reutilize the ART low-level loaded memory buffers for DEX Files. This approach is not evasive, as Android only considers security restrictions at the process boundaries, and not between the VM and its runtime engine. Furthermore, the Privacy Manager uses tuProlog [57] as its Prolog engine, which was developed to provide an extensive API for Java applications. Adapting Crystalline to work with a prolog engine aimed at a faster performance could improve the time to inspect applications.

Additionally, the current Crystalline Development Framework was implemented as a proof of concept. There are still many functionalities that we currently do not support, but can be added by trusted developers. These functionalities include WiFi Direct, Google/Facebook account integration, some UI operations, and other fitness tracker’s APIs. Following the same logic, new trusted modules can be added to support transformation policies for specific types of data. E.g., obfuscation of the user’s location, automatically hashing passwords, replacing the contact’s phone numbers with false values.
Bibliography


[36] Malicious android app had more than 100 million downloads in google play. [Accessed on September 8th, 2019].


[61] H. PACKARD. Failing to meet mobile app user expectations: A mobile user survey. Dimensional Research, Feb 2015.


Appendix A

Use Case Apps

A.1 SimpleApp

This simple application detects when the user clicks a button on the screen, and increments a counter, also displayed on the screen. It is comprised of only one Activity, which screen is presented in Figure A.1. Figure A.2 displays this Activity’s flow graph, which has three elements. Two trusted and one untrusted. Listing A.2 presents its graph descriptor. The View Clicked element notifies the untrusted Handle Click element when the user clicks on the button. This last element implementation is presented in Listing A.2. Notice that it increments a variable that counts the number of times the button was clicked. Every time it receives an event, it also sends the new text to the screen, updating the current displayed counter number.

Figure A.1: SimpleApp’s Activity 1 screen.
Figure A.2: SimpleApp’s Activity 1 flow graph.

```java
public class Activity1GraphDescriptor extends ActivityGraphDescriptor {
    public Activity1GraphDescriptor() {
        setActivityClass(Activity1.class);
        requireModule(ViewModule.class);
    }
    @Override
    public GraphStructure defineGraphStructure() {
        return new GraphStructureBuilder()
            .defineElement("HandleClick", HandleClickClass.class)
                .connect(ViewModule.Click(R.id.button), 0, "HandleClick", 0)
                .connect("HandleClick", 0, ViewModule.TextUpdater(R.id.textview), 0)
            .build();
    }
}
```

Listing A.1: SimpleApp’s graph descriptor.

```java
@CustomElement(name="HandleClick")
public class HandleClickClass extends Element {
    @EventReceiver
    public void onEvent(LayoutEvent event) {
        sendEvent(new Event<String>("Click count: " + (++counter)));
    }
}
```

Listing A.2: SimpleApp’s Handle Click untrusted element implementation.
A.2 PhotoUploader

The PhotoUploader App is a straightforward app. As displayed in Figure A.3, it allows the user to pick an image from the device’s gallery, and upload it to the image hosting service imgur.com. Its graph is displayed in Figure A.4, and the graph descriptor that defines it is presented in Listing A.3. One can notice that this app only handles one type of sensitive data – a gallery image. When a "PICK PICTURE" button is clicked, the Gallery Pick Image element presents the user with its device gallery, so he or she can select an image to disclose to the app. Once the picture is selected, then it is forwarded to the View Draw On Surface trusted element, which displays it on the screen’s Surface View with the "pictureSf" ID. The untrusted Picture Manager element, implemented as presented in Listing A.4, is also connected to the Gallery Pick Image output, which means it also receives the picture. When the "UPLOAD" button is clicked, the Picture Manager prepares an HTTP request to upload the image. The request is executed by the HTTP Post element, which was configured in the graph descriptor to communicate with imgur.com.

Figure A.3: PhotoUploader window.

![PhotoUploader window](image)

Figure A.4: PhotoUploader Main Activity flow graph.

![PhotoUploader flow graph](image)
public class MainActivityGraphDescriptor extends ActivityGraphDescriptor {
    static String imgurUrl = "https://api.imgur.com/endpoints/image/";

    public Activity1GraphDescriptor() {
        setActivityClass(MainActivity.class);
        requireModule(ViewModule.class);
        requireModule(FileSystemModule.class);
        requireModule(HttpModule.class);
    }

    @Override
    public GraphStruct defineGraphStructure() {
        return new GraphStructBuilder(this)
            .defineElement("PictureManager", PictureManager.class)
            .defineElement("PickPhoto", FileSystemModule.GalleryPickImage)
            .connect(ViewModule.Click(R.id.pickPhotoBtn), 0, "PickPhoto", 0)
            .connect("PickPhoto", 0, ViewModule.DrawOnSurface(R.id.pictureSf), 0)
            .connect("PickPhoto", 0, "PictureManager", 0)
            .connect(ViewModule.Click(R.id.uploadBtn), 0, "PictureManager", 1)
            .connect("PictureManager", 0, HttpModule.Post(imgurUrl), 0)
            .build();
    }
}

Listing A.3: GraphDescriptor file for the PhotoUploader’s MainActivity.

@CustomElement(name="PictureManager")
public class PictureManager extends Element {
    private Bitmap data = null;

    @EventReceiver(port = 0) /*Store picked image*/
    public void onImagePicked(LayoutEvent event) {
        data = ((GalleryPicturePortType) event).getData();
    }

    @EventReceiver(port = 1) /*Upload image*/
    public void onUploadBtnClicked(LayoutEvent event) {
        String encoded = ... /* convert data into a Base64 string */
        HttpHeaders s = new HttpHeaders();
        s.addParam("image", new ObjectPortType<String>(encoded));
        sendEventToPort(s, 0);
    }
}

Listing A.4: Implementation of the PhotoUploader’s PictureManager untrusted element.
A.3 HeartRateApp

We developed three very similar applications that rely on collecting the user’s heart rate from a Xiaomi Mi Band 2 fitness tracker. In this section we present the implementation for these applications.

All three applications are composed of 2 Activities, and consequently two graphs. Presented in Listing A.5, is the first Activity’s (FindMiBandActivity) graph descriptor implementation, which is the same for all apps. Its goal is to scan nearby Bluetooth devices in search of a Mi Band 2 device to connect to. Once found, the second Activity is launched, which connects to device and obtains heart rates from it. FindMiBandActivity's graph does not handle sensitive data, therefore we do not present its flow graph. However, each of the three versions of this app has a different graph for the second Activity (MiBand2Monitor). The HeartRateApp (Local) only processes the user’s heart-rate locally. The HeartRateApp(Cloud-Local) also handles this type of data locally but requires network connectivity. Finally, the HeartRateApp(Local-Cloud) app sends the heart rate to two cloud servers.

```java
public class FindMiBandActivityGraphDescriptor extends ActivityGraphDescriptor {
    public FindMiBandActivityGraphDescriptor() {
        setActivityClass(FindMiBandActivity.class);
        requireModule(ActivityLifeCycleModule.class);
        requireModule(BlModule.class);
        requireModule(DialogModule.class);
        requireModule( имеются другие модули от других пакетов).
    }

    @Override
    public GraphStructure defineGraphStructure() {
        return new GraphStructBuilder().
            .defineElement("FindMiBand2", FindMiBand2El.class)
            .connect(ActivityLifeCycleModule.ActivityResumed, 0, BlModule.ScanDevices, 0)
            .connect("FindMiBand2", 0, ICMModule.LaunchActivity(MiBand2Monitor.class), 0)
            .connect("FindMiBand2", 1, ViewModule.VisibilityUpdater(R.id.progressbar, 0)
            .connect("FindMiBand2", 2, DialogModule.Dialog("DeviceNotFound", 0)
            .connect(DialogModule.Dialog("DeviceNotFound"), 0, BlModule.ScanDevices, 0)
            .connect(DialogModule.Dialog("DeviceNotFound"), 0, ActivityLifeCycleModule.FinishActivity, 0);.
        .build();
    }
}
```

Listing A.5: HeartRateApp’s FindMiBandActivity’s graph descriptor.

HeartRateApp (Local): The HeartRateApp (Local)’s MiBand2Monitor Activity connects to a MiBand2 device, authenticates the connection, and obtains heart-rate values from this fitness tracker. It relies on elements from the Bluetooth module and the Mi Band 2 API module, as well as others from the Dialog Module, the Activity Life Cycle Module, and the View module. Because this graph is somewhat complicated, we figure it is best to show its visual representation than its descriptor file. The graph for this application’s MiBand2Monitor Activity is represented in Figure A.5. The output ports are displayed as numbers next to the elements. Notice that there are no trusted elements that disclose the data to the cloud.
outside – e.g., the cloud–, which means this app only processes data locally. We can also observe the flows of the only sensitive data type being processed – the user’s heart rate.

When the Progress Handler element is notified that the app authenticated itself with the Mi Band 2 device, it sends an event to the MiBand2 Heart Rate trusted element. This last element then sends a signal to the Mi Band 2 device to start streaming heart rate values. The received values are also interpreted at the MiBand2 Heart Rate element, which sends them to the TextView text Updater element. Then, as presented in Figure A.6a, they are shown on the screen, specifically on the layout’s TextView with the ID “heartRatev.” Additionally, the MiBand2 Heart Rate element also sends heart rates to Compare With Reference. In this element, the developer implemented an application-specific algorithm, which compares the user’s heart rate value with reference values. As presented Figure A.7. If the heart rate is outside the reference bounds, then an alert is displayed by the AlertDialog Show element.

Figure A.5: HeartRateApp (Local) MiBand2Monitor Activity flow graph.

Figure A.6: MiBand2Monitor screens.
HeartRateApp (Cloud-Local): The HeartRateApp (Cloud-Local) behaves similarly to the HeartRateApp (Local), but the reference used to determine if the user’s heart is beating within normal values is obtained from a cloud server. This application’s MiBand2Monitor Activity’s graph is depicted in Figure A.7. Notice the HTTP Get element, which receives the reference value from healthbuddy.org, and sends it to the CompareWithReference element. The element then proceeds as before, comparing the heart rate values received from the Mi Band 2 Heart Rate element with the reference and dispatching a warning if needed.

![HeartRateApp (Cloud-Local) MiBand2Monitor Activity flow graph.](image)

HeartRateApp (Local-Cloud): The HeartRateApp (Local-Cloud) sends the user’s heart rates to two cloud servers. Figure A.8 presents this app’s MiBand2Monitor Activity graph. Notice that this graph explicitly shows that the app sends the user’s heart rate to healthbuddy.org and xyz.com, as these data type flows to two HTTP Post elements, one for a connection with each webserver. The Cloud Call Prepare elements prepares the HTTP request to be executed by the Post elements.

![HeartRateApp (Local-Cloud) MiBand2Monitor Activity flow graph.](image)