Crystalline: A Privacy-Aware Middleware for the Android Platform

(extended abstract of the MSc dissertation)

Eduardo Joaquim Leitão Libânio Gomes
Departamento de Engenharia Electrotécnica e de Computadores
Instituto Superior Técnico

Advisors: Professors João Silva and Nuno Santos

***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. 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. 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 home environments into an Android compatible framework that does not require any modifications to the OS while still securing the user's personal privacy policies for each application. An experimental evaluation shows that Crystalline gives users control over their data without compromising the smooth running of applications.**

I. Introduction

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, which 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.

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 [1], [2], 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 [3]. However, some studies [4], [5], [6], [7] 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.

On the one hand, improvements to the way Android handles privacy would mostly benefit users. On the other hand, 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 [8].

Our main goal with this work 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 is able to 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 allow 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 middleware does not add a noticeable impact to user experience, but blocks the processing of sensitive data against the user privacy policies.

II. Related Work

The most revelant proposed solutions to improve privacy in Android devices can be devided in to two categories: Data-flow analysis tools to inspect apps either staticly or at runtime, and extensions to Android's permission system that provide

a finer-grained mechanism to determine what data an app has access to.

A. App Data-flow analysis tools

The main goal of the data-flow analysis tools is to inform users about possible leaks or dangerous behavior by applications when treating sensitive data. Most of these approaches [9], [10], [11], [12] 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 [13] 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 [9], [10], [12] rely on static taint-analysis of applications' code, which may compromise the results. A study carried in 2018 [14] shows that FlowDroid and IccTA fail to track flows of sensitive data when they involve ICC calls with complex strings formed from sensitive data. TaintDroid [11] and TaintART [15] achieve better results at identifying leaks, but require changing Android's core. Furthermore, these tools rely on taint-analysis at a variable level, which can be tricked by a malicious developer [16].

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. However, some solutions give users more control over their data by extending Android's permission system, which we discuss next.

B. Extensions to Android's Permission System

Almost all the proposed solutions to improve Android's permission system [17], [18], [19], [20] 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 monitor how data flows within applications [21], [22], rely on TaintDroid [11] 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 [23], [24], [25], [19], [20], [21], [22] involve changes to Android's core. Aurasium [17] and AppGuard [18] 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.

C. HomePad

HomePad [26] presents 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 [27], 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 abdicate their data into an untrusted environment. HomePad, on the other hand, has applications running on the smart home's hub. Furthermore, 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 they do, and if they threaten one's privacy concerns. The user can then make the informed decision to whether or not disable some of the app's functionalities according to his privacy preferences. Naturally, the ability to define privacy policies for functionalities, instead of the whole applications, allows the user to have much finer-grained control over his sensitive data.

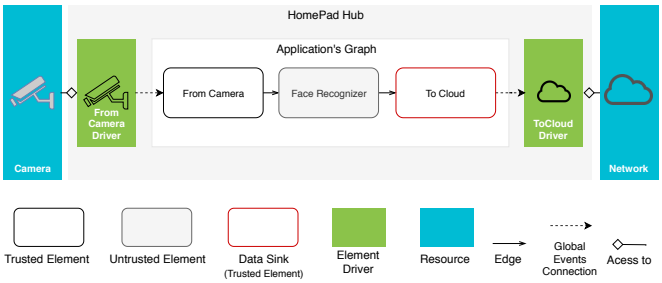


Figure 1: Example HomePad application graph.

As displayed in Figure 1, HomePad apps must be structured as a directed graph of multiple connected elements. The edges that connect them express the possible data paths, as HomePad only allows data to flow between connected elements.

HomePad is designed as a middleware between apps and the hub's operating system. As presented in Figure 2, 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 and drivers 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. 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, cloud communications, and actuators.

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

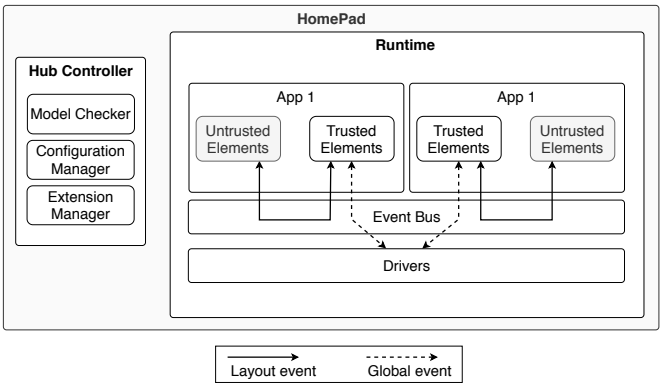


Figure 2: HomePad architecture (adapted from [26]).

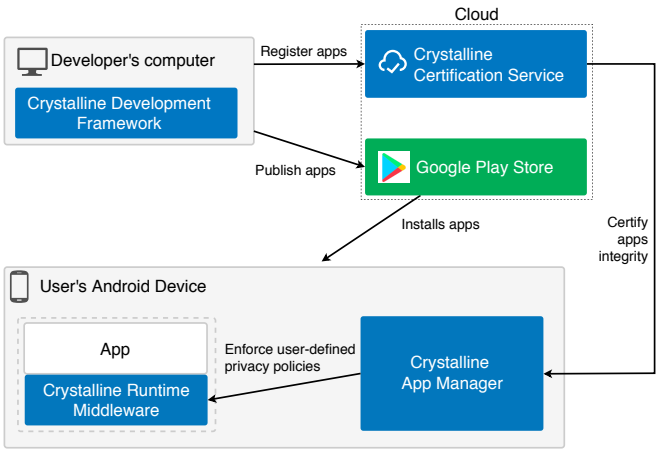


Figure 3: Overall Crystalline framework architecture.

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.

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.

III. Architecture

Our work goals require Crystalline to be a trusted medium between users and application developers. On the one hand, developers are responsible for developing their applications with our framework and its rules. On the other hand, Crystalline explicitly informs the user how apps treat his privacy while also allowing one to define personal privacy policies applicable to each app. As presented in Figure 3, 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.

A. 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 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.

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 the app has, 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 within graphs. This does not necessarily mean they have to write the file manually, as there can be some IDE feature that has a graphical panel to drag, drop and connect elements, and generate the file accordingly.

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 on runtime. Crystalline Runtime Middleware automatically does this by checking if the trusted element's functionality

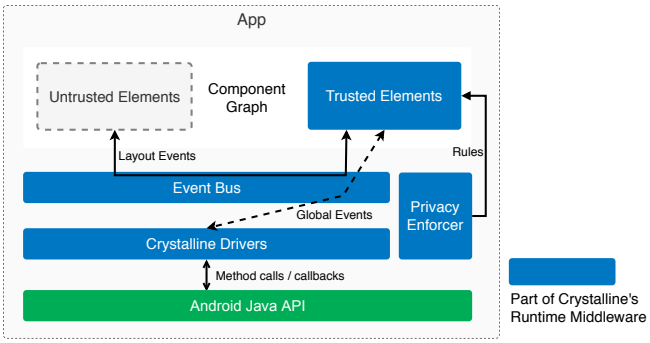


Figure 4: Runtime architecture of a Crystalline application.

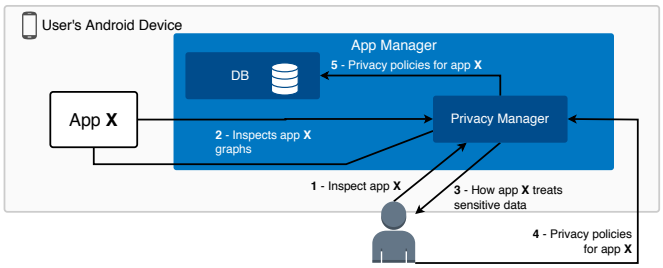


Figure 5: Process of defining privacy policies for a Crystalline app.

requires any of these permissions, not yet granted. If so, Crystalline asks the user to concede them before the trusted element operating. If the user does not grant permission, then the respective element outputs an error message, which other elements may handle.

B. 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.

Crystalline's Runtime Middleware (depicted in Figure 4) is in charge of controlling the application's graphs. It is comprised of an event bus, trusted elements, and a pool of drivers. Elements communicate between themselves through graph layout events, while drivers and trusted elements do it though global events. Nonetheless, while HomePad runs all apps on top of a single runtime engine, 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. E.g., 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 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.

C. 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.

The Privacy Manager runs a HomePad-based flow inspection of applications, adapted to Android. User-interaction with this app follows the workflow logic presented in Figure 5. While inspecting a graph's data flows, if it 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. If circular dependencies are found, then the flow is skipped, as it definitely does not carry sensitive data, since any sort of this information must arrive from outside the apps' graphs. Once the algorithm finished, the user is presented with what types of sensitive data the app processes and where it sends them to. Users can then enforce privacy policies of the following categories:

- Sink-blocking policies, in which the user stops the data from being exposed to a particular sink.
- Source-blocking policies, in which the users blocks the app from obtaining a data type.
- "Ask first" policies, in which the user forces the applications to request permission every time it attempts to obtain, or disclose, a particular data type.
- Data transformation policies, in which the user only lets the app send some types of data to the cloud it they are transformed first.

D. Certification Mechanism

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. 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.

As shown in Figure 6, 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

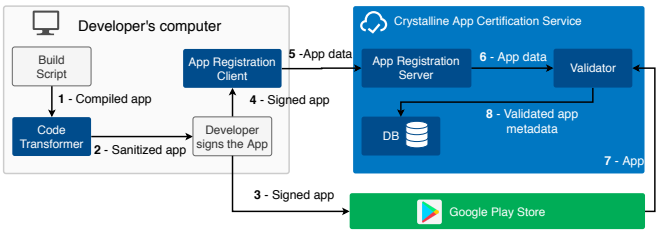


Figure 6: Certification process for Crystalline apps.

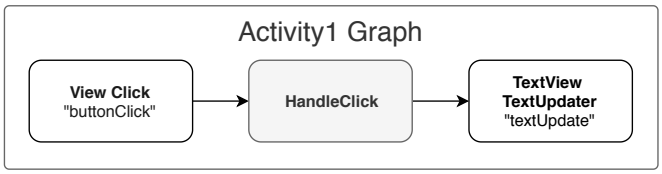


Figure 7: Example Activity Graph.

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.

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.

IV. Implementation

A. Development of a simple app

Developers do not have to, nor shall, write code directly in the components classes, as these are injected with AOP Advices to deviate their methods invocations to the middleware's classes. The graph descriptor files should be implemented as a Java subclasses of Crystalline's API ActivityGraphDescriptor or ServiceGraphDescriptor classes, according to the component each one represents. A graph descriptor must state the elements that comprise the graph and how they are interconnected.

To understand how developing Crystalline apps deviate from standard Android programming, let us consider a simple

```
@CustomElement(name="HandleClick")
public class HandleClick extends Element{
    int counter = 0;
    @EventReceiver
    public void onEvent(...) {
        sendEvent(new Event<String>("
            Count: " + (++counter)));
    }
}
```

Listing 1: HandleClick element implementation.

example, in which 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.

As depicted in Listing 1, the developer should first implement an element that outputs the updated text when receiving a graph event. Then, on the Activity's graph descriptor, one must import the View module, since the graph uses UI functionalities. The implemented element should be referenced in this file, and linked to its implementation 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 and the TextView Update Text to the TextView's. Finally, the developer should connect the graph in a way that reflects the intended behavior. Figure 7 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.

B. Runtime Middleware

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.

Figure 8 presents a UML representation of the Runtime Middleware implementation. Note that this image is not completely faithful, as it omits multiple driver classes, and does not include the trusted elements developed, 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

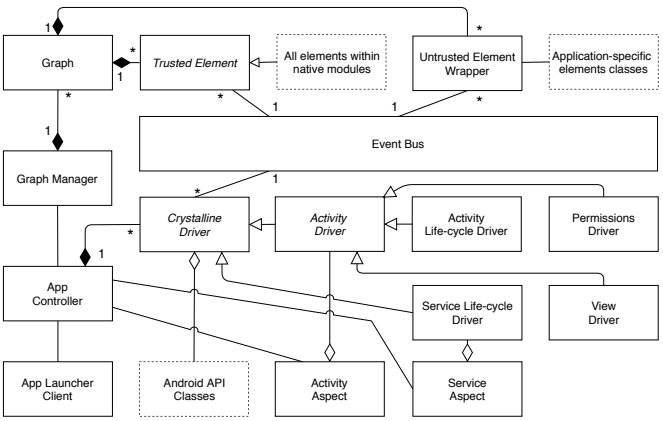


Figure 8: Simplified UML representation of Crystalline's Runtime Middleware implementation.

is launched. Then, the App Controller loads the middleware in two parallel tasks: 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. While waiting for a response, it 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, and enable or disable them during the app's lifetime, according to the components active at each moment. Once the App Launcher Client receives the user-defined privacy policies, it forwards them to the Graph Manager, which implements the Privacy Enforcer's functionalities by reconfiguring the Graphs to oblige by the policies

C. Sandboxing Untrusted Elements

By exploring Java's class loading model, we developed a Sandbox Classloader in charge of resolving classes within an isolated namespace, which we refer to as a bubble. 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. 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. Finally, **unsharable classes** are, in practice, the ones created by the application developers or even brought by in third-party-libraries. Crystalline does not trust these classes, as they may attempt to incur on 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.

D. AOP Weaving of Applications' code

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 it being sandboxed. This presents a severe vulnerability which may be responsible for data leakage or illegal processing in general. 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. With the help of Aspect-Oriented Programming (AOP), it is possible to define a set of executions 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.

Crystalline's Code Transformer was implemented as a Gradle plugin, which developers should use to weave their apps automatically. 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.

E. Certification Service

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 9 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 Crystalline applications information for the App Installer.

Once the developer submits the app, the Validator verifies if the application does not tamper with Crystalline's Runtime

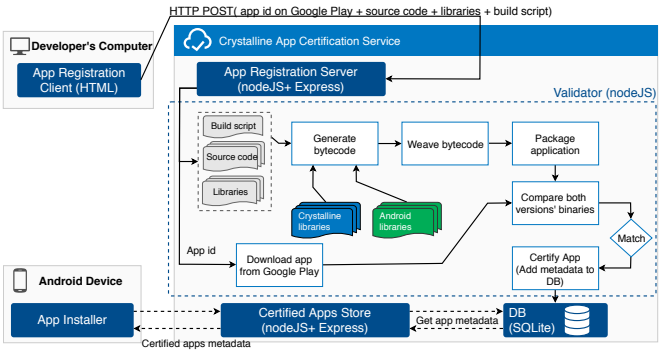
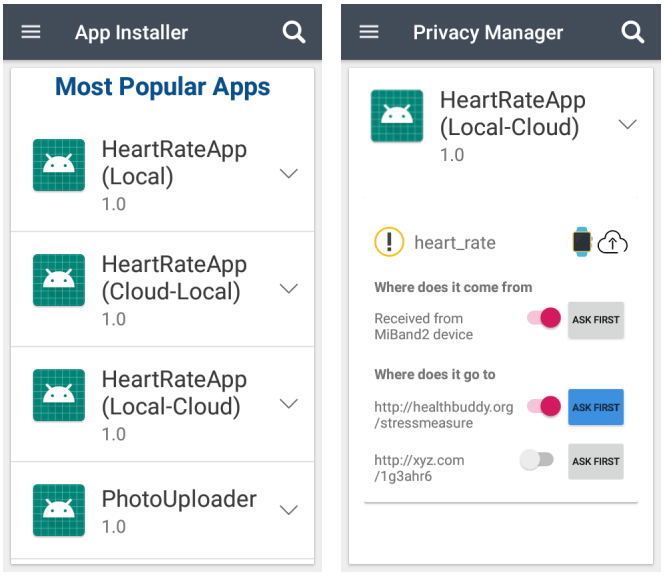


Figure 9: Certification process representation.



(a) App Installer main screen. (b) Privacy report of an app.

Figure 10: App Manager windows.

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 Pay 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 them to 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 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 comprised of the ten most popular Crystalline apps. Another to 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.

F. App Manager

The App Installer, App Launcher, and the Privacy Manager were all implemented part of the App Manager. Figure 10 presents two windows of the Privacy Manager user interface. The first one belongs to the App Installer. The second one displays the Privacy Manager's policies definition user interface.

When launched, this Android app'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 implemented as a Service.

As presented in Figure 10a, 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.

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 that connects to Android resources, and not other graphs. Crystalline adopts tuProlog [28] 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.

As presented in Figure 10b, 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.

V. Evaluation

We evaluate Crystalline in four fronts. First its effectiveness on improving the feedback and control mechanisms Android users have at their disposal. Second, how Crystalline's programming model compares to traditional Android programming. Third, the performance of our framework, and the impact it has on applications. Finally, we present a security assessment of our system.

A. Effectiveness

We observed what Crystalline's privacy manager showed us when analyzing a use-case app that sends heart rate values to two cloud servers, and compared with the information obtained from Android's permission system. Crystalline informed us with three statements: 1) the app obtains heart-rate values, 2) sends them to healthbuddy.org, and 3) sends them to xyz.com. For each of these three statements, we were presented with options to either disable the procedure or make the app ask for permission before doing it. On the other hand, when verifying the permissions granted by Android, in the device settings menu, they only state that 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 Internet.

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.

App	**Resources accessed**	**LOC (Traditional)**	**LOC (Crystalline)**
SimpleApp	UI	15	21
PhotoUploader	UI, Filesystem, Internet	98	64
HeartRateApp (Local-Cloud)	UI, Dialog windows, Internet, Bluetooth, Mi Band 2 services	480	174

Table I: Comparison between the lines of code written by the application developer, for three different apps, created with and without Crystalline's framework.

B. Programming Effort

As presented in Table I, for more complex apps, Crystalline programming model does not require as many lines of code as traditional Android. Because Crystalline apps must rely on trusted elements to interact with the outside, developers are relieved from implementing connector classes. This shift some programming responsibility from application developers to Crystalline's trusted development team, which need to implement multiple modules to support commonly accessed resources. Nonetheless, we argue that this requirement does not necessarily present a problem. Similarly 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 developers with malicious intentions, or even accidental mistakes in the code that compromise users' privacy.

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 benchmark tests we carried to evaluate the Privacy Manager's performance, the memory and battery evaluations and the runtime benchmarks ran on a Neffos C5A smartphone. This smartphone comes with a 1 GB RAM, a 1.30 GHz quad-core processor, a 2300 mAh battery, and Android 7.0 Nougat as its operating system.

C. Performance

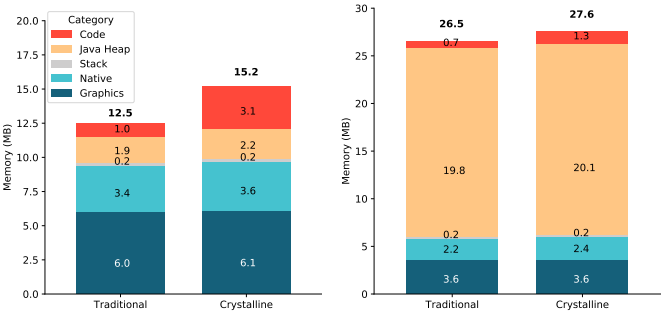
We evaluated the performance of Crystalline's implementation. First, we noticed that the Code Transformer and the Validator make the process of compiling and publishing take some more seconds. However, causing relatively infrequent events to take some extra seconds does not seem like an issue.

We then observed 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. We noticed that the Privacy Manager took between two and seven seconds to analyze application-flows, depending on the app's complexity.

Regarding runtime performance, as presented in Table II, we noted that the only significant overheads Crystalline adds to apps occurs at launch time, and when the user changes Activities. Crystalline adds, on average, 200ms to an application's launch time, and 20ms when it switches Activities.

Runtime benchmarks			
Operation	**Traditional (ms)**	**Crystalline (ms)**	**Overhead (ms)**
Launching SimpleApp	48.3	239.5	191.2
Launching HeartRateApp (Local-Cloud)	55.1	261.3	206.2
Launching PhotoUploader	51.2	255.5	204.3
Switching Activites	40.9	62.2	21.3
Changing text on screen upon click	16.6	16.8	0.2
Uploading 1KB file	11.3	11.8	0.5
Downloading 1KB file	8.3	9.1	0.8

Table II: Runtime benchmarking tests results.



(a) HeartRateApp (Local-Cloud) (b) PhotoUploader

Figure 11: Memory usage comparison between Crystalline apps and traditional Android ones.

However, we have confidence that these do not impact user-experience enough to make applications unattractive to users. We noted that Crystalline apps do not have a slower UI responsiveness, and take between 500 and 800 μs more time to process HTTP requests. As presented in Figure 11, our framework makes applications utilize somewhat more memory due to the sandboxing mechanism, which replicates classes bytecode definitions, increasing the amount of code memory used the application process. We also noted that our runtime middleware does not make an application consume more battery, in a significant amount.

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.

D. Security

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 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:

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

The sandboxing mechanism prevents the first and second attack, blocking dangerous classes that grant access to the device's resources from being loaded and copying untrusted class definitions for each untrusted element so that they do not share memory. The Code Transformer weaves all the apps bytecode, preventing attacks of the third kind. The Certification Service only validates apps that were correctly sanitized by the Code Transformer, preventing attacks of the fourth type.

VI. Conclusion

In this work, we proposed Crystalline, a privacy-aware middleware that helps both users and developers safeguard the formers' 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 without an extraordinary amount of 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.

References

[1] "Play protect," https://developers.google.com/android/play-protect, [Accessed August 4th, 2019].

[2] "Application sandbox," https://source.android.com/security/app-sandbox, [Accessed August 30th, 2019].

[3] "Permissions overview," https://developer.android.com/guide/topics/permissions/overview, [Accessed August 30th, 2019].

[4] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, 06 2012.

[5] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, pp. 3:1–3:14.

[6] N. Eling, S. Rasthofer, M. Kolhagen, E. Bodden, and P. Buxmann, "Investigating users' reaction to fine-grained data requests: A market experiment," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, Jan 2016.

[7] R. Balebako, J. Jung, W. Lu, L. Cranor, and C. Nguyen, "Little brothers watching you: raising awareness of data leaks on smartphones," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, 2013.

[8] Y. Martin and A. Kung, "Methods and tools for gdpr compliance through privacy and data protection engineering," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2018.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014.

[10] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, 2015.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2010.

[12] Z. Meng, Y. Xiong, W. Huang, L. Qin, X. Jin, and H. Yan, "Appscalpel: Combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in android applications," *Neurocomputing*, vol. 341, 2019.

[13] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017.

[14] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018.

[15] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[16] G. S. Babil, O. Mehani, R. Boreli, and M. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *2013 International Conference on Security and Cryptography (SECRYPT)*. IEEE, 2013.

[17] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, 2012.

[18] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard: Enforcing user requirements on android apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2013.

[19] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipshield: A framework for enforcing context-aware privacy," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014.

[20] D. Wu and S. Bratus, "A context-aware kernel ipc firewall for android," in *Proceedings of ShmooCon*, vol. 17, 2017.

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011.

[22] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," in *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*. Springer-Verlag, 2012.

[23] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010.

[24] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 2011.

[25] Z. Xu and S. Zhu, "Semadroid: A privacy-aware sensor management framework for smartphones," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015.

[26] I. Zavalyshyn, N. O. Duarte, and N. Santos, "Homepad: A privacy-aware smart hub for home environments," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[27] "SmartThings." https://www.smartthings.com, [Accesed on August 11th, 2019].

[28] E. Denti, A. Omicini, and A. Ricci, "tuprolog: A light-weight prolog for internet applications and infrastructures," in *Practical Aspects of Declarative Languages*, I. V. Ramakrishnan, Ed. Springer Berlin Heidelberg, 2001.