Figment: Fine-grained Permission Management for Mobile Apps

Ioannis Gasparis*, Zhiyun Qian*, Chengyu Song*, Srikanth V. Krishnamurthy*, Rajiv Gupta* and Paul Yu†

*Department of Computer Science and Engineering, University of California, Riverside, †U.S. Army Research Lab
igasp001@ucr.edu, zhiyunq@cs.ucr.edu, csong@cs.ucr.edu, krish@cs.ucr.edu, gupta@cs.ucr.edu, paul.l.yu.civ@mail.mil

***Abstract*—Today's Android systems do not allow users to manage the permissions granted to applications (apps) in a flexible and dynamic way. Recent studies show that apps often misuse these permissions to access private information, or have trapdoors via which other malicious apps can do the same. In this paper, we develop a framework *Figment*, which consists of set of libraries that developers can easily use to build in fine-grained dynamic permission management capabilities. The users of their apps can readily invoke these capabilities during execution. The apps would potentially run with reduced functionalities if the user does not wish to allow certain permissions. *Figment* also allows either the developer or a user to specify context aware permissions, which cause different permissions to be granted to the app in different functional modes (contexts). We believe that *Figment* reduces the attack surface exposed to potentially malicious apps and offers a significant step in preserving user privacy. While the rudimentary version of *Figment* uses aspect-oriented programming and does not need rooting of the phone or changes to the Android sub-system, we also provide an optional root-level fail safe implementation that facilitates the embedding of dynamic permission management functions in old applications not built by using *Figment* libraries. We show that *Figment* offers significant benefits over the Android Marshmallow permission management system with lower runtime overheads; the main penalty is a one time higher compilation overhead.**

I. Introduction

Unfortunately, today most mobile apps declare (sometimes many) permissions at install time and retain these permissions for their lifetimes. Not only are many of these permissions seldom used, they expose a large attack surface that can potentially compromise user privacy. A recent report [6] examined over a million apps available on the Google Play Store [10] and found that on average each app asks for 5 permissions. Most apps require permissions for network access (83%), for modifying data on local disk (54%), for reading phone status and identity (35%), for accessing the precise user location (24%), for finding accounts (e.g., Facebook) of the user (16%) and accessing the camera (12%). A combination of these (often permanent) permissions allows an app to compromise the privacy of the user (information is accessed and transferred over the network); a poorly developed app could allow other malicious apps to exploit its permissions [1], [14]. The aforementioned study [6] also surveyed over 400 adults and found that over 60% of the surveyed population decided not to install an app because it requires many permissions and hence, they were concerned about their privacy. Although many such app requests could be legitimate, the users might have trusted them if they could enable/disable permissions dynamically.

Android Marshmallow and what it brings: Android 6.0 viz., Marshmallow (Android M) and later versions are similar to iOS, and allow users to grant permissions to each app at run-time. Note here that without loss of generality we focus our attention on Andriod M since it was the first to support revocable permissions. However, all of our work is applicable to newer versions of Android including Android Oreo (Android-O) and also relevant to iOS.

Although this is an improvement over previous Android versions, it still has some inherent limitations. First, apps not developed for Android M or smartphones using previous Android versions, will not support fine-grained permission management. Second, even with Android M, an app that seeks access to a device feature (e.g., camera), need not declare the permission if it uses intents for that action (intents allow late runtime binding between code in different apps [3]). Third, an app can access sensor data (e.g., accelerometer) without declaring any permission at all. Recent works such as [27] have shown that this could lead to privacy leakage for a user. Fourth, an app could use third party libraries (e.g., Ad libraries) which inherit the permissions granted to the app. The user does not know when (in what context or for enabling what functionalities) sensitive APIs or resources are accessed. As an example, the app Location Tracker [13] displays a user's location but at the same time uses an ad library that tracks the user's location. While in this example, the usage of location information is likely to be legitimate, it has been shown recently that there can be cases where a third party library can misuse the privilege and compromise user privacy [29]. Last but not least, developers will have to make significant, non-trivial changes to their application code in order to support the new permission mechanisms of Android M. They will also have to ensure backward compatibility to support users of previous versions of Android.

Our goal: Given these limitations of Android M, we seek to develop a developer and user friendly, permission management framework. The framework should allow users to dynamically manage the permissions granted to the apps that run on their smartphones, either on a case by case basis or based on the context (context, in brief, refers to a coarse grained, pre-defined functionality that uses the permission and is discussed later) of usage. In facilitating this, it should ensure that apps do not crash on the user's phone. The development of such a framework is not easy and has several associated challenges.

Challenges: It is unlikely that developers will make an attempt to facilitate fine-grained permission management if it requires them to significantly increase the number of lines of code they need to write. Thus, a key challenge we need to address is: "How can our framework enable developers to easily facilitate dynamic permission management?" To ensure widespread usage, the framework must be usable not only with Android M but also previous versions i.e., we ask: "How can we make the framework Android version agnostic?" With widespread usage in mind, we ask: "How can we ensure that a user does not need to root her phone or install a customized version of Android in order to utilize the permission management functions incorporated by a developer?" Sometimes,

a developer may not properly declare the permissions that are required (or not required). Thus, a challenge we need to address is "How can we enforce proper permissions when an app either uses intents, or when permissions that aren't required are not disabled?" The framework should contain safeguards to protect the user's interests in such cases.

Last, we ask "What can we do (and how can we do it) if a developer chooses not to use the framework to support dynamic fine-grained permission management ?" In this case, a user space solution (not requiring root privileges) does not suffice. Thus, we seek to provide fine grained permission management (similar to the developer assisted user space framework) if the user chooses to root her phone.

Contributions: Our main contribution in this paper is the design and implementation of *Figment*, a developer assisted framework for dynamic, fine-grained version-agnostic permission management framework that addresses the above challenges. *Figment* is developed as an Android library and has an associated annotation language that makes it easy for developers to use. *Figment* intelligently exploits the power of Aspect Oriented Programming (AOP) to weave in permission management code into an annotated application code at compilation time to achieve its goal. It allows both (i) the user to grant/revoke permissions in a fine-grained way on a case by case basis and, (ii) context-aware provisioning of permissions (permissions are granted only if certain conditions hold). To address cases where an app is developed without using *Figment*, we design and implement an optional fail safe mechanism for users with rooted phones.

In brief, our contributions in building *Figment* are:

- We design and develop a developer friendly (simple to use) annotation language. The annotations intelligently exploit the AOP paradigm to facilitate fine-grained, dynamic permission management (Section III).
- We design and develop an Android library that can transform existing applications developed for any version of Android into applications that supports fine-grained revocable permissions. These libraries are weaved into the application code at compilation time (Section IV).
- We enhance the fine-grained revocable permission approach (similar to what is available with Android M) by providing a simple but effective context aware mechanism supporting flexible permissions as well as by introducing and managing new permissions for sensors (e.g., accelerometer) that output sensitive data (Section III and IV).
- We develop a fail safe mechanism within *Figment* to safeguard a user's permission choices when apps use intents or when developers improperly declare permissions. The mechanism essentially checks and returns either a null pointer or fake information (as in [19]) to the app if it asks for permissions disallowed by the user, if doing so does not cause the app to crash. Else, it prompts the user indicating that it is a necessary permission (Section IV).
- We develop an optional fail safe mechanism for users with rooted phones that want to have a fine-grained permission management for applications that do not use the basic version of *Figment*. The idea in this fail safe mechanism is similar to what was included in the basic version of *Figment*, but the implementation is different (Section V).

We implement and evaluate both the basic version of *Figment* and its optional fail safe component for existing and new Android apps on different versions of Android. We show that Android M's runtime overhead is higher than that of *Figment* (by approximately 272%). Further, Android M's fine-grained permission mechanism, in our case studies, requires a developer to write on average 131% more lines of code compared to when she uses *Figment*'s annotation language. However, the *one time* compilation overhead of an app using *Figment* is relatively high compared to that of Android M (180% increase on average); we believe that this is a reasonable cost given *Figment*'s benefits.

II. Limitations of Android's Permission Model

This section provides background on the Android permission mechanism. We then discuss why these are insufficient in terms of protecting user privacy either in inadvertent benign cases or from malicious applications.

Android System Permissions: Android's security design is based on the principle that no application by default has any permission to perform an action that will adversely impact other applications, the OS or the user's data. Because of sandboxing, apps must explicitly share resources and data by declaring permissions they need (for additional capabilities beyond the ones that are offered by the sandbox mechanism). An app's permissions are declared in its Manifest file. There are certain permissions that are granted to the app by default; however, certain other permissions will need to be explicitly granted by the user. Prior to Android M, such permissions were granted by the user during the installation of the app; if the user wanted to revoke such permissions, she had no recourse but to uninstall the app (and reinstall if desired). In Android Marshmallow (similar to iOS), the permissions are granted during run-time. The user is prompted when an app first seeks to use a resource. If the user does not grant the requisite permission, the app can potentially still run, but with limited functionality. Android M allows the user to modify the permissions granted to an app via a system application.

Motivation for *Figment*: Android's permission management system however, has significant limitations (even with Android M). First, the new permission management mechanism is only available with Android M. Thus, to migrate her app to Android M, a developer must ensure that her app is backward compatible with the previous Android versions. This makes the code more complex. Moreover, the process of handling requests and responses for permissions in the new Android is more complicated. If the android version is previous to M, then the permissions were already granted during the installation. If the android version is M or beyond, then the developer should perform the following steps:

- Every time the app needs to perform an action that requires access to sensitive resources, the developer should insert specific instructions within the app's code to explicitly check if the permissions are granted even if the user had already granted that permission before. The ContextCompat.checkSelfPermission() method must be called to check if the app has a particular permission.
- If the app does not have a permission to complete a restricted action, the developer should insert specific code in order to ask the user to grant that kind of permission. This can be done by calling the Activity.requestPermissions(String[], int) method.

- In many circumstances, the developer might want to help the user understand why her application needs a permission. In order to do that, the developer should specifically write code to export this reasoning to the user.
- When an app requests a permission, the system presents a dialog box to the user and when the user responds the onRequestPermissionsResult(int, String[], int[]) method is called. The app should override this method to find out whether the permission was granted by the user or not.
- The developer should appropriately handle both positive or negative responses from the user.

First, it is evident that the above steps add complexity and significantly more code to an app. Second, Android M is context agnostic; its permission mechanism handles every request of permission the same way. As a consequence, the user's privacy can be compromised. For example, a user may be willing to share her location in a specific context (e.g. show a route on a map), but not be willing to share it under a different context (e.g., the same app uses a 3rd library that tracks the user's location for analytic purposes even if she is not seeking a route). Finally, Android M provides only an ability to revoke a subset of the available permissions. For example, it does not allow the revocation of a permission granted to access the Bluetooth interface; further, permissions are not necessary for the use of sensors like the accelerometer. As prior efforts such as [27] have shown, this can result in serious privacy leaks that could have been prevented if the user had the option to not grant (or revoke) such permissions to undesired (possibly malicious) apps.

Given these limitations of Android, we design and implement a flexible, context aware permission management framework, *Figment*, that works with all versions of Android. The basic framework is available as an Android library and can be used in current or future Android projects. To enable *Figment*, the developer has to annotate methods that require restricted actions; we develop a simple to use annotation language for this purpose. In subsequent sections, we describe the components of *Figment* in detail.

III. *Figment*: Libraries and Annotations

In this section, we begin with some background on AOP. We then discuss the design of our annotation language that developers can use to provision fine-grained permission management. Subsequently, we provide an overview of how AOP and the annotation language fit in within *Figment*.

Aspect oriented programming: Aspect Oriented Programming (*AOP*) [25] allows a developer to add executable blocks into the source code without changing it. While doing so, she can specify which part of code can be modified via what is called a *pointcut*. Pointcuts are expressions which specify where code, not central to the business logic, can be added (*code injection*) in a program. Using pointcuts ensures that code that is core to the functionality of the app is not cluttered.

It is common for apps to contain functionalities that span multiple layers. These functionalities typically support operations such as authentication, logging, instrumentation, and validation. In AOP, *cross-cutting concerns*, describe such functionalities (which affect the entire app). For example, a developer may want to add logging to the communication and the UI (user interface) layers of her app in order to see how much time the CPU is spending in each method. Although

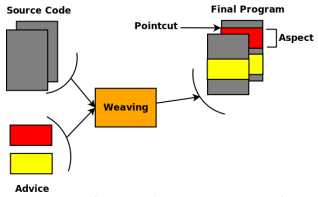


Fig. 1: Aspect Oriented Programming paradigm.

the purpose of the two layers differ, the code needed for performing the logging is identical and centralizing this code makes it easier to manage (change it) in the future.

Fig. 1 summarizes the AOP paradigm. The code that is injected (at a pointcut) is called *advice*. Typically, there are three kinds of advices, *before*, *after* and *around* which are executed before, after or instead of a method, respectively. Pointcuts can specify a single point in the code (e.g., code is inserted before the main method) or multiple points (e.g., code is inserted after the execution of any method inside a specific class). The process of injecting code is called *weaving* and the combination of an advice and pointcut is called *aspect*. With AOP new code can be injected either (i) at run-time, wherein the program has to explicitly ask for an enhanced code, or (ii) at load-time, wherein the modification is performed when the targeted classes are being loaded (e.g., Dalvik or ART), or (iii) at build-time, wherein the build process is modifying the code before packaging and deploying the app. In *Figment*, the code is injected during the build process.

Annotation Language for Permissions: In AOP, annotations are typically used as pointcuts. This makes it easy for developers to infer where the weaving process will inject code. Thus, with *Figment*, annotations are to be used by a developer for specifying which methods require what permissions.

Java Annotations: Java supports annotations [12], which provide information to the compiler to help detect errors or suppress warnings. They can also be used during compilation to help software tools generate code, or other files. They can also be made available to be examined at runtime. They start with the character "@" which tells the compiler that what follows is an annotation. They can include elements and can be applied to declarations of classes, fields, methods, etc.

Design of the Annotation Language: Android apps are developed in Java and thus, can be annotated. In Java, the functionality of an object is contained in its method. Thus, when a developer calls a restricted API, her code is written inside a method. Such methods can be annotated with *Figment*.

Figment provides two kinds of annotations for developers. First, methods requiring only one permission are annotated using the "@Permission" annotation. An annotation is described by (a) the type of permission, (b) a message that is displayed to the user if the permission is disabled and, (c) the context of the permission (discussed next), where a developer can specify under what contexts a method needs specific permissions. Contexts are labeled using an ID and annotations using the same ID are considered part of the same context. For example a method that requires the Location permission under the context with ID "MapLocator," is expressed as:

```
@Permission(requires=Location, message="Find Location",
        context="MapLocator")
public void findLocation () {
    // code
}
```

Second, if a method requires multiple permissions, *Figment* supports what is called the intersected method annotation ("@InterPermission") where the developer specifies an array of permissions, a message, and the context. For this method to be called, all the permissions should have been granted by the user. An example can be seen below:

```
@InterPermission( requires ={Location, Camera}, message="Picture With
      Location", context="CameraLocation")
public void takePictureWithLocation () {
   // code
}
```

What do we mean by context? In this paper, "context" refers to a specific functionality (for what purpose the permission is used). For example, "MapLocation" may be a context specification wherein the location is used to show the user's location on a map; in contrast "AdLocation" could be a context specification where the location is used by an ad library. Similarly, "CameraSelfie" may refer to a specification where a user is taking a selfie versus "CameraPoint" which may refer to when she uses it for a point and shoot. The developer can specify the context in his annotated code as discussed above. As discussed later, if a developer has not annotated his code, *Figment* allows the user to enforce a new context; the system simply polls the user for the permission and creates a context based on why the permission is being asked. It then saves the information for future use.

While finer-grained definitions of context are possible [21] (e.g, a permission may be granted only if the residual battery percentage is higher than a threshold), we defer such a possibility to the future. Note that since contexts are specified by a set of "method invocations" in *Figment*, they are finite and in almost all cases, are limited to a very small set of possibilities (as also pointed out in [23]).

Warning the developer of possible issues: *Figment* includes a static analysis tool for developers, based on lint [4]. It checks for potential problems during compilation between simple and annotated methods that are *dependent*. Specifically, it checks the consequences of disabling an annotated method (assuming a permission was denied) on the program execution. If the consequence is fatal (e.g. the app crashes), a warning is displayed to the developer. This helps developers refine their code; such issues can be avoided by ensuring that there are no dependencies between different simple and annotated methods.

Putting it all together: With our annotation language and AOP, *Figment* allows a developer to include revocable permissions in her app. If contexts are properly created and separated, disabling a context should allow the app to execute in other contexts (app should not crash). The user is informed of mandatory permissions needed by the app (in all contexts).

Figment is agnostic to the Android version. As part of *Figment* we package the annotation language and the AOP techniques for code injection (described in Section IV) in a library, which can be imported into an Android project and compiled with it. The code is injected during the build process. The process is shown in Fig. 2. Before the actual compilation, the annotated code and our libraries pass through the weaving process, during which code (for enabling proper permissions) is injected at appropriate places (where the code is annotated). Then, the modified code is compiled and transformed into the Dex format. Finally, the ApkBuilder constructs the final APK (Android application package) package.

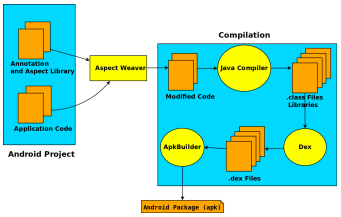


Fig. 2: Compilation Procedure.

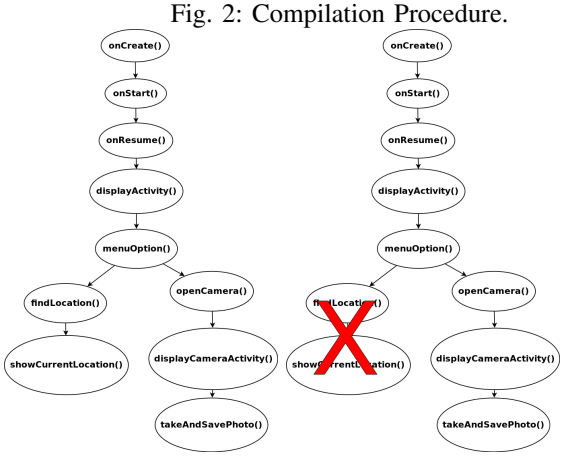


Fig. 3: Control Flow Graphs.

IV. *Figment*: Client Side Operations

In this section, we first describe how fine grained context aware permissions are enforced with *Figment*. Then we discuss how our system fits in within Android.

Fine grained permission management: Fine grained permission management in *Figment* comes into play when its code is injected into an annotated Android project. A program, during its execution, traverses many different paths that can be represented using a Control Flow Graph (CFG) [18]. The permission management mechanism determines the paths allowed (traversable) in the CFG during run-time based on user input. It thus, based on the permissions granted, creates a different functional version of the same program (with different capabilities). To illustrate, imagine that we have an Android app that displays the user's location on a map and at the same time, can take pictures. Its CFG can be seen in Fig. 3a. The nodes represent methods. The user by granting the app a permission for accessing her location but not for taking pictures, transforms the CFG during run-time into the one in Fig. 3b. As seen in the latter figure, everything below the getLocation node is truncated in the graph by *Figment*.

To support changing of an app's functionality during run-time, we design the *Figment*'s revocable permission mechanism. Its functions are described with the help of the example in Fig. 4. As the app is executed, at time t_1, it encounters an annotated method that needs a permission. Thus, the action (advice) taken by an aspect at that pointcut will be: **(1)** Read the central database. **(2)** If the app has not asked the relevant permissions yet (for this particular annotated method), then a message is displayed to the user. **(3)** If the permission is granted, then the permission together with the context and the app name, is saved to the database. Thus, the user is not asked again in the future, for that permission for this particular annotated method. Later, if the program execution encounters the same annotated method, the database will return a success and the method will be executed right away. **(4)** Finally, our

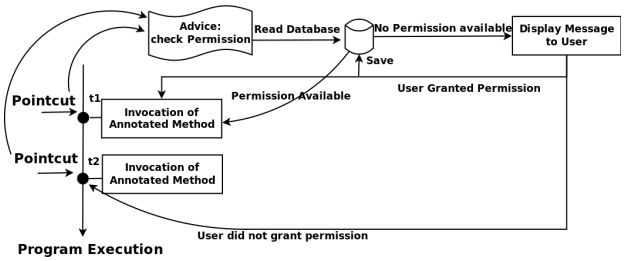


Fig. 4: *Figment*'s Revocable Permission Mechanism.

library performs the invocation of the annotated method.

At time t_2, the program encounters an annotated method that requires a permission that is different from the one at t_1. The steps performed by the encountered aspect are the following: **(1)** Read the central database. **(2)** If the application has not asked for the relevant permissions yet (for this particular annotated method), then a message is displayed to the user. **(3)** If the permission is denied, the *Figment* library will not invoke the annotated method and the execution of the program will continue after the pointcut.

The user can dynamically change her decisions with regards to granted/denied permissions using a front end that is part of *Figment*. It reads the database and displays the enabled/disabled permissions for each installed app. If the user decides to change her decision, she can do so by using the front end and the central database is updated accordingly.

Handling dependencies: Consider a case where a method (say A) calls an annotated method B. However, the permissions sought by B are not granted by the user. If the two methods are independent (meaning that the other parts of A can be executed without B), then A is safely executed without calling B. If on the other hand, A depends on a value returned by B (e.g., location), then *Figment* tries to return the safest value (which ensures that the app does not crash) as we discuss later. However, A may expect a value from B that has nothing to do with a permission (e.g., calculation); in such a case, *Figment* cannot return a safe value. It simply informs the user that A cannot be executed without the permissions needed by B. Note that as discussed earlier, *Figment* already warns the developers of such cases which can be avoided if there is a clean separation between methods (no dependencies).

How does *Figment* Fit in? We show how *Figment* fits in within the Android system in Fig. 5. When an app has to access a resource (e.g., Camera) or a system service (e.g., Location), the Android system checks if the app has been granted the required permission (Permission Check). If the permission has not been granted, the app will stop working. If Android M is used, *and if* the app supports revocable permissions, it can function without the functionalities that require the corresponding permission. As shown, with an annotated app using the *Figment* library, when an action requires a permission, an additional check is invoked by *Figment*'s library, prior to Android's permission check. Thus, revocable permissions are now possible with older Android versions.

Protection from missed annotations: A developer may compile her project with the *Figment* library but "not" annotate all the methods or intents. This may cause some unnecessary permissions to be granted, which in turn can lead to privacy leaks and/or a large attack surface. To protect users in such cases, *Figment* creates pointcuts for each API call that accesses any resources needing permissions. The only requirement here

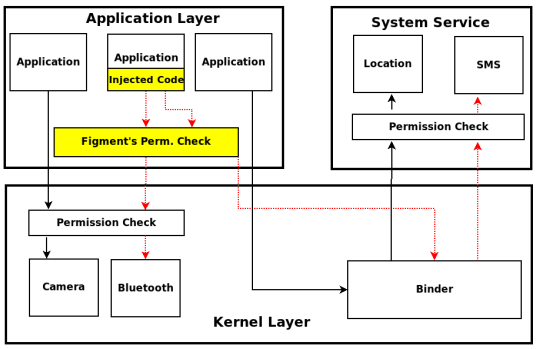


Fig. 5: Permission Check in Android and *Figment*. The shaded blocks represent *Figment*'s components.

is that the app has to be compiled with the *Figment* library.

To illustrate, consider a case where a developer (using *Figment*) provides a camera functionality to her app via a third party app by leveraging the intent mechanism in Android [5]. When the program execution reaches the camera intent, *Figment* will automatically check the database if the corresponding app, under that context, has the camera permission. If not, a message is displayed to the user, who can then choose to grant or not grant the permission. If the user denies the permission, then the intent is not called.

If the method is returning a value, then *Figment*'s protection mechanism will try to return the safest value based on the documentation for the Android API [2], that ensures that the app will not crash. For example, say the method is getLastKnowLocation which requires a permission for Location, and the developer has not performed a null check for the return value. If the method returns a null value the program will crash with a NullPointerException. *Figment* recognizes this by registering a global exception handler for each Android app to monitor the uncaught exceptions that result in a crash. It then returns either a random or a predefined value that ensures otherwise (again a proper value is chosen based on the Android API and the function). For example, an acceptable value for a location is one that provides a latitude between -90° to 90° and a longitude between -180° to 180°. Thus, *Figment* will return a random or a predefined location within that range. As another example, consider the International Mobile Equipment Identity (IMEI) number. IMEI is a 15-digit number assigned to all cellular devices. If the user denies access to her IMEI, *Figment* returns a random or a predefined 15-digit number.

V. *Figment*: Fail Safe Option

If a developer does not use the *Figment* library (e.g., for older apps), fine-grained permission management for that app is not possible. For such cases, *Figment* includes an optional fail-safe mechanism for users with rooted phones. Note that only root access is needed and there is no need to make kernel level modifications to the Android system.

Hooking Methods: One can modify the functionality of an Android app by obtaining its APK, decompiling it (assuming that the code is not obfuscated) and inserting new code in desired locations. However, one will have to recompile and sign the entire APK. Now, the APK will not have the same signature as the developer/company that originally developed it. Thus, the modified APK cannot be made available in Google's Play Store [10]. Every time a new version of that app is available, one will have to repeat the procedure in order to

bring about the desired functionality. However, Android allows rooted users to "hook" method calls, meaning that the user can inject her own code before and after method calls.

After the initialization of the kernel, the first process that runs is called "`init`," which initializes the elements of the Android system. `init` starts Zygote, which is a daemon for launching Android apps (i.e., it is the parent process of every Android app). Zygote preloads all necessary Java libraries and starts the "system server," which is responsible for initializing all system services. Zygote also opens a socket /dev/socket/zygote to listen for requests for starting apps.

When an Android app is to be launched, Zygote receives a request through this socket and triggers a fork() call to create a clone of itself. Because Android is based on the Linux Kernel, during the fork process no memory is actually copied. It is shared and marked as Copy-on-Write (COW). Thus, all apps use the exact same copy of libraries and resources. Because of this, with root access one can easily change the Java libraries and resources of the Zygote process making all the Android apps follow a different classpath. By introducing a library that wraps all the Android's API methods in the classpath, one can force the system to use the "hooked" methods instead of the original ones. When a caller calls an Android's API method, the hooked method is called first, followed by the calling of Android's API method. One can change the functionality of the app before or after the API method call or entirely ignore it (meaning that never execute it).

Fail-Safe Root Level Management: Hooking methods are exploited to enable users with rooted phones, enforce fine-grained permission management on apps not developed using *Figment*. The hooked method checks a database on the local disk (as before) and asks for needed permissions. If such a permission is granted, the database is updated with the app name, and the context (deciphered) for which the permission was granted. If the permission is denied, the hooked method tries to return the safest value that ensures that the app will not crash (as with the basic version discussed previously). Note that, the user can still change her decision at a later time by making changes to the database entries for an app.

VI. Evaluations

In this section, we first present the details of our implementation of both the core version of *Figment* as well as its optional fail safe component. Then, we evaluate *Figment* via case studies and comparisons with Android M.

A. Implementation

***Figment* Library:** We implement the *Figment* Library using the Android SDK (Android Software Development Kit) version 10. The annotation language is designed by leveraging the java.lang.annotation package [12]. Dynamic permission management is achieved using AOP [25] as described in Section III. We use the AspectJ [8] compiler during the build process of an Android app. It can be used on any phone that runs Gingerbread or a newer version (as of 2018 [16], the phones that run Gingerbread or newer versions of Android, account for 100% of the Android phones).

***Figment*'s Fail-Safe Mechanism:** The optional fail-safe mechanism of *Figment* requires a rooted phone. It has been developed as an extension to the Cydia Substrate Framework cydia-substrate (a popular code modification platform for iOS and Android). The extensions to the Cydia Substrate are regular classes that are loaded immediately after the Java VM is initialized. These classes have a static method named initialize and once loaded by the classloader, this method is executed allowing the user to run code that uses the public API provided by the Cydia Substrate. Once the fail-safe mechanism is installed, the Cydia Substrate framework automatically discovers it. Upon rebooting the phone, it is activated.

B. Android M vs Figment

First, we perform a case study with a custom app whose functions are common to existing popular messaging apps. The app consists of 5 fragments. Table I shows the permissions required by each fragment with (a) *Figment* and (b) Android M. The SMS fragment is used for sending SMS messages and requires the SEND_SMS permission. The READ_CONTACTS permission is needed by the Contacts fragment (to show the contacts). The Camera fragment requires the CAMERA and the WRITE_EXTERNAL_STORAGE permission (to save the picture to the local disk). Similarly, the Voice fragment needs the RECORD_AUDIO and WRITE_EXTERNAL_STORAGE permissions. The Location Fragment displays the user's location *and* tracks her movement using the accelerometer. It also uses a third party custom library that tracks the user's location periodically and sends it to a remote server. These functionalities in Android M require the ACCESS_FINE_LOCATION permission. The *Figment* library requires two permissions here, because the tracking of the user's location is performed under two different contexts; further, with *Figment*, also required is a permission for using the "ACCELEROMETER".

Listing 1: Code snippet using *Figment*.

```
@Override
public void onStart () {
 super.onStart();
 requestLocation();
 requestAccelerometer();
}

@Permission(requires = Permissions.LOCATION, message = "We require
      this permission to display your location.", context =
      "FragmentLocation")
private void requestLocation() {
 lMng = (LocationManager)
  getActivity().getSystemService(
        Context.LOCATION_SERVICE);
 if (lMng != null) {
  lMng.requestLocationUpdates(
    LocationManager.GPS_PROVIDER, 2000, 10, this);
 }
}

@Permission(requires = Permissions.ACCELEROMETER, message =
      "We require this permission to track your movement.", context =
      "FragmentLocation")
private void requestAccelerometer() {
 Sensor acc;
 sMng = (SensorManager)
  getActivity().getSystemService(Context.SENSOR_SERVICE);
 if (sMng != null) {
  acc = sMng.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  sMng.registerListener(this, acc,
        SensorManager.SENSOR_DELAY_NORMAL);
 }
}
```

Listings 1 and 2 show snippets of code for accessing the location of the user using *Figment* and the API of Android M, respectively. Upon the creation of the Location Fragment, it returns the view hierarchy associated with it by invoking the onCreateView method. The onStart method is called to make the fragment visible to the user. In our example, *Figment*

Fragments	Figment	Android M
SMS	SEND_SMS	SEND_SMS
Contacts	READ_CONTACTS	READ_CONTACTS
Camera	CAMERA, WRITE_EXTERNAL	CAMERA, WRITE_EXTERNAL_STORAGE
Voice	RECORD_AUDIO, WRITE_EXTERNAL	RECORD_AUDIO, WRITE_EXTERNAL_STORAGE
Location	LOCATION, ACCELEROMETER	ACCESS_FINE_LOCATION
3rd Library	LOCATION	ACCESS_FINE_LOCATION

TABLE I: Permissions required for each fragment with *Figment* and Android M.

registers two listeners, one for the location and the other for the accelerometer, respectively (lines 14 and 26 in Listing 1). A developer using Android M will have to check each time if the permission is granted and if it is not, she will have to request it. For each permission therefore, she has to write around 10 lines of code (e.g., lines 8 to 29 in Listing 2 for the Location permission). If the developer uses the *Figment* library, she only needs to annotate the requestLocation and requestAccelerometer methods and our library will handle everything automatically (lines 8 and 19 in Listing 1) , which entails one line of code per method.

Listing 2: Code snippet using Android Marshmallow API.

```
1  private static final int REQUEST_LOC = 21;
2
3  @Override
4  public void onStart () {
5   super.onStart();
6   requestAccelerometer();
7
8   if(ContextCompat.checkSelfPermission(
9          this.getActivity(),
                 Manifest.permission.ACCESS_FINE_LOCATION) ==
                 PackageManager.PERMISSION_GRANTED) {
10   requestLocation();
11  } else {
12   if(ActivityCompat.shouldShowRequestPermissionRationale(
13               this.getActivity(),
14              Manifest.permission.ACCESS_FINE_LOCATION)) {
15    ShowDialog dialog = new ShowDialog("Location Permission", "We
           require this permission to display your location.");
16    dialog.execute();
17   } else {
18    ActivityCompat.requestPermission(
19             this.getActivity(),
20             new String[]{
21             Manifest.permission.ACCESS_FINE_LOCATION},
22             LOCATION_REQUEST);
23   }
24  }
25 }
26
27 public void onRequestPermissionResult(int reqCode, String
       permissions[], int[] results) {
28  if (reqCode == LOCATION_REQUEST) {
29   if (results.length > 0 && results[0] ==
          PackageManager.PERMISSION_GRANTED) {
30    requestLocation();
31   }
32  }
33 }
34
35 private void requestLocation() {
36  lMng = (LocationManager)
37   getActivity().getSystemService(
          Context.LOCATION_SERVICE);
39  if (lMng != null) {
40   lMng.requestLocationUpdates(
41     LocationManager.GPS_PROVIDER, 2000, 10, this);
42  }
43 }
44
45 private void requestAccelerometer() {
46  Sensor acc;
47  sMng = (SensorManager)
48   getActivity().getSystemService(Context.SENSOR_SERVICE);
49  if (sMng != null) {
50   acc = sMng.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
51   sMng.registerListener(this, acc,
          SensorManager.SENSOR_DELAY_NORMAL);
52  }
53 }
```

Next, we take two existing open source apps and modify them to support revocable permissions. We make two constructs; in the one the API of Android M is used, and in the other, *Figment* is used. The first app is called Ringdroid [15] and the second is SoundRecorder [17]. The first requires five, what we consider sensitive permissions (Read/Write Contacts, Writing to the local disk, recording audio and writing system settings) while the second requires only 2 such permissions (writing to the local disk and recording audio).

Overhead in terms of lines of code: Our library in general requires one line of code (LOC) for initialization and one line of code for each method that requires such a permission. In contrast, depending on the situation, Android M requires 10 or more lines of code to achieve the same functionality that *Figment* achieves with one line of code (the developer follows the procedure described in Section II with Android M; multiple lines of code are needed to check for permissions each time, prompting the user for the permission, etc.). We count the lines using the cloc [9] program. For the custom app, our library requires only 8 more lines of code while Android Marshmallow requires 62 (i.e., a 675% difference).

With Ringdroid, a modified Android M version requires 59 more lines of code. However, with *Figment*, it requires only 5 lines of code. The difference is approximately 169%. In contrast, with two permissions, SoundRecorder needed 53 and 25 more lines of code respectively (approximately a 72% difference). The higher number of lines in code with SoundRecorder is because we had to make small changes in the code in order to support modular annotations. Note here that for much more complex applications with several permissions, sought in different contexts, using *Figment* and our annotation language can be expected to significantly reduce the number of lines of codes needed for fine-grained permission management in the absolute.

Compilation Overhead: Our case study was performed on Android Studio 1.4.1, running on a machine with a quad core Intel Core i7 2.00GHz CPU with 8GB of RAM and a hard drive of 1TB at 5400 rpm. Due to the weaving process and the size of the *Figment* library, the compilation process with *Figment* is longer than with Android M's API. To quantify this overhead, we performed a clean before each compilation in order to achieve the maximum overhead that can be incurred. With 10 runs, both the average results and the 95% confidence intervals are shown in Figure 6a. On average, the custom app developed using *Figment* compiled after 78 seconds while with Android M it took 30 seconds (a decrease of 61.54% is with Android M). Ringdroid with Android M compiles in 22.43 seconds while the SoundRecorder does so in 28.03 seconds, on average. Using *Figment* the times are 67.53 and 73.33 seconds,

respectively. This corresponds to an increase of approximately 201% and 161%. Although this overhead is significant, it is incurred only once, when the app is compiled for the first time.

Runtime Overhead: Next, we compare the overheads to check and ask for a permission, with *Figment* and Android M. The experiments below were run on a Nexus 5 phone using the custom app. We use a different Android version (lollipop) with *Figment* (not Android M) so as to make a clean comparison of these runtime checks. The results are in Figure 7. The results from 100 runs in terms of the 95% confidence intervals and the averages are in Fig. 7. We see that the average delay with *Figment* is around 0.55 ms while it is 2.05 ms with Android M (*Figment* is 4 times faster). We find that the results are very similar with RingDroid and SoundRecorder since the permission management functions are similar. Thus, we do not present them due to space constraints.

There are two reasons why the runtimes are much lower with *Figment*. First, during runtime, the access control mechanism of Android M has to check whether this app has been granted each specified permission. Furthermore, when the app requests a permission, Android M creates a background thread in order to perform that check. With *Figment* on the other hand, the access control checks are performed during the installation of the app. *Figment* in essence provides an lightweight overlay mechanism (as described earlier) on top of the what Android provides. Further, *Figment* checks the permissions using the same thread and thus, the lookup is very fast (no time is taken for creating a background thread.)

C. Figment's Context Awareness

We next illustrate how *Figment* supports context aware permissions. We develop a simple Android app with which a user can press a button and display her last known location. The app also uses a library that tracks the user's location and using a background thread it sends this information to a server. The findLastLocation method finds and displays the last known location of the phone, while the startTracking method tracks the user's location and sends this information to a remote server. Since the app's code accesses the location in two places in the code, the developer should annotate the startTracking and findLastLocation methods to facilitate context aware permissions. *Figment* asks the user for permissions for both methods explicitly since the contexts are different. The user can choose "not" to grant the permission in one of the contexts and thus has a finer grained control of how the app operates. With Android M, the developer *cannot* provide context awareness in her app. When the app is launched, a dialogue is displayed to the user asking her to grant a permission for accessing her location. In case she grants it, the app will be able to get and display her last known location; however, the key issue is that the background thread will implicitly inherit the permission to track her location (thereby violating privacy). We have verified that this is the case.

D. Figment's Protection Mechanism

Next, we seek to show how *Figment* prevents intents from accessing resources without an explicit permission from the user. Towards this, we develop an app that takes a picture and transmits it to a remote server. The app leverages Android's intent mechanism of taking pictures; it sends an intent to a different third party app (which takes the picture on its behalf). This functionality of the app is shown as a snippet in Listing 3.

Listing 3: Camera Intent

```
1 private void takePictureIntent () {
2  Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
3  if ( intent . resolveActivity (getPackageManager()) != null ) {
4    startActivityForResult ( intent , REQUEST_IMAGE);
5  }
6 }
```

In order to get the picture from the other app using inter process communication (IPC), our app overrides the following method (Listing 4).

Listing 4: Getting picture from other app

```
1 @Override
2 protected void onActivityResult ( int requestCode, int resultCode ,
        Intent data ) {
3  if (requestCode == REQUEST_IMAGE && resultCode ==
        RESULT_OK) {
4   Bundle extras = data . getExtras () ;
5   Bitmap bitmap = (Bitmap) extras . get ("data") ;
6   sendImageToRemoteServer(bitmap);
7  }
8 }
```

The app's code is compiled with *Figment*. There is no annotation with the takePictureIntent method (Listing 3). However, *Figment* determines that the app seeks to use the camera and prompts the user for an explicit permission. This is because line 4 in Listing 3 is in fact *a pointcut* (see Section III). Thus, *Figment* extracts the Intent object that was in the startActivityForResult method. It finds out (by calling the getAction method of the Intent's class) that it is an intent for taking pictures. Then the procedure follows what was described in Section IV. Note here that Android M's permission mechanism by itself, does not ask for any permission to perform this action. We also point out that this feature of *Figment* is compatible with Android M.

E. Figment's Optional Fail Safe Mechanism

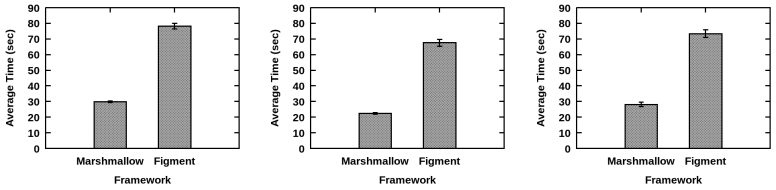
In this subsection we describe how *Figment*'s fail safe mechanism protects the user with a case study. Specifically, we download an existing app (the flashlight app) from Google Play Store [10] and perform this study on a rooted Moto X running Android Ice Cream Sandwich.

With *Figment*, the user can open the permission manager and see what permissions this flashlight app requires. As shown in Fig. 8a, Flashlight requires permissions for accessing the location of the user, reading and writing data to the disk, accessing the state of the phone and accessing the camera. Most of these permissions do not affect the core functionality of the app! Let us assume that the user decides that the app should have access only to her camera (and should not be granted the other permissions). *Figment*'s optional fail safe framework allows her to do so. Using the dialogue presented using the phone's front end (see Fig. 8b) she can disable the undesired permissions. We have verified that the flashlight app continues to function without any of these permissions.

VII. Related Work

In this section we discuss relevant related work.

Privacy and Permissions: To protect privacy, AppFence [24] and MockDroid [19], return fake data in lieu of real data to specified apps. However, these systems need a modified version of Android. Moreover, they do not provide any context aware mechanisms for fine-grained permission management.



(a) Custom App. (b) Ringdroid. (c) SoundRecorder.

Fig. 6: Average Compilation Time.

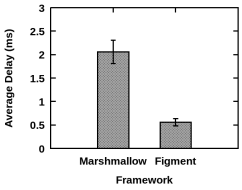
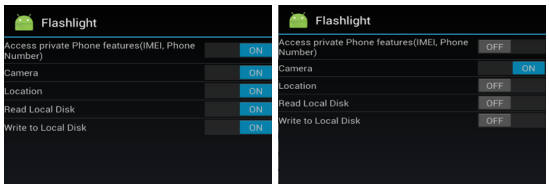


Fig. 7: Average Runtime Overhead for Custom App.



(a) Enabled Permissions. (b) Disabled Permissions.

Fig. 8: Flashlight Permissions.

Context Awareness: ipShield [20] and CSAC [28] provide context aware mechanisms for access control in Android. However, both approaches require changes to the Android operating system; different changes are needed for different versions of Android (with different kernels). Furthermore, no tools such as *Figment* are provided to build apps that support fine grained permission management. *Figment* can work with any Android version and provides user-space based, developer assisted permission management. Unlike in these approaches, thus, new contexts can easily be introduced within *Figment*.

Annotation and AOP: Google [11] has developed a set of annotations for permissions in Android but it is used only in code inspection tools such as *lint*. APE [26] uses an annotation language that eases the development of energy-efficient Android apps. In Java, AOP is used by frameworks like Spring [7] to provide declarative enterprise services and to allow users to implement their own custom aspects. WeaveDroid [22] allows a user to use AOP on Android by taking any existing Android app adding an aspect as input, and weaving them together; the goal however is not fine-grained permission management.

VIII. Conclusions

In this paper, we design and implement a framework, *Figment*, that facilitates fine-grained permission management for mobile apps. *Figment* contains a set of libraries which can be readily called by developers using its annotation language, to allow users to grant and revoke permissions on a case-by-case basis. *Figment* works with any version of Android, and does not need any changes to the Android system. We show via case studies that *Figment* reduces the code complexity compared to Android M in addition to providing context aware operations. It also reduces the runtime overhead compared to Android M, but experiences an increase in the one time compilation overhead.

IX. Acknowledgments

This research was partially sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work was also partially supported by NSF Award 1617481. The authors would like to thank the anonymous reviewers for their constructive feedback.

References

[1] Aggressive advertisers pose privacy risks. http://goo.gl/cl8HuK.
[2] Android api. http://goo.gl/kOQqRE.
[3] Android: Intents and intent filters. http://goo.gl/zr6vBY.
[4] Android: lint. http://goo.gl/OTshmb.
[5] Android: Taking photos simply. http://goo.gl/0ZQrf6.
[6] Apps permissions in the google play store. http://goo.gl/ph7KGk.
[7] Aspect oriented programming with spring. http://goo.gl/1UnkGS.
[8] Aspectj. https://goo.gl/LHLhDv.
[9] Cloc. https://goo.gl/PsfjQP.
[10] Google play store. https://goo.gl/kN0Nhz.
[11] Improving code inspection with annotations. http://goo.gl/qSE9dh.
[12] Java se annotations. http://goo.gl/g9b0Dh.
[13] Location tracker. https://goo.gl/X2LuAd.
[14] Mobile operating system wars - Android vs. ios. http://goo.gl/V0wbT5.
[15] Ringdroid. https://goo.gl/MhLqGW.
[16] Share of android platforms on mobile devices with android os. http://goo.gl/2WaEEI.
[17] Soundrecorder. https://goo.gl/apNf3X.
[18] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.
[19] 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*, pages 49–54. ACM, 2011.
[20] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. B. Srivastava. ipshield: A framework for enforcing context-aware privacy. In *NSDI*, pages 143–156, 2014.
[21] S. Elmalaki, L. Wanner, and M. Srivastava. Caredroid: Adaptation framework for android context-aware applications. In *MobiCom*, 2015.
[22] Y. Falcone and S. Currea. Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 350–353. ACM, 2012.
[23] I. Gasparis, A. Aqil, Z. Qian, C. Song, S. V. Krishnamurthy, R. Gupta, and E. Colbert. Droid m+: Developer support for imbibing android's new permission model. In *ACM AsiaCCS*, 2018.
[24] 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*, pages 639–652. ACM, 2011.
[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.
[26] N. Nikzad, O. Chipara, and W. G. Griswold. Ape: an annotation language and middleware for energy-efficient mobile application development. In *ACM ICSE*, 2014.
[27] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: password inference using accelerometers on smartphones. In *ACM Workshop on Mobile Computing Systems & Applications*, 2012.
[28] A. Rahmati and H. V. Madhyastha. Context-specific access control: Conforming permissions with user expectations. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.
[29] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. Citeseer, 2012.