Mediating Applications within the Android System

Usman Darwesh
usman.darwesh@ucalgary.ca

Raymond Duong
raymond.duong@ucalgary.ca

Nizar Maan
maann@ucalgary.ca

Syed Ahmed Zaidi
syed.zaidi@ucalgary.ca

ABSTRACT

Android smartphones are becoming an essential part of everyday life. For the sake of convenience, storing sensitive information on these devices has become common amongst users. Applications developed for users often require the use of said sensitive information. Consequently, Android applications are a potential attack vector for data extraction. There are various applications, called "AppLockers", which claim to provide an access control mechanism to one's sensitive data.

This research paper explores the flaws found in currently existing AppLockers. It will then present an alternative solution which will involve the modification of the Android operating system, and an improved implementation of currently existing solutions.

Keywords

Android, AppLocker, security, threat, kernel, permissions, IPC, binder, intent

1. INTRODUCTION

The emergence of smartphones has revolutionized personal computing for people all around the world. These devices are no longer tools for communication, but have become an integral part every day life. From paying bills to indulging in entertainment, tasks which were once exclusive to personal computers, are now available in the palm of your hands. Much of this has been made possible by the Android framework. The most common operating system (OS) for smartphones, as shown in Figure 1, is the Android OS[1] which was developed by Google. Android OS supports a variety of applications which are widely accessible through the Google Play Store. These applications (apps) are developed by third parties and often forced to uphold specific policies regulated by Google. Some of these apps require access to restricted features (e.g. Camera, Contacts, Photos) in order to function properly. These features are protected by "*permissions*", which mediate access to them. For example, the Snapchat app requires the use of the CAMERA permission in order to capture pictures and videos.

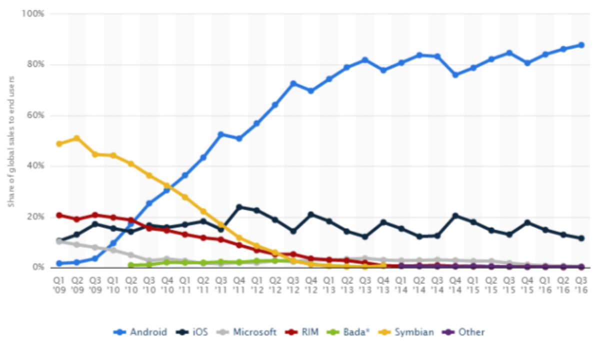


Figure 1: Global OS Market Share[14]

An app must have its requested permission declared in its Manifest file. Upon installation, an app can only be granted its declared permissions. Granted permissions must adhere to Google's outlined policies. While some granted permissions have legitimate usages, other apps request permissions which are otherwise irrelevant to their apparent functionality. For example, *Super-Bright LED Flashlight* is a third party flashlight application which should essentially make use of the CAMERA permission in order to use its flash to provide light (Ref). However, in addition to the CAMERA permission, *Super-Bright LED Flashlight* also requests READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions to modify or delete the contents of USB storage[5].

Since some applications are accessing a device's storage, it becomes apparent that sensitive data could be exploited which results in a breach of the user's privacy. Methods by which users can preserve their privacy are as follows:

- Be wary of the requested permissions prior to installing an app
- Utilize an access control mechanism to protect resources

Many users may find the first to be rather inconvenient in that app installation should be a rather seamless process. The second, however, can be accomplished in one of two ways: either through some application (called an AppLocker), or through a modification to the Android operating system. The AppLocker approach bears many problems assuming the user's phone isn't rooted, and even then, some

.

AppLockers don't necessarily stop other applications from accessing sensitive data. The modification approach resolves many of the issues imposed from using an application, however such an implementation is difficult but certainly achievable.

2. THE ANDROID ARCHITECTURE

Android is an open source project primarily based on a variant of the Linux kernel. Since it is an open source project, there are multiple different versions of this OS customized by third parties running alongside the official version which is updated and maintained by Google. Despite its different versions, the underlying mechanism for the OS remains the same, namely its use of a Software Stack which holds four layers; the Linux Kernel, Libraries, Application Frameworks, and Applications, as seen in Figure 2. At the bottom of the Android Software Stack lies the Linux Kernel, which makes it possible for a level of abstraction between a device's hardware and higher levels of the software stack to exist. On top of this, the Kernel provides multi-threading capabilities as well as support for several drivers, such as camera, sound, video, and bluetooth drivers.

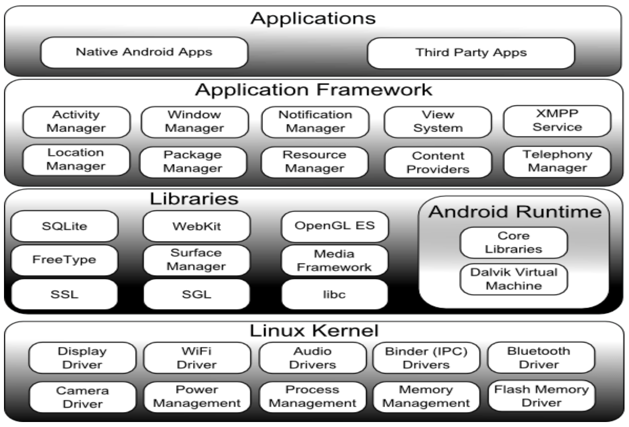


Figure 2: Android's Software Stack

The Android Application Framework allows access to the Application Program Interface (API), which simplifies usage of its components. It is a set of services that make up the environment where applications are run. This framework is implemented so that its components are reusable, interchangeable, and replaceable. The Content Provider, along with the Android.OS library, is a key service provided by the framework that makes Inter-Process Communication (IPC) possible, which will be explained more in depth in a later section. Applications, written in Java, sit at the top level of the software stack, they consist of activities, services, broadcast receivers, and content providers and are usually third party applications installed by a user.

A level above the Linux Kernel there is the Library, which is divided into two parts; the Android Library and Android Runtime. The Android Library is written in C++ and links the application framework to the kernel.[7][16][3][20]

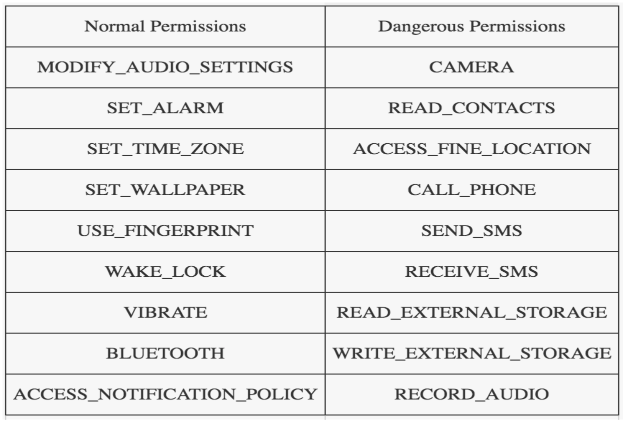
2.1 Android Security Model

The Android Runtime is further divided into two parts: the Java Core Library, and the Dalvik Virtual Machine (DVM), which provides the runtime environment for each application. Despite the natural assumption, applications do not run directly on the Kernel. Applications actually run on individual instances of the DVM. Thus applications are sandboxed so they cannot interfere intentionally or with the OS and other applications, nor can they directly access the hardware. When an app is installed on the platform, by default the system assigns the app a unique Linux user ID which is known only to the system. Each app's essential information is contained in a file called the *Manifest file*. Initially, the system sets the information in the manifest file to a certain user ID which is then associated with the app. Only the user ID assigned to that app can access the information. One example of such information would be the permissions required by the app. Because each app operates independently within the scope of a sandbox, if an app wants to use a resource or data outside of its sandbox then it must have the authorization to do so. Sensitive resources are protected through the use of permissions. An app that is requesting to access a critical resource must have the permissions listed in its manifest file. Authorized permissions are used by the application's sandbox to mediate the use of critical resources.[7][8][12][21]

Depending on the type of permission an app requests, the system may grant the permission automatically, or ask the user to grant the permission. Within the Android framework, there are two important categories[11] of permissions we should familiarize ourselves with:

- Normal permissions
- Dangerous permissions

The *Normal* permission category involves low risk permissions. These are used when the app requires data or resources that do not pose a risk to the user's privacy. On the other hand, the *Dangerous* permissions involve the use of sensitive data that can potentially be exploited (hence the term dangerous). If an app requests a *normal* permission, then the system automatically grants the permission. However, for a *dangerous* permission, the user has to explicitly grant the permission as we discussed earlier. Figure 3 depicts some examples of *normal* and *dangerous* permissions.[22][12][15]



Normal Permissions	Dangerous Permissions
MODIFY_AUDIO_SETTINGS	CAMERA
SET_ALARM	READ_CONTACTS
SET_TIME_ZONE	ACCESS_FINE_LOCATION
SET_WALLPAPER	CALL_PHONE
USE_FINGERPRINT	SEND_SMS
WAKE_LOCK	RECEIVE_SMS
VIBRATE	READ_EXTERNAL_STORAGE
BLUETOOTH	WRITE_EXTERNAL_STORAGE
ACCESS_NOTIFICATION_POLICY	RECORD_AUDIO

Figure 3: Normal and Dangerous Permissions

Currently, Android's security model employs the *principle of least privilege*, which means that every app operates using the least set of privileges necessary to complete its job. This is at the heart of Android's security model because an app cannot access a resource for which it does not have permission. However, once a permission has been granted, an app can access a given resources with the specific permission. Apps can communicate the resources among themselves via different types of IPC such as Binders, and Intents. We will now discuss about IPC, Binders, and Intents.

2.2 IPC, Binder, and Intents

The Binder structure is one of the most prominent subsystems built inside the Android platform and its core is very influenced by OpenBinder (Linux structure for IPC). The main purpose of the Binder is to provide a communication channel between processes, apps, and/or the system. A major feature the Binder possesses is the ability to provide "bindings" to data and functions in the same or different point of execution (sandboxes as referred to section 2.1). The Binder's core is comprised of the following:

- *Binder Object*: an instance of any class in which the Binder interface is implemented. A Binder object can be implemented by multiple Binders
- *Binder Protocol*: a very low level protocol which allows for communication with the hardware
- *IBinder Interface*: subclasses of sets of methods, properties, and events in which a Binder can use
- *Binder Token*: token/value which can allow each Binder to be uniquely identified

The Binder framework uses a form of communication primarily based on a client server structure. This form of communication involves an idle server waiting for incoming requests from a server. Once the request is received from the client, the server will send back a response through a thread from a thread pool. With this thread pool, it allows for multiple concurrent Binder objects to be handled.

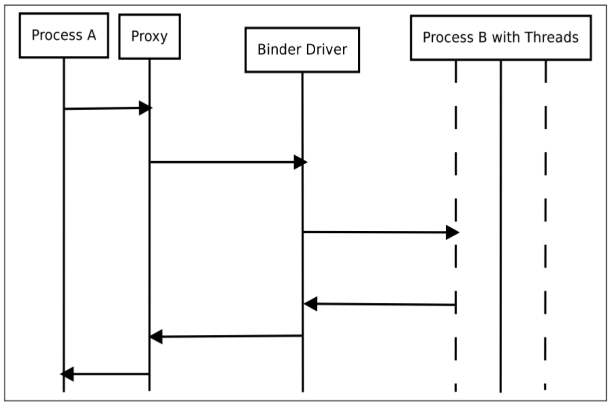


Figure 4: Binder Communication

The Binder's functionality is largely based upon transactions, a request and reply structure. These transactions are based upon process sending data to other process. Transactions are done in a way where the data is combined with the transmission payload data which includes:

- *Target*: destination Binder node
- *Cookie*: stores internal information
- *Sender ID*: contains information about the sender relevant to the transaction
- *Data*: contains a serialized data array of commands and arguments which can be processed by the Binder.

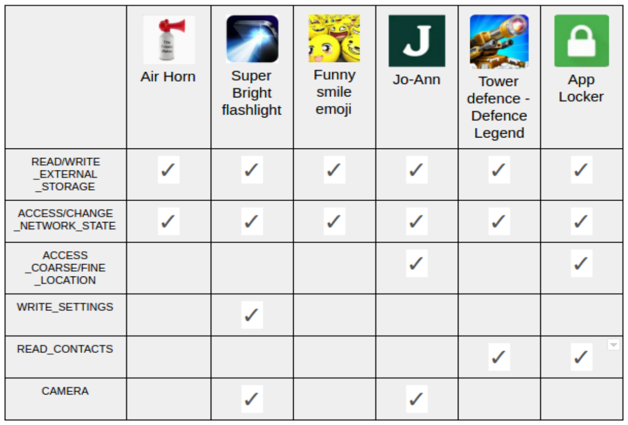
As we can see in Figure 4, Process A sends a transaction through a client-side proxy (this is what the Binder framework uses for communication) which then reaches the Binder driver to be processed. Once processed, the transaction will be sent to Process B which has multiple Binder threads running which will then deliver the message to the intended object. Developers are able to take advantage of the Binder's IPC through the use of *Intents*. An intent is a message that is sent with the Binder IPC. It is a representation of what an application hopes to achieve. It has fields such as an *action* and *data*, where an action is generic action to perform, and data is on which the action should be acted upon. Once an intent message is initialized, it will be sent via the intent reference monitor to the Binder to with the corresponding action request to be processed like how transactions are processed.[10][3][16][6][9]

3. MOTIVATION

Prior to considering that Android's security model doesn't provide adequate privacy measures, it is worth mentioning that the easiest way for an adversary to exploit one's sensitive data is by simply stealing the device that holds said sensitive data. Of course, Google has implemented countermeasures for such a situation. The most basic one being the "Swipe" option, which is not secure at all. Other well-known authentication methods that Google provides are; Passwords, PINs, and Patterns. But how secure are these methods? Android has a considerably "strong" policy to encourage users to employ strong password, PINs, and Patterns. For a password and PIN styled lock-screen, the minimum password length is 4, where as the maximum is 16. Whereas, Patterns have a minimum of 4 nodes used, and a maximum of 9. However, with Graphics Processing Units (GPUs) becoming faster and more reasonably priced, these methods can be cracked in a matter of days. Some commercial products claim to have the ability to test up to 112,000 passwords per second using a high quality GPU. For example: a 6 letter, lower case password can be guessed in one day.[4] Similarly, a pattern styled lock screen with 9 distinct nodes yields 389,112 possible patterns, which can be cracked in approximately 4 days. Consequently, using strong passwords can reduce the risk of a security breach but these mechanism are not an adequate mechanism for security controls.[13][19][17]

3.1 Threats on User Privacy

With 2.4 million apps - and counting - on the play store, a portion of these apps have malicious intent to invade user privacy. As mentioned previously, an app must have its requested permissions declared in the Manifest file. With the given manifest file, an app obtains the permissions once the user authorizes it to do so during the installation process. Users can either accept the permission requests to install an app, or not install the app. Moreover in the heat of the moment, users tend not to read the permission requests and trade their privacy for the riches. This is a concerning matter, as studies have shown that 70% of free apps request data or resources that involve user's private information.[18] For example, Consider Figure 5. The apps listed request permissions which have nothing to do with its functionality.



	Air Horn	Super Bright flashlight	Funny smile emoji	Jo-Ann	Tower defence - Defence Legend	App Locker
READ/WRITE _EXTERNAL _STORAGE	✓	✓	✓	✓	✓	✓
ACCESS/CHANGE _NETWORK_STATE	✓	✓	✓	✓	✓	✓
ACCESS _COARSE/FINE _LOCATION				✓		✓
WRITE_SETTINGS		✓				
READ_CONTACTS					✓	✓
CAMERA		✓		✓		

Figure 5: Over-Privileged Applications

This is a potential security and privacy concern because the user may not necessarily know the reason why some apps are requesting resources which do not have anything to do to the app's apparent functionality. To answer such apps' deceptive behaviour, scientists from the Computer Laboratories at Cambridge University conducted research and found that 80% of free apps on the Play Store are "supported by targeted advertisements".[18] These free apps request permissions irrelevant to their intended use in order to sell the collected data to third parties.

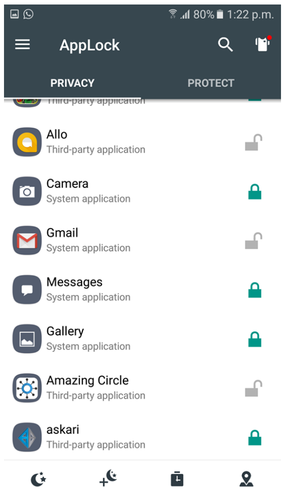
It is only natural for one to protect their personal and private data, therefore a security control mechanism is needed. Currently, Android does not provide such implementation and relies purely on its permission mechanism to provide security. However, applications have found a way to take advantage of the current implementation and invade the user's privacy. To protect against apps with malicious intent, it is recommended that one carefully reads the permission requests during the installation process, yet many users believe that installing an app should not be a time consuming process. Therefore, an access control mechanism is essential to protect one's resources. There are access control applications that supposedly provide an additional security control mechanism for the user, and are available for free on the Play Store. These apps are commonly referred to as "AppLockers".

3.2 Current Solutions

AppLockers provide a wide variety of features that aid a user's privacy, this includes the ability to restrict access to apps without authorization from the user (through a user-defined password or PIN), the ability to set different locks for each individual application, and it makes use of the front camera by taking a picture of an invader taking too many attempts at unlocking the app that is set to be locked.

Generally upon first launch, an AppLocker would prompt the user to select an unlocking mechanism such as password, PIN, or pattern. Some go as far as to provide support for biometric authentication (like a fingerprint). Once the user has set their security questions that would be used in the case that they forget passcode, the initialization of the AppLocker is complete.

Using an AppLocker is rather straightforward. As shown in Figure 6, one simply selects the individual apps they would like to have "locked".



80% 1:22 p.m.

AppLock

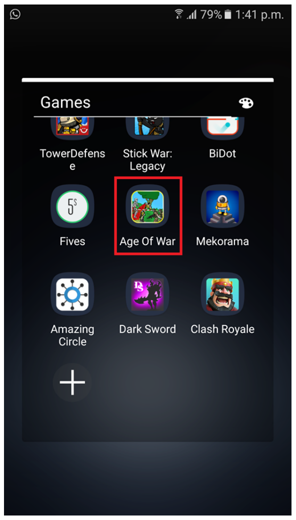
PRIVACY PROTECT

Third-party application

Allo	Third-party application	unlocked
Camera	System application	locked
Gmail	System application	unlocked
Messages	System application	locked
Gallery	System application	locked
Amazing Circle	Third-party application	unlocked
askari	Third-party application	locked

Figure 6: "AppLock" V2.21.1 - Select Apps to Lock

A locked app's behaviour is outlined in Figure 7 and 8. When one wishes to open the locked app, they are prompted to provide authorization to the previously set key.



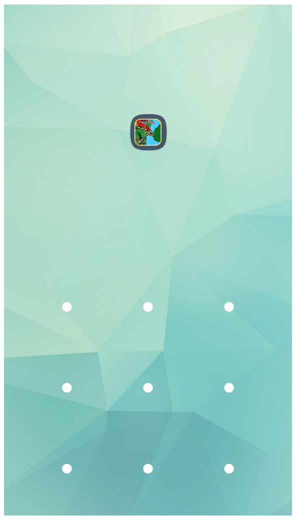
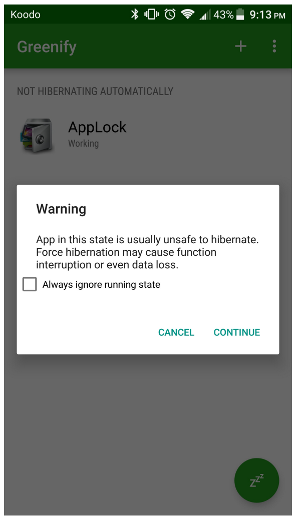


Figure 7: A Locked App **Figure 8:** App Behaviour

Despite its functionality, there lies a problem with most Applockers, this problem being that they are susceptible to attacks specifically against the AppLocker. If the AppLocker does not provide the ability to lock the PackageInstaller activity (which is responsible for installing/uninstalling apps), then a malicious user can simply uninstall the AppLocker. Similarly, if the settings aren't locked, than the service of the AppLocker can be stopped. During our experimentation against multiple Applockers available on the Google Play Store (AppLock, Hexlock App Lock & Photo Vault, CM Security AppLock, and Smart AppLock), we uncovered a major attack vector. Consider how these AppLockers work, they have their services constantly running in the background. An attacker can then do the following: flash an app capable of stopping services, then simply stop the AppLocker service, then uninstall the app. For example, through the use of the app "Greenify" as shown in Figure 9 and 10, one can simply cause the AppLocker to "hibernate" which essentially stops the service for a period of time until the service is reinitialized. To further abstract the idea, a user with malicious intent who has access to one's phone may create an app capable of disabling services. They can then either use a computer to install the application via pressing the install button prior to the AppLocker's overlay appearing, or by using Android Studio to install the app directly. This is a very serious threat against most if not all AppLockers available on the Play Store. The former means of installing a disabler can be mitigated through the use of a faster overlay, however the latter cannot.



Warning

App in this state is usually unsafe to hibernate. Force hibernation may cause function interruption or even data loss.

Always ignore running state

Figure 9: Kill AppLock Service

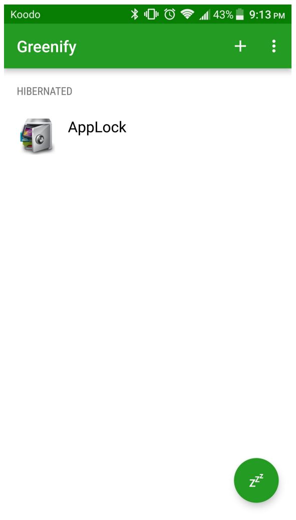


Figure 10: AppLock Service Killed

Assuming the aforementioned threats have been dealt with, an attacker is still capable of accessing locked applications through applications that are not locked. An example of this is having the Camera app locked, and still being able to access the camera via Snapchat. This happens primarily because Snapchat already has permission to access the camera. More abstractly, an app can still access locked apps that they require since the AppLocker only shows an overlay, and doesn't stop the actual process of the called app. Using the previously mentioned attack vectors, an app with a malicious intent can still access sensitive resources despite these resources being "locked". Therefore, the common Applockers' claim to provide an access control mechanism are incorrect.

4. PROPOSED SOLUTION

Let us consider a different approach to "locking" resources. Instead of using a user-space app to handle the security of resources through apps, we can implement a similar mechanism on the system-level. Such an implementation would require listening to application intents as they happen, and not just when the app is being executed. This way, resources can be blocked off until the user provides authorization. Primitives existing in the Android's application framework can be used to achieve this.

4.1 System-level Solution

In order to provide an access control mechanism as a part of Android system, we would have to create our own variant of the Android kernel and software by configuring the existing implementation and flashing it to a device. This is a difficult task but certainly achievable. For the custom implementation, we propose to pause the communication channel between two process. As previously shown in Figure 4, a process, process A, sends a transaction through the use of a client-side proxy to the Binder driver. The Binder driver then redirects the transactions to the correct process. This is done by parsing the transaction for the destination UID. If the destination process is a process that the user has "locked", then the Binder driver will hold the transaction until the user gives confirmation for the transaction. Our proposed solution is to modify the Binder's implementation to include such a functionality. The Binder implementation is available in the kernel source as part of Android Open Source Project (AOSP). It is located in drivers/android/Binder.c. Another approach that can be taken is to introduce another primitive between the Binder driver and destination process. This primitive will have the same implementation as we proposed to include in Binder driver. The following diagrams depict the aforementioned approaches:

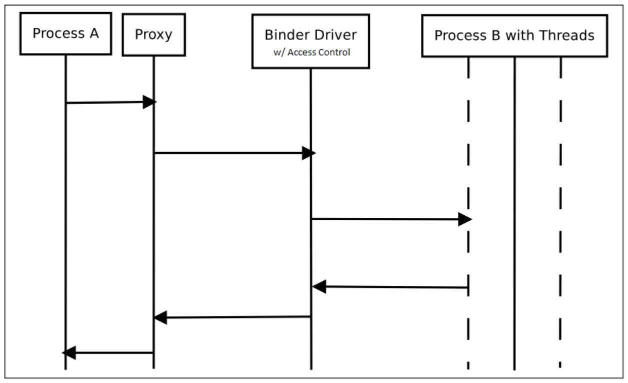


Figure 11: Binder Driver w/ Access Control

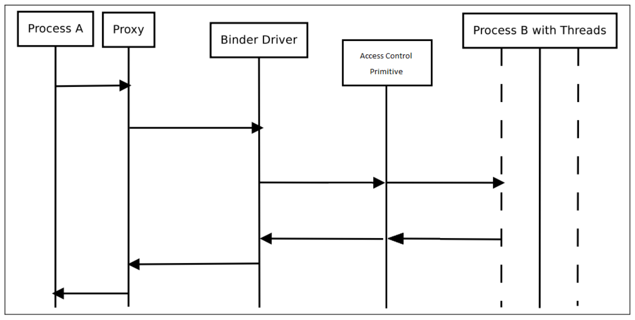


Figure 12: New Primitive

Of course, as mentioned before such a solution would be difficult to implement. In fact, with the time given for this project we could not complete the modifications. At most, we could outline the concept at a higher level with the amount of research done. With this however, we instead attempted at creating an AppLocker of our own and deal with the issues presented in section 3.2 and learned that emulating the solution presented in the previous section would be far more difficult than we had originally anticipated. This would explain why many AppLockers do not deal with the threats mentioned before. Our application is meant to mitigate as many threats as possible without having to root the device, as we assume the common user would not root their device.

4.2 User-level Solution

Our implementation of an Applocker makes use of intents and permission checking. To develop our Applocker, the following tools were used:

- Android Studio 2.2.2
- An Android Device running Android 6.0 (API level 23)[2]
- A class called *MaterialLockView*

We used the class *MaterialLockView* to create the actual pattern screen with unlocking mechanism. The unlocking mechanism was achieved through a simple implementation as outlined in Algorithm 1 and shown in Figure 13:

Algorithm 1 Lock Screen Behaviour

```
1: MaterialLockView object
2: inputPattern = object.patternListener()
3: if !inputPattern.equals(CorrectPattern) then
4:     object.setDisplayMode(Wrong)
5:     object.clearPattern()
6: else
7:     object.setDisplayMode(Correct)
8:     unlockApp()
```

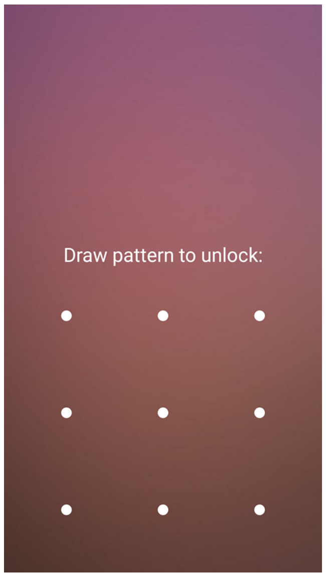
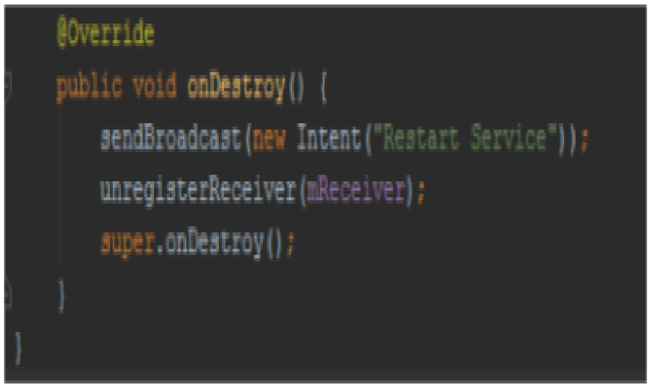


Figure 13: Personal AppLocker

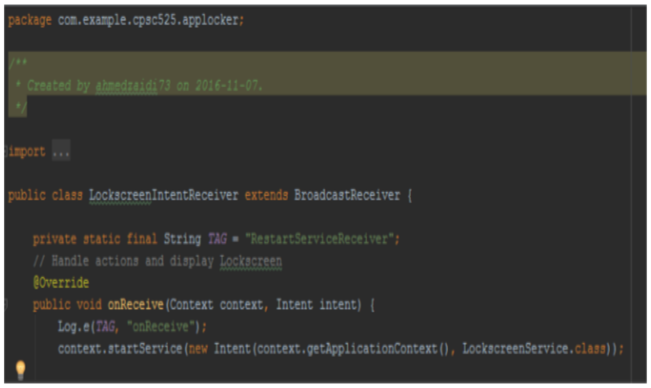
We implemented some basic functionalities such as handling key events, specifically handling the *KeyEvent.KEYCODE_BACK* which is the code associated with pressing the back button. During the design process, we also considered to mitigate the attack vectors we explored earlier. One such attack vector we dealt with was dealing with the app's service being killed. To deal with that we found it appropriate to restart the service immediately if the service is killed by the OS or by another app, such as Greenify. We discovered that the native *onDestroy()* method of an app is called only when a service is not killed by the OS or force stopped. Therefore, in the case that the user forcefully stops the app's service via Settings, only then *onDestroy()* method is called. To provide a constant restart ability, we decided to use a *BroadcastReceiver* which is known for its ability to respond to system-wide broadcast announcements. Once the *onDestroy()* method is called, a broadcast is sent out to the *BroadcastReceiver* which starts the service once again. This is achieved through the use of intents. When the service is about to be destroyed, it send out a message (an intent) to the BroadcastReceiver which recreates the service. In our implementation, the AppLocker's service - "LockscreenService" sent out a broadcast to a "LockscreenIntentReceiver" class which extended *BroadcastReceiver*, as shown in Figure 14. The method *sendBroadcast()* was used where a custom intent was passed to it.



```
@Override
public void onDestroy() {
    sendBroadcast(new Intent("Restart Service"));
    unregisterReceiver(mReceiver);
    super.onDestroy();
}
}
```

Figure 14: Send a Broadcast

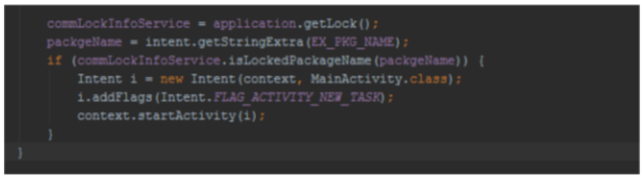
On the receiver's side, the *onRecieve* method listens for an intent and creates a new instance of "LockscreenService". The *startService* method automatically calls the *onStartCommand* method of the service. This is shown below in Figure 15.



```
package com.example.cpsc525.applocker;

/**
 * Created by ahmedraidi73 on 2016-11-07.
 */

import ...

public class LockscreenIntentReceiver extends BroadcastReceiver {

    private static final String TAG = "RestartServiceReceiver";
    // Handle actions and display Lockscreen
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.e(TAG, "onReceive");
        context.startService(new Intent(context.getApplicationContext(), LockscreenService.class));
```

Figure 15: Restarting a Service

At the heart of our proposal we have LockscreenIntentReciever's ability to recieve intents and compare them with apps that the user has already locked. As shown in Figure 16, we first retrieve the locked application's name and store it in a local variable. We then proceed to retrieve the extended data from the intent, primarily the name of the application the intent is intended for. Using an If statement, we compare whether the intent is for the locked application or not. If true, we create a new intent for the AppLocker's MainActivity and start it.

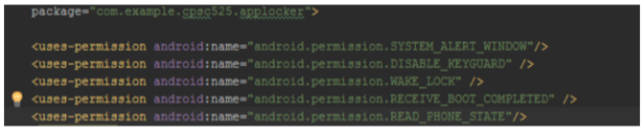


```
    commLockInfoService = application.getLock();
    packgeName = intent.getStringExtra(EX_PKG_NAME);
    if (commLockInfoService.isLockedPackageName(packgeName)) {
        Intent i = new Intent(context, MainActivity.class);
        i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        context.startActivity(i);
    }
}
```

Figure 16: Access Control

As we've mentioned before, apps accessing core system features such as the camera and external storage require the *permissions* to do so. Thus upon installation of our AppLocker, we prompt the user to lock apps that use certain features. It is rather easy to read what permissions applications use by simply retrieving their permissions list. Once read, the apps are grouped up based on what permissions they require, and the user is free to choose which category they want locked. The only downside that this brings to the user is that, even if they don't use the feature behind the permission when using the app, they are still prompted for authorization when the app is executed. This ensures the user's privacy in regards to core features is held.

This might spark the question: what permissions does our AppLocker require? Figure 17 shows the permissions declared in the manifest file. All of these are normal permissions, with READ_PHONE_STATE being the only dangerous permission. This particular permission is required since we found that when there is an incoming call, the lock is still displayed on the screen, not allowing the user to pick up. Sometimes when there is an incoming call, the system decides to close the AppLocker's service. Of course we weren't able to uncover the root cause, given more time we may have been able to fix this. Thus adding this permission was our only choice, as a last resort. In comparison to many AppLockers available on the Play Store, ours does not require that many permissions.



```
package="com.example.cpsc525.applocker">

<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-permission android:name="android.permission.DISABLE_KEYGUARD" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

Figure 17: Manifest file

5. FINAL THOUGHTS

To reiterate, many apps on the Play Store require permissions that don't necessarily have to do with their apparent functionality. To remedy this, the Android OS provides users the ability to allow or deny permissions upon installation of apps that do require them. Of course, many users will choose not to read what permissions that the app needs authorized for the sake of making the installation process seamless.

Our sights then focused on applications meant to help the user in this regard, coined as "AppLockers". At the very least, these provide the ability to put apps behind a lock, to which the user is prompted to unlock prior to actually using the apps. This isn't a great solution since at best, they overlay the locked app while it executes. Similarly, apps that make use of portions of other apps don't prompt the user at all.

After some research, we found that implementing the features that would maximize the user's privacy would be exceptionally difficult if not impossible without the user having root access on the phone. We were under the assumption that the average user wouldn't root their phone since it is a rather involved process depending on the device. Instead we focused on implementing the features on the kernel rather than using an application, to which we could not do considering the amount of time we were given. Of course, this is something that we may work on in the future.

Instead of going the kernel approach, we attempted at creating an AppLocker of our own that would minimize the amount of permissions it needs, as well as mitigate the threats that are apparent against current AppLockers on the Play Store. With more time, our AppLocker can be perfected, but with the time given our app is at best better than only a select few AppLockers already available. This is another prospect we will consider working on in the future, prior to working on the kernel level.

6. REFERENCES

[1] Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2016, sep 2016.

[2] A. Aghazadeh. Api level 23 permissions, apr 2016.

[3] S. Brahler. Analysis of the android architecture. Technical report, Karlsruher Institute for Technology, 2010.

[4] ElcomSoft. Corporate & forensic solutions.

[5] T. Fox-Brewster. Check the permissions: Android flashlight apps criticised over privacy, oct 2014.

[6] A. Gargenta. Deep dive into android ipc/binder framework, feb 2013.

[7] Google. Application fundamentals.

[8] Google. Developer policy center.

[9] Google. Interacting with other apps.

[10] Google. System and kernel security.

[11] Google. System permissions.

[12] Google. Working with system permissions.

[13] Y. Heisler. Watch out, android users: Your lock pattern isn't as secure as you think, aug 2015.

[14] J. Hindy. Are all flashlight apps really out to get you?, dec 2014.

[15] H. K. Jina Kang, Daehyun Kim and J. H. Huh. Analyzing unnecessary permissions requested by android apps based on users' opinions. Technical report, Department of Computer Science and Engineering, Sungkyunkwan University, 2014.

[16] L. G. Li Ma and J. Wang. Research and development of mobile applications for android platform. Technical report, International Journal of Multimedia and Ubiquitous Engineering, 2014.

[17] T. Mahlaeva. Built-in android and ios security mechanisms: Looking at their effectiveness, sep 2014.

[18] U. of Cambridge. What is the price of free?, mar 2012.

[19] B. Rashidi and C. Fung. A survey of android security threats and defenses. Technical report, Virginia Commonwealth University, 2015.

[20] Techtopia. An overview of the android architecture.

[21] P. M. William Enck, Damien Octeau and S. Chaudhuri. A study of android application security. Technical report, Systems and Internet Infrastructure Security Laboratory, The Pennsylvania State University.

[22] K. Yamada. The seven deadly android permissions: How to avoid the sin of slothful preparedness, mar 2013.