Efficient Resolution of Security-Sensitive Values in Android using Abstract Interpretation

by

Dmitrij Petters

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2014

Signature redacted

Certified by.... .....
Martin C. Rinard
Professor of Computer Science, MIT Computer Science and Artificial Intelligence Laboratory
Thesis Supervisor

Signature redacted

Accepted by .....
Prof. Albert R. Meyer
Undergraduate Officer, Department of Electrical Engineering and Computer Science

Efficient Resolution of Security-Sensitive Values in Android using Abstract Interpretation

by

Dmitrij Petters

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis I present a design for an efficient and sound abstract interpretation-based Value Analysis which calculates field values of security-relevant Android API class instances. The analysis is an important component of DroidSafe, an Android malware detection system designed to prove important properties of sensitive program behaviors before the programs appear in an application marketplace. The resolved program values provide important context for other DroidSafe analyses and the generated application summary, improving their precision. This in turn helps a trusted analyst avoid false positives and determine whether a particular application is malicious in a shorter amount of time.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science, MIT Computer Science and Artificial Intelligence Laboratory

Contents

List of Figures

Chapter 1

Introduction

The ubiquity of mobile devices in recent years has led to widespread adoption of a number of mobile platforms, the most popular one being Android [18], holding 79.3% of the total market share [6]. While these platforms provide the core smartphone experience, much of a user's productivity depends on third-party applications (apps). These apps have become an indispensable part of a mobile user's life, providing a wide variety of social, commercial, and other functionality. This growing reliance has increased the negative impact that any service interruptions, erroneous calculations, and misuse of device resources may cause. Another effect of this trend is that apps now have access to more sensitive data than ever, including not only users' personal information but also the data collected via sensors throughout the day. When mobile apps have access to this growing amount of sensitive information, they may leak it carelessly or maliciously [17]. Gaining assurance that these or any other malignant behaviors (malware) are not going to occur is a critical challenge for smartphone platforms, security-conscious users, and any other organizations looking to tap into the success of mobile platforms.

1.1 Mobile Malware

As the adoption of smartphones has grown, so have the incentives for malware authors to come up with new ways to circumvent existing mobile malware detection

techniques. The trend is not likely to stop - the growing use of mobile devices as a security check, usually as a form of secondary or two-factor authentication for user credentials or online transactions, opens the door to even bigger potential profit from successful malware attacks. The risks that mobile users face are hence greater than ever, and the best way to address them is not yet known.

What is known is that malware authors focus their efforts on the most widely-used platform - out of 259 new threat families and new variants of existing families discovered in Q3 of 2013, 252 were Android threats. [2]. For this reason, and others, I decided to address these challenges in the context of Android, the most popular mobile platform, holding 79.3% of the total market share [6].

So far, the vast majority of Android malware remains unsophisticated. Recent studies have shown that most malicious apps are repackaged versions of existing clean apps [26] with inserted malware that either leaks private data or abuses allowed functionality [9]. Both of these attacks are easy to implement due to Android's course-grained permissions. In the next three sections I describe these two classes of malware and argue that despite being easy to implement, they are still difficult to detect using current malware-detection approaches, especially if the author puts in the effort to hide them.

Data Leakage

Access to security-relevant parts of Android's API is controlled by an install-time application permission system. Each application must declare upfront what permissions it requires and the user is informed during installation about what permissions it will receive. This shifts the focus on security to the end-user - if he or she does not want to grant a permission to an application, he or she can cancel the installation process. The problem is that less than 17% of all regular users understand or even pay attention to this requested permissions list [11].

Even a bigger problem, and one that affects even the most advanced of users, is that these permissions are course-grained and the system provides little to no control over how any disclosed information is actually used. Let us suppose an SMS message

backup app asks for the READ_SMS and INTERNET permissions. Based on the app name, the permission requests seem reasonable and raise no red flags. However, without the source code and a significant time investment, a user has no way to gain complete assurance that his or her messages are indeed sent only to the backup server.

Consider the snippet of source code presented in Listing 1.1, which may very well be a part of the example app. Using available API calls, the use of some of which requires the requested READ_SMS and INTERNET permissions, the app sends the text messages to the address "ads.doubleclick.net", an ad server. This is clear abuse of the privileges that the user gave to the app when first installing it. Even worse, the user will never even suspect that the data leakage is occurring and may continue to willingly use the app.

Third-party app and library developers have taken advantage of this shortcoming and therefore it is not uncommon to find private information being leaked. A recent study of 30 popular apps chosen at random from the Android app marketplace showed that two-thirds of them exhibit suspicious handling of sensitive data and that half of them sent location data to advertisement servers [8]. Only two presented EULAs and neither of them indicated the above practices.

```
1  Uri uri = Uri.parse("content://sms/inbox");
2  Cursor cursor = getContentResolver().query(uri, new String[]{"body"}, null, null, null});
3  String allMessages = "";
4  while (cursor.moveToNext()){
5      allMessages += ";" + cursor.getString(0);
6  }
7  HttpClient httpClient = new DefaultHttpClient();
8  HttpPost httpPost = new HttpPost("http://ads.doubleclick.net");
9  httpPost.setEntity(new ArrayList<NameValuePair>({'messages':allMessages}));
10 httpClient.execute(httpPost);
```

Listing 1.1: Example Data Leakage in SMS Backup App

Two key observations can be made about the above example:

1. The leaked data originates in the API, using the ContentResolver.query method. Assuming coding practices encouraged by Google, other sensitive data besides SMS messages can also only be queried in this manner, through an API-guarded

method. It thus may be possible to analyze API method usage to paint a finer picture of how an application handles sensitive data.

2. Analyzing API usage must be supplemented with the resolution of the related context. It is not enough to know which methods are called to figure out what data is read and how it is used in the example above. The values of some key variables must also be determined - the string passed into Uri.parse, the argument to HttpPost, and the ArrayList contents passed to setEntity.

Abuse of Functionality

The second common class of Android malware involves mobile apps performing actions that are not necessary to accomplish their target functionality. The common example used due to its severe invasion of privacy is a phone secretly recording video or audio even when the user has not pushed the record button. Any app which is given the RECORD_VIDEO or RECORD_AUDIO permissions is capable of doing this. While not as disturbing, the example in Listing 1.1 is also a great demonstration of abuse of allowed functionality. The app gained the INTERNET permission with the promise to back up SMS messages to the cloud, but then abused the privilege by also sending the messages elsewhere.

It is again important to note that the only way for the app to maliciously send the messages through the internet is to call API-provided methods, with the messages as payload. Hence an analysis of the API calls made and respective context can be used to determine that this malicious activity is taking place.

1.2 My Approach

Based on the observations made above, I developed a pre-install malware detection system aimed at presenting possible API-guarded information flows and their context. In this thesis, I present an enabling contribution to this system, a global context-sensitive inter-procedural Value Analysis which calculates the values of important fields of objects of security-sensitive Android API classes. The resolved program

values provide important context for the detected information flows and can be incorporated into other analyses, improving their precision.

Because many of these flows originate in and flow through the API, I performed all of my analyses in the context of the entire API, modified to capture possible dynamic behaviors not expressed in the original source code.

I used abstract interpretation to scale Value Analysis to be able to analyze large applications alongside the entire API. I also made the underlying heap abstraction offered by the points-to analysis flexible, allowing precision to be traded off for scalability with ease.

I also decided to perform an analysis of Android API usage patterns using clustering techniques in order to try and get a signature of an app based on how it uses the API. I think that this signature may eventually help detect malware that comes in the form of repackaged clean apps.

1.3 Background

Before exploring my approach in-depth, it is useful to understand the limitations of other malware detection techniques and the kinds of challenges that any static analysis must overcome to successfully analyze an Android app.

1.3.1 Current Malware Detection Technique Limitations

Manually curating app store

The most common distribution methods of third party apps, app stores, have made the production and consumption of apps easy. They have enticed developers by placing low economic and technical barriers to entry and streamlined purchase and installation to serve even the most casual users with ease. These characteristics have allowed app markets to reach staggering levels of success. One of the best known, Apple's App Store, launched in 2008, averaged about a $1 million in application sales a day in its first month of existence [24], and served nearly 3 billion applications in

just 18 months [4].

To release an app through the Apple App Store, a third-party developer has to participate in Apple's iOS developer program and submit the app to Apple for review. The app is signed and published only after it passes the review process. The curation process is highly secreted but I do know that each app is examined by at least two reviewers and 95% of apps are approved within 14 days. The presence of a human in the chain as well as the sheer number of apps and updates being published make this a very time consuming process, but one that has been worth it as Apple has stuck to it for over 5 years now.

The catch is that the manual curation approach is only as effective as the underlying analyses that the analyst relies on to help him or her detect malicious behaviors. Based on the volume of apps submitted and the number of reviewers that Apple employs, each reviewer spends less than 10 minutes on per app [1], which is not nearly enough to analyze an entire application's source code. The only way an analyst could vet an entire app is for there to exist technologies capable of focusing his or her attention on just the security-relevant parts.

Limiting the API

A required manual inspection of every submitted app is not the only thing that has earned iOS the label of a closed platform. Apple also bans developers from using use a subset of the available API calls, dubbed "private API calls". These calls restrict access to certain security-sensitive resources in an effort to protect them. Nothing actually prevents a developer from using these calls, but doing so is almost guaranteed to get an app rejected during the curation process. (Although a security researcher has demonstrated that it is possible to successfully hide their use enough to get an app approved [23].)

Limiting developers' access to functionality and sensitive information, such as device ID, may increase the difficulty of writing successful malware, but it may also stifle innovation. Apple may have achieved a temporary balance between the two that works, but no one can argue against giving developers full API access if the apps

they write could be vetted for malware with high assurance.

Software Emulation

The vast difference between manual inspection and the technique presented in this section, automated software emulation, is representative of the major difference between Apple's and Google's approach to their mobile platforms and app stores. Unlike Apple, Google does not limit the API and does not manually curate their app store. The lower standards and more powerful API make it easier for developers to publish apps, a fact reflected in the store's huge app publishing rates, which vary between 7500 and 22500 per month [3].

Unsurprisingly, these rates limit the amount of resources that an automated malware detection system put in place of manual inspection can expend per app. Such a system, named Bouncer, was deployed in 2011. It uses software emulation in an attempt to detect malware in apps uploaded to Google Play and Android developer accounts [15]. Each app that is tested is loaded in a software emulator using Google's cloud infrastructure which simulates an Android device. Bouncer watches for indications that the application might be misbehaving, and compares it against previously analyzed apps to detect possible red flags. If the application does not do anything suspicious, it is given a pass. Google claims that Bouncer was responsible for a 40% drop in the number of possibly-dangerous programs available on Google Play between 2011 an 2012 [15].

Security researchers were able to quickly find ways to for malicious apps to circumvent Bouncer. For instance, Bouncer's analysis is purely dynamic: it only flags apps that misbehave during the five-or-so minutes Google runs the app in the emulator. If an app is subtle and just waits for a while before engaging in risky behavior, it could be categorized as safe. Similarly, Bouncer seems to use a very limited set of contacts, pictures, and other fake personal information, making it easy for malware authors to special-case those items and avoid reading them. Bouncer does let the apps it is testing connect out to the Internet; however, those connections all come from IP ranges easily identified as Google, making it simple for malware developers to

let remote Web services behave differently in Bouncer's environment than they would on an Android device in the wild. Google has been updating Bouncer to work around some of these issues, but the fact remains that malware that delays its attacks long enough to evade Bouncer's scrutiny will probably still pass. Similarly, apps that have totally innocuous installers but then download malware via update mechanisms can bypass Bouncer entirely.

Signature-based Detection

Android 4.2 Jelly Bean includes a signature-based app verification service as part of the Google Play app [13]. Once app verification is activated, the service sends the application's name, URL, and a unique signature string (a checksum) representing a scan of the application's files to Google. Google then compares that information to data in its records about known malware apps: if there is a problem, the Google Play app then presents a warning to the user or blocks the application outright.

Unfortunately prior research has demonstrated this technique to be only marginally effective. Out of 1,260 samples of Android malware (representing 49 different "families"), Google's App verification service detected just 193 of them, or a bit over 15% of the total [14]. Google will likely improve its app verification service, it nevertheless will always be playing catch-up to malware authors. Android malware developers are known to mutate and repackage their malware so it can have different checksum values and thus avoid detection.

In this thesis I investigate Android API usage pattern detection using API call clustering as a means of extracting a signature of an app that is representative of the way the app uses the API. This improved signature may prevent malware authors from being able to mutate and repackage their apps to avoid detection by systems such as the Google Play verification service.

Dynamic Analysis Techniques

Dynamic analysis techniques have been used by many proposed mobile malware detection systems as a way to detect privacy leaks [8], UI-based trigger conditions [25],

and other sensitive behaviors. On the whole, dynamic analysis techniques can be broken down into two groups, pre-install techniques, and runtime techniques.

Pre-install Techniques Pre-install techniques are typically used to test a program for malicious behaviors by exercising various code paths. Doing so may reveal subtle defects or vulnerabilities whose cause is too complex to be discovered by static analysis alone. Achieving high coverage by exercising every possible code path, however, is very difficult to do and the failure to do so may lead to undiscovered malware.

Pre-install techniques also require significant configuration. For example, many ad libraries check if the app they were bundled with has a given permission before utilizing it - e.g. only if they have the permission to access location data will they send this information to an ad server in an effort to localize ads and increase revenue. There is nothing preventing ad libraries from checking if they have access to any number of types of sensitive information and attempting to leak them only if they are able to. If the apps are not carefully configured and do not declare enough permissions, a dynamic analysis approach could watch many applications with a malicious advertising library and never see this functionality.

Runtime Techniques Runtime techniques are different from pre-install techniques in that they monitor instrumented program execution on the device itself instead of in a controlled environment. This requires changes to the runtime and incurs overhead costs, which can be prohibitive for a mobile phone with limited CPU power and memory. The exact slowdown is dictated by the complexity of the analysis. The best dynamic information-flow tracking system out there today, Taintdroid, already incurs a 14% performance overhead [8] without even attempting to track implicit flows.

A dynamic monitoring approach also begs the questions as to what should be done if malicious activity is detected while the app is being used. Immediate termination may seem appropriate depending on the severity of the detected

harmful activity, but may be even more detrimental than the malware itself if the app is being used in a mission-critical environment.

1.3.2 Analyzing Android Statically

I forgo all the techniques described above in favor of static analysis, or analysis used to infer properties directly from the app source or an Android Application Package (APK) without ever having to run it.

Unlike dynamic analysis which requires the explicit identification and testing of every possible code path, a sound static analysis automatically offers complete code coverage. It thus has the benefit of discovering vulnerabilities that may not be exposed at runtime or during testing. This is important if I am to try and achieve high assurance of the lack of malware.

Static analysis of Android does not come without significant challenges - the framework uses many event-driven idioms, is made up of a mixture of programming languages (mainly Java and C), and apps written for it are made up of very dynamic components.

Driving Example: PickContact

For the purpose of making the challenges posed by Android framework concrete, I will explore them in the context of a simple, clean app called PickContact[1].

Upon start-up, the app presents the user with four different buttons, as shown in Figure 1-1. Depending on which button the user presses, he or she gets to pick one piece of contact data of the chosen type and then have it displayed back to them in the form of a toast (a small pop-up message).

In the next three sections I highlight characteristics of Android that make it difficult to understand this relatively simple behavior from a static analysis point of view.

[1]Source code in its entirety is available in Appendix A

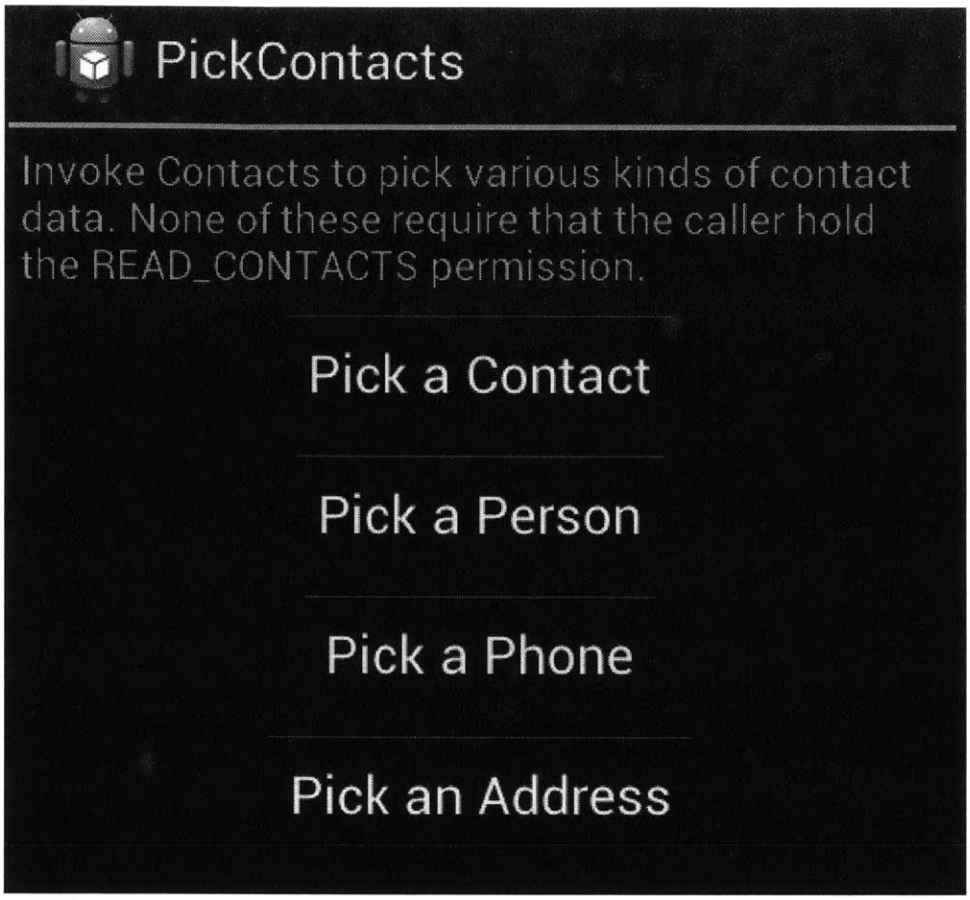

Figure 1-1: PickContact UI

Event-Driven Architecture

The first thing that happens when the application is launched is that its onCreate function, the abbreviated version of which is shown in Listing 1.2, is called. This function is an example of an entry point - a method that is defined by the app and intended to be called only by the framework. It is also an event listener, or a function called in response to a particular event happening. In this case, the onCreate function is called in response to an instance of its enclosing class, the first and only Activity, being created by the runtime when the app first launches.

```
1  protected void onCreate(Bundle savedInstanceState) {
2
3      // Set the onClick listener to an instance of ResultDisplayer
4      ((Button)findViewById(R.id.pick_contact)).setOnClickListener(
5                  new ResultDisplayer("Selected␣contact",
```

```
6                   ContactsContract.Contacts.CONTENT_ITEM_TYPE));
7       ...
8       }
```

Listing 1.2: Abbreviated PickContact.onCreate

This event-driven pattern is repeated all throughout Android - every single UI element in an app can define multiple event listeners to be called at different moments as various events happen. In my app's case, the onCreate method sets an onClick event listener for each of the four buttons presented to the user. The listener in each case is an instance of the ResultDisplayer class, whose onClick method, shown in Listing 1.3, gets called whenever the button is pressed. These event handlers are called by Android during runtime, and the failure to properly identify and include them as entry points in a static analysis may cause the analysis to be incomplete. Another important thing to note is that the Button class which implements the onClick method will be called in four different contexts, one for each of the four buttons. Unless a static analysis is able to separate out these contexts, they will be conflated in the final results.

Inter-Component Communication (ICC)

Android application components interact through Inter-Component Communication (ICC) objects - mainly Intents. Developers can specify an Intent's target component (or components) in two ways:

Explicitly By specifying the target's application package and class name.

Implicitly By setting the Intent's action, category or data fields.

Resolving the target of these intents is important for the purposes of accurate and precise static analysis such as information flow analysis. Intents may carry data in the form of key-value mappings or context-specific references to external resources or data. If this data is considered sensitive, then knowing precisely where it may flow may decrease the number of false-positive data leaks detected. Ignoring it, on the other hand, could lead to unsound results.

```
1  public void onClick(View v) {
2
3      // Build up intent
4      Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
5      intent.setType(mMimeType);
6      mPendingResult = this;
7
8      // Start an activity using the intent
9      startActivityForResult(intent, 1);
10 }
```

Listing 1.3: ResultDisplayer.onClick

The Android runtime system uses a late-binding mechanism for invoking system services that relies on passing string arguments to identify the system services (and arguments) to be invoked. For example, rather than a direct Java API call to invoke a system service (e.g. read the device address book), in an Android app the programmer instead will construct an Intent with specific fields set to values that will cause it to target the desired service.

This can be seen in ResultDisplayer's onClick method, where an implicit Intent is built up and used to start an Activity. The action field of it is set to a constant Intent.ACTION_GET_CONTENT, while the type of the intent is set to the mMimeType variable, as seen in Listing 1.3 line 5. These two fields determine which component this Intent may target and eventually the system service that will be invoked. In this particular case, it is impossible to tell what kind of data is going to be retrieved by looking at the onClick method alone - I do not know the value of mMimeType. It is an attribute of the ResultDisplayer instance that was passed into the constructor when that instance was created in PickContact's onCreate, an entirely different section of the application code.

In this case the application only consists of one Activity and I knew beforehand that the Intent was going to be handled by the API. However, if the application had more activities, I would want to know precisely which of these activities may receive it and which never could. The only way I could do so is if I knew the values of the Intent's action, type, and data fields unambiguously.

ICC is a great demonstration of how values of Android API objects can significantly affect the behavior of an application. Any static analysis that wants to gain a complete picture of an app's behavior has to be able to determine these program values and their impact on the call graph, component interaction, and visual elements.

Android API

As seen in 4, the chosen contact information is referenced by a Uri that is the value of the data field of the Intent passed into onActivityResult. This Uri is retrieved and stored in a local Uri before being used to query for the actual contact data which is subsequently leaked onto the phone screen.

The actual contents of the data field are directly dependent on the value of the type field of the Intent passed to startActivityForResult in PickContact's onCreate. However, the actual translation of the type field of starting intent to the data field of the resulting intent, as well as the invocation of onActivityResult, both happen deep in the Android API in native code.

```
1  void onActivityResult (int reqc, int resc, Intent result){
2
3      // get the Uri referencing the chosen contact data
4      Uri uri = result.getData();
5
6      // get the contact data
7      ContentResolver cr = getContentResolver();
8      Cursor c = cr.query (uri, null, null, null, null, null);
9          c.moveToFirst();
10         int id = c.getInt(0);
11
12         // show the contact data in a Toast
13         Toast t = Toast.makeText (this, id);
14         t.show();
15 }
```

Listing 1.4: Abbreviated PickContact.onActivityResult

The Android API is huge and complicated. Tracking down an information flow such as this that flows through native code is pretty much impossible using a direct analysis of the API. For this reason, any static analysis that wants to remain sound

requires a very complete API model, supplemented with code necessary to catch these dynamic flows and behaviors that would otherwise be missed. Making sure that such a model captures all side effects is very difficult but necessary for any static analysis that wants to be accurate.

1.4 Thesis Contributions

The contributions of this thesis are -

- A design of an efficient and sound Value Analysis which leverages abstract interpretation to determine field values of security-relevant Android API class instances. I include the abstractions I developed and scalability optimizations I made that enabled the analysis to scale to large apps analyzed alongside the entire Android API.

- A client component of Value Analysis that uses the resolved field values to successfully determine app component interaction and refine the Information Flow Analysis.

- An evaluation of the techniques presented in this thesis on 52 examples of malicious Android applications listed in section 4.4. These applications were provided to us by a third party military contractor (a "red team") and designed to be next-generation malware, using techniques not currently in the wild. I show that my implementation of Value Analysis and the underlying analyses strike a good balance between precision and scalability, enabling us to assert at pre-install time that large classes of malware may or may not ever occur.

- A workflow for identifying common API usage patterns from large sets of Android applications using clustering techniques, which may be used to detect malware in the form of repackaged clean apps.

1.5 Summary of Qualitative Results

My approach and its results demonstrate that given the right choices of sensitivities, program values that impact security-sensitive flows can be resolved with high-precision. Incorporating these resolved values into later static analyses increases the precision of my pre-install malware detection system. This is reflected particularly well in a malicious Sudoku-solving application from my app dataset. The 49 tainted program values and 1 sensitive flow into a sink[1] detected by the information flow analysis were found to be false positives when more precise ICC Resolution was implemented, enabled by the program values resolved by the Value Analysis.

1.6 Summary of Quantitative Results

For my benchmark suite of 52 Android Apps listed in section 4.4, Value Analysis was able to unambiguously resolve the possible values of targeted security-sensitive object fields 97.53% of the time. A subset of these values relevant for ICC resolution enabled us to unambiguously resolve the possible targets of 96% of all Intents (100% for explicit, 75.44% for implicit). The determined in-app intent destinations were used to make my Information Flow Analysis more precise, resulting in a 15.18% decrease in the number of program values that have information flow taint and a 14.71% reduction in the number of sensitive flows into sinks for a set of 10 applications with existing inter-component information flow.

1.7 Chapter Summaries

The rest of this thesis consists of the following five chapters, followed by a conclusion.

[1]Sinks are defined as API calls which expose data beyond application boundaries such as a non-private file system, network, SMS, the logging framework, email, etc

1.7.1 DroidSafe Chapter

In the DroidSafe Chapter I give an overview of the end-to-end malware detection system in the context of which Value Analysis was developed, describing the various phases and its output.

1.7.2 Background Analyses Chapter

In the background analyses chapter I discuss two supporting analyses of Value Analysis - string analysis and point-to analysis. I describe how the system employs a modified version of an off-the-shelf string analysis to resolve a variety of complex string manipulations instead of only being able to handle string constants and give a comprehensive overview of the kinds of sensitivities I experimented with before finding a combination that gave us the most precision in the context of Android. In particular, I discuss the various degrees of context and object sensitivity I tried and why I settled on selective object sensitivity and no context sensitivity.

1.7.3 Value Analysis Chapter

In the Value Analysis chapter I present, describe, and evaluate the main contribution of this thesis - a scalable and sound Value Analysis that determines field values of objects of security-relevant Android API classes using abstract interpretation.

1.7.4 Inter-Component Communication Chapter

In the Inter-Component Communication (ICC) chapter I present and evaluate a client component of Value Analysis, the ICC Resolution transformation. This transformation uses the results of Value Analysis to gain an understanding of how an application's components may interact and uses this information to refine the semantics of the modeled application, improving the precision of the information flow analysis by 15%. This increased precision manifests itself in the form of a lower number program values that had information flow taint and sensitive flows into sinks.

1.7.5 Android API Call Clustering Chapter

In the Android API Call Clustering chapter I take a look at the possible ways in which identifying common Android API usage patterns may be used to identify malware in the form of repackaged clean apps. I describe a workflow using which API usage patterns can be extracted from large samples of Android applications and present qualitative and quantitative results of executing an initial version of this workflow.

Chapter 2

DroidSafe

This work was developed in the context of DroidSafe, an automated pre-install end-to-end malware detection system capable of analyzing Android applications deployed in application marketplaces. Its end output is a succinct representation of the application it is analyzing, called the security specification. This summary of the application is 10x+ shorter in length than the original application source code and analyzable through an Eclipse plugin. The system decreases the amount of time that it takes a security analyst to vet an application against its expected behavior. The promise is that the security summary will present all sensitive actions possible from an application, with necessary context, such that a trusted agent can make an accurate determination as to whether malicious behaviors are possible.

2.1 System Architecture

The DroidSafe system, presented in /autoreffig:droidsafe-architecture is made up of a series of analyses and transformations, roughly broken up into two stages -

Generation of the Modeled Application Before whole-program analyses can be applied to the distributed form of the application, a number of transformations are performed. Elements defined in XML are inflated into their corresponding View objects. Entry points into the application are discovered so as to

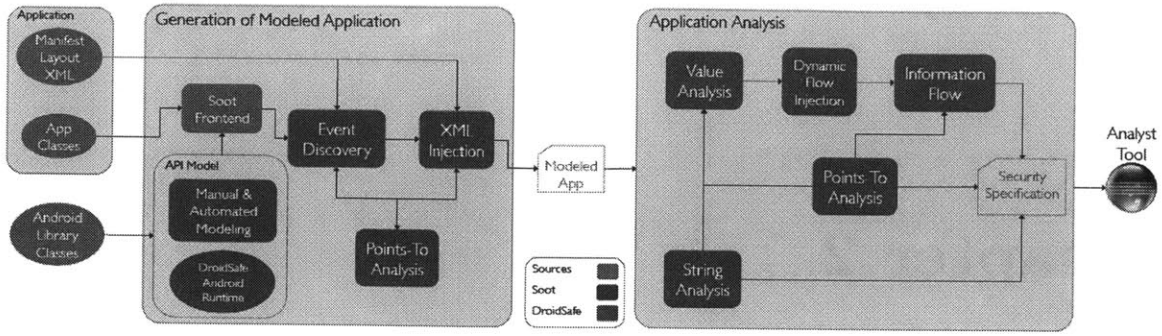

Figure 2-1: DroidSafe Architecture

make sure. The application is supplemented by my model of the Android API, generating the modeled application.

Application Analysis A series of whole-program analyses are ran to determine the security-relevant behavior of the app that is to be presented to the analyst vetting the application.

2.2 Security Specification

DroidSafe's end output is a high-level succinct representation of the application it is analyzing called the security specification, summarizing the application's actions and flows. The format of this specification is highly tied to the fact that Android applications are event driven. For each event handler, the specification lists security-relevant API calls that may be triggered by the event. If a call appears in the specification, then it is not guaranteed to be called at runtime but only may. Because the security-relevant behavior of an Android application, such as communication, is limited to what is made possible by the Android API, the listed API calls provide a holistic overview of an application's security-relevant behavior.

The security specifications are compact, providing a 10x+ reduction from the number of lines in the original source code and allowing the security analyst to focus on security-relevant behavior of the code. The goal is that with enough resolved context, the security summary can supplant manual source code inspection.

2.3 Android API Model

The Android API source code is large and complex. Many calls are overloaded, there are multiple ways of performing the same action, and its event-driven architecture makes it difficult to build a precise and compact abstract program graph. Its direct analysis would lead to overly conservative conclusions and drastically increased analysis times.

For these reasons and others, DroidSafe instead relies on a modified version of the Java Android API source code, supplemented with Java code that captures possible dynamic behaviors not expressed in the original source code, along with points-to, information flow, event, and other security information. These behaviors are normally implemented in native code, which is currently out of scope for DroidSafe.

Analyzing the application source code in the context of this augmented android source is one of the distinguishing features of DroidSafe, increasing assurance, precision, accuracy of analysis results. Before the system, it was beyond the realm of precise static analyses to analyze applications in the context of the entire API. But in DroidSafe, even a simple 100-line app is analyzed alongside 1.3 MLOCs of the modeled Android API. This allows the system to track the various behaviors, callbacks, and flows more accurately and completely than other systems which have more primitive summarize or altogether ignore the API.

2.4 Execution Phases

The DroidSafe system analyzes an application in a series of phases. A phase is usually an analysis or transformation that may rely on the results or effects of a previous phase. The rough ordering of the phases with respect to each other is shown in /autoreffig:droidsafe-architecture and the purpose of each is summarized below.

2.4.1 Harness Creation

As described in section 2.3, DroidSafe analyzes an app in the context of the entire Android API. The harness creation phase 'plugs' the app into this this model by identifying and calling the entry points in the app code.

2.4.2 XML Injection

Android incentivizes developers to define the UI of their apps using XML instead of Java with the promise that Android will then automatically support different screen resolutions, sizes, orientations, and different languages. Doing so is considered best-practice, as it helps separate the presentation of the app from the code that controls its behavior. When the app gets compiled, each XML layout file is turned into a View resource, the allocation of which is inserted into the harness. During runtime, this View is inflated in a way that best suits the user's device and settings.

It is important for DroidSafe to mimic the injection of all these visual elements because many event-driven behaviors of the app may be closely tied to them. For example, an onClick listener may be tied to a button with some specific label. The label's text is vital information for a security analyst attempting to classify the handler code that gets executed when the button is clicked as malicious or benign.

2.4.3 String Analysis

A precise string analysis, even when the string is not fully determined, may yield information that simplifies an analyst's task. For example, consider a string passed to the constructor of a URI that could not be fully identified but instead is determined to be characterized by the regular expression "http://darpa.mil/.*". If the analyst determines that darpa.mil is a trusted domain, then any requests that use that URI can perhaps be classified as benign.

DroidSafe accomplishes this with the string analysis phase, which is based on the Java String Analyzer (JSA), a Soot-based tool "for analyzing the flow of strings and string operations in Java programs."[10][5]

The tool allows DroidSafe to indicate 'hotspots', string expressions in the program that should be analyzed. As an example, running JSA on an Android app with the first argument of the 'android.content.Intent: void ¡init¿(String)' API method identified as a hotspot will produce a characterization of the string (corresponding to the action of the Intent) passed to each individual Intent constructor. JSA generates, for each hotspot, an automaton characterizing the possible string values for that argument.

DroidSafe's string analysis phase is a run of JSA, followed by a custom conversion of the determined automatons into easy-to-read regular expressions. Both of these are described in more detail in section 3.1.

2.4.4 Points-to Analysis

DroidSafe uses an insensitive Andersen-based Points-to algorithm (PTA) to determine the set of possible allocation sites for each object reference in the complete app code. This information is used in multiple later analyses. PTA is described in detail in section 3.2.

2.4.5 Value Analysis

Value Analysis is the core contribution of this thesis sand resolves possible values of fields of objects I deem security-sensitive. The resolved values are displayed in the generated security specification and are integral to some of other analyses that run after it, such as the information flow analysis.

2.4.6 Information-Flow Analysis

The Information Flow Analysis phase runs last. It is a high-precision data flow analysis which enables sensitive information to be tracked from source to sink in the modeled application.

The analysis begins by identifying sources of sensitive information within the application using the results of Value Analysis and classifications present in the modeled

Android API. It then calculates flows for values at each program point, resulting in a set of ways that sensitive information may be used.

The identified sensitive flows into sinks give the analyst insight into behaviors of the app that would be difficult to track down without meticulously analyzing the application code.

2.5 Related Works

There have been a number of end-to-end systems developed that attempt to detect malware in Android applications -

TaintDroid TaintDroid [8] provides system-wide dynamic taint tracking for Android. It discovered 68 potential information misuse examples in 20 applications. Unlike DroidSafe, TaintDroid focuses solely on data flow and does not consider action-based vulnerabilities. Also, TaintDroid is meant to be a post-production tool for real-time analysis, while Droidsafe can be used as either a pre- or post-production tool. TaintDroid and Droidsafe are complementary tools.

SmartDroid SmartDroid [25] uses a mix of static and dynamic analyses to detect UI-based trigger conditions that may expose sensitive behavior of several types of Android malware. However, unlike DroidSafe, the SmartDroid tool does not detect sensitive behaviors that are not UI-related or data-dependent.

AndroidLeaks AndroidLeaks [12] is similar to DroidSafe in that it aims to reduce the amount of effort that it takes for a security auditor to vet an application. It focuses on automatically finds potential leaks of sensitive information in apps, presenting a number of traces for the auditor to verify manually. While their work is focused specifically on data leakage, DroidSafe's output allows an analyst to check for a number of additional malware types, such as functionality abuse, denial of service, and time and state attacks.

2.6 Summary

In this chapter I gave an overview of the end-to-end malware detection system in the context of which Value Analysis was developed.

Chapter 3

Background Analyses

3.1 String Analysis

A static characterization of strings manipulated in an Android application allows the Droidsafe tool to generate summaries that are significantly more informative than direct source code inspection.

The DroidSafe string analysis is an extension of the existing Java String Analyzer (JSA) program analysis framework. In this section, I give a brief summary of JSA and describe a number of extensions and modifications that have been implemented to make it more useful for DroidSafe.

3.1.1 Driving Example - UltraCoolMap

Malicious applications may utilize code obfuscation techniques of varying sophistication to hide the true behavior of the code from manual inspection. Consider the following code fragments in Listing 3.1, taken from one of the application in my dataset, UltraCoolMap.

```
1      String real_badlyName = "http://maps.google.com/maps?saddr="
2                              + (float) realBadName / 1000000 + ","
3                              + (float) realbadName / 1000000 + "&daddr="
4                              + (float) realBadname / 1000000 + ","
5                              + (float) real_bad_name / 1000000;
```

```
6
7       String really_BadName = really_bad_Name.substring(0, 18);
8       really_BadName = really_BadName.concat("-");
9       really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
10      really_BadName = really_BadName.concat(".cc");
11      really_BadName = really_BadName.concat(really_bad_Name.substring(22));
```

Listing 3.1: UltraCoolMap String Obfuscation Example

While it is obvious that the application is making a deliberate attempt to obscure the source code behavior, the string analysis allows DroidSafe to infer that the calculated string 'really_BadName' represents a URL of the following format -

http://maps.google-com.cc/maps?saddr=⟨*float*⟩,⟨*float*⟩&daddr=⟨*float*⟩,⟨*float*⟩

The '⟨*float*⟩' portions of the URL indicate parts of the string that are dynamically determined but will necessarily have the lexical form of a printed Java 'float' value. Further inspection of the application reveals that the 'float' values represent geographic positions. Though the initial prefix of this URL in the first line of the code fragment appears to be 'http://maps.google.com' (ostensibly a trusted network endpoint), the string analysis results reveal that the true network endpoint is (suspiciously) 'http://maps.google-com.cc'.

3.1.2 JSA Background

The Java String Analyzer [5] [10] is a Java program analysis framework that performs a precise analysis of string expressions occurring within a Java program. The design of the this analyzer is well documented, but I briefly summarize the method that JSA uses to infer string values to give the context in which my Droidsafe improvements and extensions to JSA are situated.

First, JSA converts a Jimple intermediate representation of the app into a string analysis intermediate format. This intermediate format preserves the method and control-flow structure of the Jimple representation, but abstracts values into a small

lattice of types (roughly segmented into String-like types, Primitives, Arrays, known-null values, and other irrelevant types). The conversion to intermediate form inserts JSA-specific 'assert' invocations that capture control-flow induced invariants For example, the body of an if statement will be amended with an assertion that the if predicate is true. These assertions will be used in later stages of JSA analysis after control-flow structure has been removed.

From intermediate form, JSA then creates a flow graph which more abstractly characterizes the program behavior. In this format, most of the control-flow of the program is discarded, with the graph nodes representing variable initialization or assignment; string concatenation; or one of a small collection of pre-defined string operations (e.g. 'split', 'replace') or a string assertion inserted during conversion to intermediate form. Conversion from intermediate form to flow graph representation utilizes a number of program analyses, including field usage analysis, liveliness analysis, detect invalid alias assertions, alias analysis, reaching definitions analysis, and detect invalid operation assertions [jsamanual2009].

Following conversion to a flow graph, a context-free grammar is constructed that characterizes the language of strings accepted at each program point. Rather than produce a grammar for every possible intermediate string value in the program, JSA exposes an API for identifying specific 'hotspots', which are specific program points of interest. DroidSafe is only interested in hotspots corresponding to String arguments (and String-valued results) of a predetermined collection of Android API endpoints. Identifying a subset of the such program points reduces the size of the analysis.

From the generated context-free grammar, JSA then constructs a regular grammar that approximates the context-free grammar. The language of the regular grammar contains the language of the context free grammar that the regular grammar is derived from. This regular grammar is then converted to a space-efficient multi-level finite automata representation, and finally each hotspot is converted into a deterministic finite automaton (DFA) that recognizes the language of the hotspot.

String operations and assertions inserted in conversion to intermediate form are preserved through the various stages culminating in DFA generation. At the level

of DFA, the string operations and assertions are implemented as manipulations on finite automata. For example, a string-prefix extraction operation with unknown end position, performed over an input DFA is implemented adding ϵ-transitions from each non-accepting state to an accepting final state. Defining string operations over automata allows the operations to be eliminated from the resulting automata, at the cost of occasional explosions in the size of the resulting automata.

3.1.3 DroidSafe-Specific Extensions

A number of extensions and modification to JSA have been implemented to make the analysis output more directly usable for DroidSafe.

Regular Expression Generation Recall that JSA generates a DFA which accepts (a sound approximation of) the language of values that may flow to a hotspot. The JSA manual describes two main modes to use these calculated automata - either they can be used with source-code annotations for runtime assertions (and to refine the JSA analysis) or by performing the static analysis then extracting the generated automaton for inspection. It is not clear that the first use case is relevant for DroidSafe (as it requires source modification), while the second use case yields automata, which are not suitable for cursory inspection. Contrast the regular expression for URL given in the example at the beginning of this section with the (hypothetical) DFA recognizing the language of the regular expression.

To make the results of JSA amenable for human inspection (e.g. in an summarized security specification of application API calls), DroidSafe includes a mechanism to convert the generated finite automata to regular expressions. The implementation of the conversion from automata to regular expressions uses Brzozowski's [20] algebraic method to perform the conversion.

Moreover, the conversion performs a number of abstraction steps that make the resulting regular expression more compact. For example, the '¡float¿' element of the regular expression for the URL above serves as an abbreviation for the

expanded regular expression shown below.

0—-?[1-9][0-9]*.(0—[0-9]*[1-9])(E0—-?[1-9][0-9]*)?—NaN—Infinity—-Infinity

When generating regular expressions without abbreviating these "primitive" regular expressions would often result in regular expression multiple pages long. It was observed that a compact (though sometimes approximate) regular expression is more useful than a more precise but longer representation, so the default behavior is to abbreviate regular expressions corresponding to primitive types such as float.

The method of compressing regular expressions by using abbreviations is effective for many applications, yet a number of example applications revealed major performance issues when performing the conversion. Typically, applications with poor performance contained large static strings as arguments to more complex string operations such as 'substring'. The resulting finite automata were too large for the implemented Brozowzski conversion algorithm, the space complexity of which is quadratic and time complexity cubic in the number of states of the automaton to be converted.

The principal source of state size explosion is the implementation of string operations. JSA's implementation of these operations over automata is conservative and necessarily converts any resulting non-deterministic automaton to a potentially exponentially-bigger deterministic automaton. Removing this conversion would potentially avoid the performance issues of pathological applications, but this would require major internal changes to JSA.

As an alternative to substantially modifying JSA, DroidSafe includes an additional (and by default, primary) method for conversion to regular automata, by converting directly from the approximate regular grammar to a regular expression. With this approach, a regular expression is generated for each non-terminal grammar production bypassing the conversion to multi-level finite automata and (in turn) DFA. The method for doing this conversion is

straightforward, utilizing Arden's lemma [20] for recursive productions (similar to Brozowzski's method used to convert DFA to regular expressions).

The direct conversion avoids eagerly applying string operations. Moreover, some string operations can be implemented to preserve more of the original regular expression structure. For example, when applying an operation to the union of two languages, I can in some instances distribute the operation through the regular expression union operator. This is in contrast to the convert-to-DFA-and-apply-operation approach, which will first construct a DFA for the union of the two languages, and then apply the operation to that DFA. Doing so can yield less-comprehensible regular expressions.

The downside of converting directly from regular grammar to regular expressions is that it is not immediately obvious what the effect of a string operation should be on a regular expression besides as an operation on the automaton accepting the language of the regular expression. The current Droidsafe implementation takes a conservative approach when such operations are encountered, by reverting to the regular-expression-from-DFA approach for those instances.

Better Hotspot identification JSA performs program analysis given a set of hotspots - string values for which a regular approximation is to be calculated. However, the cost of the analysis is determined in part by the number of hotspots being analyzed. To reduce the runtime of JSA, I used the results of the Points-to analysis to collect call targets for each virtual call instead of marking every possible target a hotspot.

Performance Improvements Modifications to JSA have also been made to improve its performance. The modifications include factoring the generated ¡wrapper¿ method to reduce complexity and changing some data structures in alias analysis to improve efficiency, and changing hotspot calculation to only iterate the modeled application code once.

3.2 Points-to Analysis

A points-to analysis creates an over-approximation of all the heap values that can possibly flow into each reference by tracing data flow through assignments.

Pointers include program variables and also pointers within heap-allocated objects, e.g., instance fields. The result of the analysis is a points-to relation η, with $\eta(p)$ representing the points-to set of a pointer p. For decidability and scalability, points-to analyses must employ abstraction to finitize the possibly-infinite set of pointers and heap locations arising at runtime. In particular, a heap abstraction represents dynamic heap locations with a finite set of abstract locations.

3.2.1 Context-Sensitivity

A context-sensitive points-to analysis separately analyzes a method m for each calling context that arises at call sites of m. A calling context (or, simply, a context) is some abstraction of the program states that may arise at a call site. Separately analyzing a method for each context removes imprecision due to conflation of analysis results across its invocations. For example, consider the program in Listing 3.2.

```
1  id(p) { return p; }
2  x = new Object(); // o1
3  y = new Object(); // o2
4  a = id(x);
5  b = id(y);
```

Listing 3.2: Context Sensitivity Example

A context-insensitive analysis conflates the effects of all calls to *id*, in effect assuming that either object $o1$ or $o2$ may be passed as the parameter at the calls on lines 4 and 5. This assumption leads to the imprecise conclusions that a may point to $o2$ and b to $o1$. Now, consider a context-sensitive points-to analysis that uses a distinct context for each method call site. This analysis will process *id* separately for its two call sites, thereby precisely concluding that a may only point to $o1$ and b only to $o2$.

3.2.2 Object-Sensitivity

Rather than distinguishing a method's invocations based on call strings, an object-sensitive analysis [16] uses the (abstract) objects passed as the receiver argument to the method. The intuition behind object sensitivity is that in typical object-oriented design, the state of an object is accessed or mutated via its instance methods (e.g., "setter" and "getter" methods for instance fields). Hence, by using receiver objects to distinguish contexts, an object-sensitive analysis can avoid conflation of operations performed on distinct objects.

In general, the precision of an object-sensitive analysis is incomparable to that of a call-string-sensitive analysis[16]. Object sensitivity can lose precision compared to call-string sensitivity by merging across call sites that pass the same receiver object, but it may gain precision by using multiple contexts at a single call site (when multiple receiver objects are possible)

3.2.3 Choosing a Sensitivity

The inclusion of an interface called the 'PTABridge' made experimentation with various sensitivities easy. All PTA client analyses, including Value Analysis, relied on this interface to query the heap abstraction and therefore switching between context-sensitivity and object sensitivity was as easy as flipping a switch.

The search for the right sensitivity for my points-to analysis to use was all in an effort to increase the precision of all client analyses. My primary benchmark was the precision statistics of Value Analysis. When run with no sensitivity at all, only 80.82% of the fields I wished to resolve were resolved unambiguously, as seen in Table 4.4.1. When run with context-sensitivity, 91.84% of the fields got resolved unambiguously, as seen in Table 4.4.1. When run with object sensitivity, 97.53% of the fields got resolved unambiguously, as seen in subsection 4.4.1. The precision offered by object-sensitivity was hence strictly greater than using no sensitivity or 1CFA and that is the sensitivity I chose.

3.3 Summary

In this chapter I discussed two supporting analyses of Value Analysis - string analysis and point-to analysis. I described how the DroidSafe system employs a modified version of an off-the-shelf string analysis to resolve a variety of complex string manipulations instead of only being able to handle string constants and give a comprehensive overview of the kinds of sensitivities I experimented with before finding a combination that gave us the most precision in the context of Android. In particular, I presented the various context sensitivities I tried and why I settled on selective object sensitivity and no context sensitivity.

Chapter 4

Value Analysis

Value Analysis is the core contribution of this thesis - a context-sensitive, inter-procedural analysis that I use to resolve values of important fields of Android API objects.

Because I am dealing with potentially large applications with many possible entry points and inputs in the context of a huge API, it is clear that no direct analysis would scale to the set of all possible executions of an app in all possible execution environments. Analyzing each concrete semantics, a single execution of an application in a single execution environment, would be computationally prohibitive. Therefore I turn to abstract interpretation, the theory of soundly approximating the semantics of programs, to help us efficiently resolve these values.

4.1 Abstract Interpretation

Abstract interpretation is a mathematical theory of semantics approximation developed by Patrick and Radhia Cousot about 30 years ago [7]. It allows an analysis to answer questions which do not need full knowledge of program executions or which can tolerate a correct, but imprecise answer. My goal of resolving values of fields is such a question - I can tolerate not knowing the values of a few fields if the majority are resolved.

This is achieved by allowing the analysis designer to build new semantics through

abstraction of existing ones. In my case, I begin with the semantics of Jimple, a typed 3-address intermediate representation of Java [22]. Jimple provides an alternative to working directly with Java bytecode, but it's semantics are largely the same and thus too complex for our analysis to handle directly efficiently. Therefore I applied the abstractions describe in the next few sections to achieve an abstract semantics that strikes a good balance between begin efficient and being complete.

4.1.1 Abstract State

Before presenting the abstract semantics, I define the abstract state of the program.

Primitive Field Abstraction

The high-level idea behind our primitive field abstraction is that I want to use one abstract value to represent all the values that a primitive field of an object may ever take on. Abstract interpretation mandates that these values form a lattice. In order to construct a lattice from the abstract values, I will use a powerset construction, so our abstract domain will correspond to subsets of all the possible values of each primitive type that I wish the analysis to support - *int*, *long*, *float*, *double*, and *String*. $\top$ will be equal to "ANYTHING", $\bot$ will be the empty set, and the partial order will be determined by the subset relation. I will use to $\mathcal{P}$ to refer to the powerset lattice, constructing one for each type - $\mathcal{P}(int)$, $\mathcal{P}(long)$, $\mathcal{P}(float)$, $\mathcal{P}(double)$, $\mathcal{P}(String)$.

The abstract state of the program that our analysis updates thus consists of 5 maps, one for each of the primitives that our analysis supports. Each one maps an allocation site and field name to a lattice element of the corresponding primitive type abstract value:

$$\sigma_i : (Alloc \times Field) \rightarrow \rho(int) \tag{4.1}$$

$$\sigma_d : (Alloc \times Field) \rightarrow \rho(double) \tag{4.2}$$

$$\sigma_l : (Alloc \times Field) \rightarrow \rho(long) \tag{4.3}$$

$$\sigma_f : (Alloc \times Field) \rightarrow \rho(float) \tag{4.4}$$

$$\sigma_S : (Alloc \times Field) \rightarrow \rho(String) \tag{4.5}$$

4.1.2 Heap Abstraction

The above abstract state only tracks the collective values of primitives. I rely on the Points-to Analysis to provide us with an abstract domain that can be used to reason about the heap. The heap memory locations are abstracted by sets of allocation sites, denoted $\mathcal{S}(Alloc)$. All the abstract values of elements in an array are joined together. The resulting heap abstraction can be summarized by the following mapping.

$$\eta : Addr \rightarrow \mathcal{S}(Alloc) \tag{4.6}$$

$$\tag{4.7}$$

4.2 Abstract Semantics

I perform abstract interpretation on the Jimple Intermediate representation. Our abstract semantics include a rule for each primitive type I support. I ignore all constructs of Jimple except for assignment statements of primitive values to primitive type fields. I instead delegate their resolution to other analyses such as constant propagation, string analysis, and the points-to analysis. For example, assignments of reference values to reference-type fields are handled by the points-to analysis, which traces data flow through them to create an over-approximation of all the heap values that can possibly flow into each reference.

$$\frac{\eta(o) \to A \quad f \text{ is a field of type } int \quad v \text{ is of type } int}{\langle o.f := v, (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)\rangle \to (\eta, \sigma_i[\forall a \in A.(a, f) \to \{v\} \sqcup \sigma_i[a, f]], \sigma_d, \sigma_f, \sigma_l, \sigma_S)}$$

$$\frac{\eta(o) \to A \quad f \text{ is a field of type } double \quad v \text{ is of type } double}{\langle o.f := v, (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)\rangle \to (\eta, \sigma_i, \sigma_d[\forall a \in A.(a, f) \to \{v\} \sqcup \sigma_d[a, f]], \sigma_f, \sigma_l, \sigma_S)}$$

$$\frac{\eta(o) \to A \quad f \text{ is a field of type } float \quad v \text{ is of type } float}{\langle o.f := v, (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)\rangle \to (\eta, \sigma_i, \sigma_d, \sigma_f[\forall a \in A.(a, f) \to \{v\} \sqcup \sigma_f[a, f]], \sigma_l, \sigma_S)}$$

$$\frac{\eta(o) \to A \quad f \text{ is a field of type } long \quad v \text{ is of type } long}{\langle o.f := v, (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)\rangle \to (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l[\forall a \in A.(a, f) \to \{v\} \sqcup \sigma_l[a, f]], \sigma_S)}$$

$$\frac{\eta(o) \to A \quad f \text{ is a field of type } String \quad v \text{ is of type } String}{\langle o.f := v, (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)\rangle \to (\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S[\forall a \in A.(a, f) \to \{v\} \sqcup \sigma_S[a, f]])}$$

Whenever I cannot resolved the value v, I widen to $\top$.

4.2.1 Informal Proof of Soundness

A key property of the semantics designed by our abstraction is that updates to the abstract state are not destructive. If I know that a field f of object o may take on a value x, then the best thing I can do when I see an assignment $o.f = y$ is update σ_i for the allocation sites of o to $x \sqcup y$. If I cannot determine the value of y, I instead use $\top$, meaning I do not know anything about the possible values of the field. This non-destructive approach guarantees that our approximations do not compromise the truth of the results.

4.3 Implementation

As mentioned in section 4.2, Value Analysis performs abstract interpretation on Jimple. As the analysis traverses the app code and the API code (the two together are referred to as the 'modeled app'), it utilizes reflection heavily to enforce the abstract semantic rules by updating each σ in the abstract state, $(\eta, \sigma_i, \sigma_d, \sigma_f, \sigma_l, \sigma_S)$.

The abstract state is implemented as a series of container classes, one for each class whose primitive field or fields I wish to resolved. For example, Listing 4.1 is the implementation of the container class of Intent. Each primitive field [1] whose value I want to track has a corresponding field in this container. Each of these fields is a concrete representation of the abstract values I defined in section 4.1.1.

```
1  public class Intent extends RefVAModel {
2      public StringVAModel mAction = new StringVAModel();
3      public IntVAModel mFlags = new IntVAModel();
4      public StringVAModel mPackage = new StringVAModel();
5      public StringVAModel mType = new StringVAModel();
```

Listing 4.1: Intent Value Analysis Result Container Class

Whenever the analysis encounters a statement of the form $o.f = v$, it looks up the container class instance for each allocation of o that ηo returns and then uses reflection to access the field instance referred to by f.

This design enables us to separate the abstract state implementation for the core analysis as well as the modeled application code. In fact, I automatically generate the container classes, such as the one in Listing 4.1 from the API model automatically.

4.4 Evaluation

I evaluated Value Analysis on a dataset of 52 unique Android applications listed in section 4.4. This dataset was provided to us by a DARPA-sponsored third party military contractor (a "red team"). The applications range from just over 100 lines of code to over 29k lines and fall into a variety of categories, from IRC clients, to games, to cookbooks. A subset of these apps were designed to contain next-generation malware that is harder to diagnose that that which is currently out in the wild.

[1] I consider Strings primitives because I rely on our String Analysis to convert every string reference to a String constant regular expression

4.4.1 Results

I evaluated Value Analysis in its ability to determine a finite set of values that a field instance could take on, which I call unambiguous resolution. As presented in subsection 4.4.1, the analysis provided over 93% unambiguous resolution for every field I targeted and 97.53% on average. The size of this set was greater than 1 for only 8.34% of total field instances, meaning that for 91.66% of field instances, the field was empty or took on 1 value. The average set size for this 8.34% of fields with set size over 1 was just 1.58. This means that in the vast majority of cases the resolved value sets were not so large as to yield our results too imprecise to be useful. Our choice for the fields I resolved reflects our focus on using the resolved values to improve Information Flow Analysis through precise inter-component communication resolution as described in the next chapter.

For the purposes of comparing Value Analysis resolution with various choices of sensitivities, I also include the same statistics for Value Analysis using the 1-CFA heap abstraction, Table 4.4.1 and heap abstraction with no sensitivity, Table 4.4.1.

Class	Field	Unambiguous	%	Ambiguous	%	Avg Set Size	Set Size ¿ 1	%	Total
android.content.ComponentName	java.lang.String mClass	375	93.05%	28	6.95%	1.91	113	28.04%	403
android.content.Intent	android.content.ComponentName mComponent	563	100.00%	0	0.00%	1.00	0	0%	563
android.content.Intent	android.net.Uri mData	528	93.78%	35	6.22%	1.37	44	7.82%	563
android.content.Intent	java.lang.String mAction	560	99.47%	3	0.53%	1.03	9	1.60%	563
android.content.Intent	java.lang.String mType	563	100.00%	0	0.00%	1.00	0	0%	563
android.net.Uri	java.lang.String uriString	103	97.17%	3	2.83%	1.00	3	2.83%	106
android.widget.TextView	java.lang.CharSequence mText	315	97.83%	7	2.17%	2.53	88	27.33%	322
Total		3007	97.53%	76	2.47%	1.58	257	8.34%	3083

Table 4.2: Value Analysis Results (selective object sensitivity heap abstraction)[2]

[2]The number of object instances is greater when PTA is run with object sensitivity than 1CFA or no sensitivity due to class cloning.

Class	Field	Unambiguous	%	Ambiguous	%	Avg Set Size	Set Size ¿ 1	%	Total
android.content.ComponentName	java.lang.String mClass	30	37.04%	51	62.96%	1.07	52	64.20%	81
android.content.Intent	android.content.ComponentName mComponent	432	100.00%	0	0.00%	1.02	5	1.16%	432
android.content.Intent	android.net.Uri mData	394	91.20%	38	8.80%	1.62	54	12.50%	432
android.content.Intent	java.lang.String mAction	426	98.61%	6	1.39%	1.01	8	1.85%	432
android.content.Intent	java.lang.String mType	432	100.00%	0	0.00%	1.00	0	0%	432
android.net.Uri	java.lang.String uriString	82	95.35%	4	4.65%	1.00	4	4.65%	86
android.widget.TextView	java.lang.CharSequence mText	230	73.95%	81	26.05%	21.98	275	88.42%	311
Total		2026	91.84%	180	8.16%	6.16	398	18.04%	2206

Table 4.3: Value Analysis Results (1CFA heap abstraction)

Class	Field	Unambiguous	%	Ambiguous	%	Avg Set Size	Set Size ¿ 1	%	Total
android.content.ComponentName	java.lang.String mClass	26	23.64%	84	76.36%	1.19	87	79.09%	110
android.content.Intent	android.content.ComponentName mComponent	496	100.00%	0	0.00%	1.44	21	4.23%	496
android.content.Intent	android.net.Uri mData	451	90.93%	45	9.07%	4.16	73	14.72%	496
android.content.Intent	java.lang.String mAction	323	65.12%	173	34.88%	2.05	186	37.50%	496
android.content.Intent	java.lang.String mType	494	99.60%	2	0.40%	1.21	8	1.61%	496
android.net.Uri	java.lang.String uriString	0	0.00%	89	100.00%	n/a	89	100.00%	89
android.widget.TextView	java.lang.CharSequence mText	228	72.61%	86	27.39%	36.53	298	94.90%	314
Total		2018	80.82%	479	19.18%	13.01	762	30.52%	2497

Table 4.4: Value Analysis Results (insensitive heap abstraction)

4.5 Related Works

There have not been many applications of abstract interpretation in the Android development environment as of yet. Julia, however, is a static analyzer, based on abstract interpretation, that performs formally correct analyses of Android programs [21]. It has applied about a dozen static analyses, including classcast, dead code, nullness, and termination analyses, on a variety of Android applications, demonstrating its ability to find bugs, flaws, and inefficiencies. However, as far as I know the developers of Julia have not applied its abstract interpretation techniques towards the resolution of values (aside from null as part of nullness analysis) of security-relevant field of the Android API objects.

4.6 Summary

In the Value Analysis chapter I described and evaluated the main contribution of this thesis - a scalable and sound Value Analysis that determines field values of objects of security-relevant Android API classes using abstract interpretation.

App Name	LOCs	String Analysis	Points-to Analysis	Value Analysis	Infoflow Analysis	Total Runtime
AndroidGame	320	0:00:31	0:07:48	0:00:14	0:01:32	0:27:52
AndroidIRC	1253	0:00:19	0:03:55	0:00:03	0:00:14	0:13:40
AndroidMap	5587	0:00:28	0:18:02	0:00:28	0:04:08	0:44:05
AndroidsFortune	2812	0:00:12	0:15:21	0:04:45	0:21:30	0:59:39
backupHelper	2554	0:00:15	0:08:50	0:00:06	0:01:22	0:23:03
bites	2542	0:00:20	0:28:07	0:00:13	1:20:14	2:17:45
butane	1705	0:00:23	1:04:44	0:01:38	0:58:34	3:11:02
CalcA	435	0:00:02	0:04:40	0:00:03	0:02:53	0:17:44
CalcB	515	0:00:02	0:05:51	0:00:05	0:05:47	0:23:39
CalcC	431	0:00:02	0:04:49	0:00:03	0:02:55	0:17:41
CalcE	477	0:00:02	0:05:57	0:00:05	0:05:20	0:23:27
CalcF	596	0:00:26	0:11:55	0:00:30	0:09:26	0:42:03
ColorMatcher	582	0:00:13	0:24:24	0:00:05	0:32:33	1:16:46
com.p.morsecode	197	0:00:02	0:06:35	0:00:03	0:01:26	0:17:48
com.url.sourceviewer	166	0:00:09	0:11:58	0:00:19	0:06:48	0:36:22
CountdownTimer	948	0:00:16	0:17:52	0:00:08	1:21:31	1:57:15
DeviceAdmin2	1607	0:00:11	0:24:53	0:00:05	0:01:50	0:43:39
DvorakKeyboard	1036	0:00:09	0:06:02	0:00:04	0:00:38	0:16:10
Expenses	2145	0:00:23	1:12:44	0:00:15	2:06:43	3:54:06
FullControl	1565	0:00:20	0:24:21	0:00:09	1:22:52	2:16:07
InstantMessage	1153	0:01:21	0:20:08	0:18:30	0:38:29	6:00:22
InstantReplay	1945	0:00:25	0:35:13	0:00:14	1:13:02	2:13:04
KittyKitty	593	0:00:05	0:04:40	0:00:04	0:01:10	0:15:59
MediaFun	5346	0:00:15	0:06:48	0:00:07	0:00:11	0:20:48
MyDrawA	401	0:00:02	0:11:55	0:00:05	0:01:07	0:28:33
MyDrawC	412	0:00:01	0:06:09	0:00:02	0:00:17	0:14:55
MyDrawD	428	0:00:02	0:06:09	0:00:02	0:00:13	0:15:03
NetPhone	1480	0:00:13	0:26:11	0:00:14	0:04:38	0:51:30
NewsCollator	3503	0:00:12	0:15:40	0:00:44	0:21:54	1:01:26
Orienteering	6907	0:00:31	0:10:02	0:00:08	0:20:37	0:50:42
PasswordSaver	436	0:00:05	0:28:28	0:00:13	0:26:56	1:22:18
PersistentAssistant	4191	0:00:16	0:13:16	0:01:47	0:44:21	1:16:11
PicViewer	135	0:00:01	0:09:10	0:00:14	0:03:15	0:26:29
podcast	1711	0:00:18	0:29:33	0:02:58	0:32:02	1:36:57
shareloc	243	0:00:02	0:20:44	0:00:34	0:10:21	0:56:36
SmartcamWebcam	1375	0:00:11	0:38:23	0:01:02	0:18:51	1:45:35
smsbackup	293	0:00:02	0:10:05	0:00:04	0:07:51	0:28:24
SMSBlocker	2307	0:00:33	0:15:14	0:00:34	0:25:47	1:03:25
SMSReminder	2850	0:00:16	0:16:57	0:00:07	1:17:42	1:55:02
SnapshotShare	5832	0:10:36	0:30:05	0:01:15	0:10:00	6:00:21
SortingApp	1669	0:00:14	0:32:34	0:00:05	0:24:17	1:15:05
SuperNote	3225	0:00:13	0:50:01	0:00:17	2:05:47	3:28:06
SuperSudoku	1487	0:00:14	0:05:23	0:00:04	0:01:14	0:19:01
SysWatcherA	2963	0:00:25	1:03:04	0:01:16	1:54:51	3:56:00
SysWatcherB	2990	0:00:22	0:32:20	0:00:39	1:31:33	2:52:52
UdonLauncher	1279	0:00:10	1:00:13	0:00:37	0:22:19	2:00:44
UltraCoolMap	1466	0:00:18	0:12:28	0:00:28	0:07:10	0:40:51
VideoGame	1827	0:00:12	0:31:58	0:00:41	0:15:06	1:11:33
WhereMyAppsAt	2010	0:00:17	0:19:28	0:00:21	0:45:46	1:42:07
WiFinder	3294	0:00:11	0:13:44	0:00:10	0:12:45	0:41:27
WordHelper	3611	0:00:17	0:28:24	0:00:29	3:14:36	4:15:00
YARR	759	0:00:15	0:11:21	0:00:17	0:03:10	0:29:28
Total Time Spent		0:23:20	18:04:36	0:43:43	26:45:34	73:55:47
%		0.53%	24.45%	0.99%	36.20%	100.00%

Table 4.1: 52 Android App Dataset

Chapter 5

Inter-Component Communication Resolution

In this chapter I present a client component of Value Analysis, the Inter-Component Communication (ICC) Resolution Transformation. This transformation uses the results of Value Analysis to gain an understanding of how an application's components may interact and uses this information to refine the semantics of the modeled application, improving the precision of the Information Flow Analysis by 15%.

Information flow analysis tracks sensitive information guarded by API calls. The DroidSafe system classifies over 18000 API calls as injecting sensitive information into a user program. Examples of sensitive API-guarded information include the contact list, location, device id, and image information.

This increased precision manifests itself in the form of a lower number of program values that have information flow taint and a lower number of sensitive flows into sinks. Sinks are defined as API calls exposed data beyond application boundaries such as non-private file system, network, SMS, the logging framework and email.

5.1 Intent Target Resolution

An Intent is a messaging object that can be used to request an action from another app component. Although intents facilitate communication between components in

several ways, there are three fundamental use-cases:

To start an Activity An Activity represents a single screen in an app. The Intent describes the activity to start and carries any necessary data.

To start a Service A Service is a component that performs operations in the background without a user interface.

To deliver a Broadcast A Broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging.

my ICC Resolution technique focuses on intents that are used to start activities. (Future work will likely include Service and Broadcast resolution as well). Developers can specify an Intent's target component (or target components) in two ways -

Explicitly By specifying the target component using its application package and class name to instantiate a ComponentName instance and set it to the Intent's 'mComponent' field.

Implicitly By setting one or more of the Intent's action, category or data fields, and leaving the component field blank.

An example of an implicit Intent being used to start an activity can be found in the PickContact app from the introduction. In /autoreflst:ResultDisplayer.onClick the app starts a system activity that lets the user pick a piece of contact information. It does so by creating an Intent and setting the action and type field to the appropriate values, leaving the component field blank.

5.1.1 IntentFilters

Components advertise their capabilities - the kinds of intents they can respond to, through IntentFilters. Since the Android system must learn which intents a component can handle before launching the component, intent filters are specified in the

manifest as ¡intent-filter¿ elements. A component may have any number of filters, each one describing a different capability.

An intent that explicitly names a target component (explicit intent) will activate that component; the filter doesn't play a role. But an intent that doesn't specify a target by name (implicit intent) can activate a component only if it can pass through one of the component's filters.

5.1.2 Intent Target Resolution

Resolving the possible targets of intents is important for the purposes of having accurate and precise static analyses such as the information flow analysis. Intents may carry data in the form of key-value mappings or context-specific references to external resources or data. If this data is considered sensitive, then knowing precisely where it may flow may decrease the number of false-positive data leaks detected. Ignoring it, on the other hand, could lead to unsound results.

ICC Resolution Results

The Intent target resolution described in this chapter was evaluated on the same set of apps (listed in section 4.4) that I used to evaluate Value Analysis. Therefore I began already knowing that I was able to resolve the values of every field relevant to Intent target resolution unambiguously more than 90% of the time, suggesting that I should be able to resolve the targets of most Intents successfully.

As shown in section 5.1.2, my expectations were fulfilled and I was able to unambiguously resolve the targets for 96% of the 350 intents used to start activities in the 52 applications. Explicit intent targets were unambiguously resolved 100% of the time, while implicit intent targets were resolved at a lower rate of 75.44% of the time. This is due to the fact the implicit intent targeting involves more fields (action, type and data) than explicit intent targeting, and that due to my inability to disambiguate runtime context in a few cases I was not able to unambiguously resolve all involved data field URIs. If I ignore this field, then the implicit intents target resolution rate

jumps up to 100%. I include this statistic so that I can compare my results to another ICC target resolution approach, Epicc, which is not yet capable of resolving URIs. I perform this comparison in subsection 5.2.3.

Type	Unambiguous	%	Ambiguous	%	Total
Explicit	293	100%	0	0%	293
Implicit	43	75.44%	14	24.56%	57
Total	336	96%	14	5%	350
Implicit (ignoring data field)	57	100%	0	0%	57
Total (ignoring data field)	350	100%	0	0%	350

Table 5.1: ICC Resolution Results

5.2 Information Flow Analysis Improvements

As mentioned earlier, the ability to resolve the targets of Intents has a direct impact on the precision of inter-component analyses such as the Information Flow Analysis. This is because intents may carry data in the form of key-value mappings or context-specific references to external resources or data. If this data is tainted, then knowing precisely where it may flow may decrease the number of false-positive data leaks detected. Ignoring it, on the other hand, could lead to unsound results.

By using the result of Value Analysis and incorporating the results of my ICC resolution using a transformation, I managed to gain a 15% reduction in the number of program values that have information flow taint and sensitive flows into sinks that my Information Flow Analysis detects.

5.2.1 ICC Resolution Transformation Implementation

Every Activity instance includes a field called 'intent', whose value is supposed to contain another Activity instance (possibly of the same class) which started it using an Intent. The way that the Android API offers to find out this value is using the method 'public Intent getIntent()'. I developed a transformation that calls the

corresponding setter, 'public void setIntent()', once for each Activity that may have used an intent to start it using any one of the following methods:

- void startActivity(Intent)
- void startActivity(Intent, Bundle)
- void startActivityForResult(Intent, int)
- void startActivityForResult(Intent, int, Bundle)
- boolean startActivityIfNeeded(Intent, int)
- boolean startActivityIfNeeded(Intent, int, Bundle)

The first version of this transformation was extremely conservative - I called setIntent once for every Activity that the application defined, essentially assuming any Activity may start any other Activity (including itself).

I improved on this conservative approach by incorporating the results off ICC resolution. I modified the ICC Resolution Transformation to traverse the modeled app code and look for invocations of any of the 'startActivity' methods listed above. Whenever it found one, it queried the results of Points-to analysis for a set of allocation sites of Intents that the first argument may point to.

From here, the transformation continued to query the Points-to analysis results as well as the abstract state produced by Value Analysis until it has aggregated a set of Intents and categorized them as "Implicit" or "Explicit". For each set of Intents, it then applied the appropriate intent resolution rules to the field values determined by Value Analysis to come up with a more precise set of the in-app Activities that could have been started by the ICC location.

5.2.2 ICC Resolution Transformation Results

In order to evaluate the impact that the transformation described in this section had on my Information Flow Analysis, I focused on a subset of my application dataset.

my of the 52 apps, 10 had inter-component flows which were false-positives due to originally conservative ICC Resolution Transformation. Upon refining the transformation to leverage the Intent resolution results, I saw a 15% drop in both the number of program values that had information flow taint and sensitive flows into sinks that my Information Flow Analysis detected, presented in subsection 5.2.2.

Type	# of tainted program values	# of sensitive flows into sinks
Conservative ICC Resolution Transformation	1133	68
Precise ICC Resolution Transformation	961	58
Improvement	15.18%	14.71%

Table 5.2: ICC Resolution Transformation Results

5.2.3 Related Work

There have been a number of works closely related to mine -

Epicc Epicc's approach is similar in its that they attempt to determine values of variables/locations but different in that they reduce the discovery of inter-component communication to an instance of the Inter-procedural Distributive Environment (IDE) problem whereas I rely on an abstract interpretation-based value resolution analysis [19]. They applied their analysis to 1,200 applications selected from the Play store unambiguously resolved 93% of ICC locations. I cannot compare this number directly to ours because I claim to resolve the targets of 96% of all intents used to start activities as opposed to a percentage of ICC locations.

However, I can claim two improvements in my approach over Epicc's -

URI Resolution The Epicc paper says that they do not resolve the values of URIs. Consequently, they are not able to determine the values of any intent's 'data' field, which plays a vital role in implicit Intent resolution. The effect of this is an increased number of false positives for any resolved targets of the 88% implicit ICC locations that they claim to resolve unam-

biguously. I resolve all ICC-related fields and hence avoid the added risk of false positives.

Improved String Resolution The DroidSafe string analysis which is capable of resolving a number of dynamically-generated strings, whereas Epicc claims to only be able to handle constants.

ComDroid ComDroid examines Android application interaction and attempts to detect application communication vulnerabilities. It directly analyzes Dalvik bytecode instead of the source or re-targeted Java bytecode. Their analysis is not fully inter-procedural and does not leverage object sensitivity, leading to many false positives.

5.3 Summary

In this chapter I presented and evaluated a client component of Value Analysis, the ICC Resolution transformation. This transformation uses the results of Value Analysis to gain an understanding of how an application's components may interact and uses this information to refine the semantics of the modeled application, improving the precision of the information flow analysis by 15%. This increased precision manifests itself in the form of a lower number program values that had information flow taint and sensitive flows into sinks.

Chapter 6

Android API Call Clustering

6.1 Motivation

A recent study has shown that out of a sample of 1260 Android malware-infected applications, 1083 of them (or 86.0%) were repackaged versions of legitimate applications with malicious payloads [3]. Discovering these repackaged applications is not as simple as comparing source code, however, because the vast majority of these applications are obfuscated before being disseminated. The clustering approach presented in this chapter could be employed in comparing the API-usage thumbprints of applications inside of a marketplace.

6.2 Data Retrieval

In order to extract meaningful library usage patterns, I knew I would need to work with a lot of applications, which by my estimates was at least 1000. For data sets of that size, I knew that traditional static program analysis methods would be too slow and difficult to apply. Instead, I constructed a top-down hierarchical clustering method using top-level canopy clusters and bottom level Fuzzy C-Means clusters to extract API-method clusters from the collected applications.

6.2.1 Data Retrieval

The source of my raw data was the Google Play store. I crawled the Google Play websites Top Chart and New Release lists for free applications, downloading 1050 APKs of various functionality, size, and popularity. To do so I wrote a script which supplied temporary Google credentials and Device IDs to overcome the downloading restrictions imposed on Google Play store users.

6.2.2 Data Processing

In order to extract the feature vectors for each unique method in the app, I used Soot, which provides a module called Dexpler capable of converting Dalvik bytecode, which is what is found in an Android app APK, into Jimple.

Before iterating over the source code, I collected a list of possible API methods that would make up my feature space. To do so, I downloaded a jar of Version 17 of the Android API, unzipped it, and used javap, the Java Class File Dissassembler, to get a list of the public methods defined in each class. The total number of API methods came out to be 37,913. I wrote this list to a separate file, prepending the containing class to each method signature to make sure each one was unique. The line number of each method was my id for it from here on out.

Next I traversed the Jimple code of each application in Java, writing out a list of API call ids for each method to a CSV file, skipping methods that I suspected came from an included 3rd-party library. Each line in this file thus ended up being the feature vector for a method found in the application. In order to be able to tell which methods got clustered together at the end, I also wrote out the name, containing class, and containing application name of each method out to the corresponding line number in another CSV file.

6.2.3 Data Representation

Before clustering the data, I needed to translate the feature vectors of each method into a numeric vector that Apache Mahout would understand. I used the N-grams

data representation with the term-frequency (TF) weight metrics. I interpreted application methods as documents, sequences of API calls as text, and individual API calls as words. This broke sequences of API calls into overlapping N-API call sequences. Each possible permutation of N API call sequences was given a position in the vector. The value at each position was determined by the TF weight metric. I ended up using 1-grams (unigrams) for the clustering runs. A unigram essentially models the presence and frequency, or API call mix, of API calls in a function. Any higher dimensional n-grams put too much stress on the system and could not be used with the servers I had available to us.

6.2.4 Data Characterization

The resulting dataset was challenging to analyse for a few reasons. The feature space is very large; there are 37,913 possible API calls. High dimensionality data is difficult to cluster because enumerating all the possible values becomes increasingly expensive. In my case, each method summary consists of, on average, 3-7 API calls. Since the feature space is large but the actual number of features used in each data point is small, I can characterize the data as sparse. The number of methods I wish to cluster is also very large - 390,709 method summaries from 1050 android apps.

6.3 Clustering

Given that there is no precedent for how the API calls may be grouped together, I had to use an unsupervised algorithm. I settled on clustering because it made sense to think of application methods as being close together in the space of all API call patterns. I also chose clustering algorithms which would allow API calls to belong to multiple clusters, since a single API method could be a part of multiple macro-operations.

I constructed a clustering pipeline, where I implemented a top-down clustering method. Top Down Clustering is a hierarchical method that first cheaply partitions the data into overlapping subsets (canopies), and then performs more expensive clus-

tering within each canopy. I used canopy coarse clustering for the former, and fine grained Fuzzy C-Means clustering for the latter.

To build the clustering pipeline I used Mahout, Apache's open-source scalable machine learning library. Mahout is built on top of Hadoop, a distributed, large-scale data processing platform. I reserved several Amazon EC2 Hadoop nodes to process the data and synchronize my local filesystem with Amazons S3 filesystem.

6.3.1 Canopy coarse clustering

I chose the fast and coarse canopy clustering algorithm as the top-level method. Canopy clustering methods scale well with dataset size, at the cost of precision. A canopy is a subset of points that are within some distance of a central point, where every point must belong to one or more canopies. I constrain a canopy layout by defining two distance thresholds, t1 and t2. T1 ensures all elements are under a canopy, and t2 prevents the creation of multiple canopies. These distance thresholds are highly dependent on the dataset, so I evaluated the cluster quality metrics for various parameter settings in order to find acceptable t1 and t2 values.

To compute the distance between the unigram TF vectors I used the Cosine Distance Metric. This metric measures the angular difference between two vectors and produces a normalized distance value. It is commonly used to analyze the similarity between term-frequency vectors.

When running canopy clustering, the number of desired clusters needs to be specified. I chose to look for sqrt(n/2) clusters, where n is the number of application methods in the data set. This is a heuristic for estimating the number of clusters in absence of domain information.

6.3.2 Fuzzy C-Means Clustering Algorithm

For the fine-grain clustering algorithm I used the more rigorous Fuzzy C-Means. This algorithm minimizes the distance between the chosen cluster centers and each point's cluster membership coefficient. The outcome are microclusters which consist of a

center, the set of clustered points, and the cluster membership weights for the points. As in Canopy clustering, I looked for sqrt(n/2) clusters, where n is the number of application methods in the macrocluster.

6.4 Results

6.4.1 Quantitative Results

Cluster quality

Looking at the results obtained for the permutations of t1 and t2 in Figure 6-1, I can infer that the distance between macro-clusters is very large and the inter-cluster density is low. I chose the permutation of t1 and t2 that yielded a large Composed Distance Between and Within Clusters (Cdbw) separation metric and a low cluster count and average size. This is because I wanted clusters to have a high Cdbw (which means they are more compact and more separated) and to be relatively small so I could analyze them better qualitatively.

t_1	t_2	# Clusters	# Small Clusters	Avg Size	Max Size	Cdbw-Intra	Cdbw-Inter	Separation
0.7	0.5	1,724	872	107	2,036	132.01	0.127	$2.47 \cdot 10^6$
0.7	0.6	3,577	1085	431	11,158	160.186	0.1744	$9.93 \cdot 10^6$
0.9	0.4	1,845	364	3,305	20,517	337.58	0.126	$2.07 \cdot 10^6$
0.9	0.5	1,390	319	2,068	13,396	283.269	0.040	$1.32 \cdot 10^6$

Figure 6-1: Canopy Cluster Properties for Varying t1, t2. *Cdbw-Intra*: Cdbw Intra-Cluster Density, *Cdbw-Inter*: Cdbw Inter-Cluster Density, *Separation*: Cdbw Separation.

I further validate the effectiveness of the clustering by observing in Figure 6-2 that the intra-cluster density (density of a cluster) is higher than the inter-cluster density (density of overlapping points between clusters).

API Trends

I plotted the number of API calls per micro-cluster in Figure 6-3. The Figure shows that most microclusters, or macro-operations, consist of 1-3 API calls. It also shows

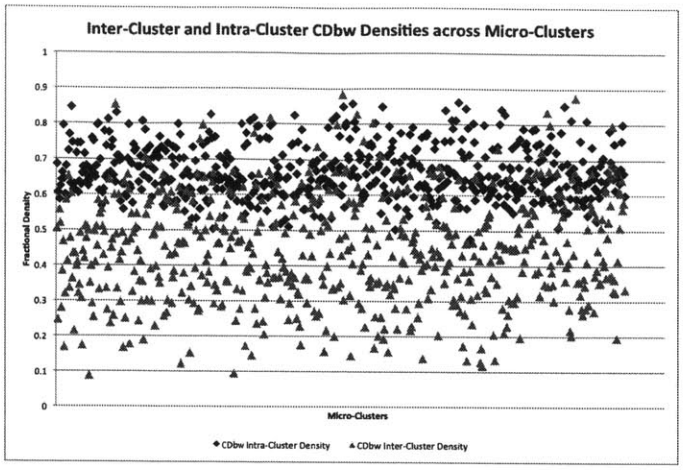

Figure 6-2: Inter-Cluster and Intra-Cluster Densities across Micro-Cluster

that there is a negative correlation between the number of API calls in a cluster and the number of those clusters. I believe that the number macro-operations with 19-20 API calls is relatively high because these macro-operations represent functions that attempt to do more than one thing and should ideally be broken down into smaller methods.

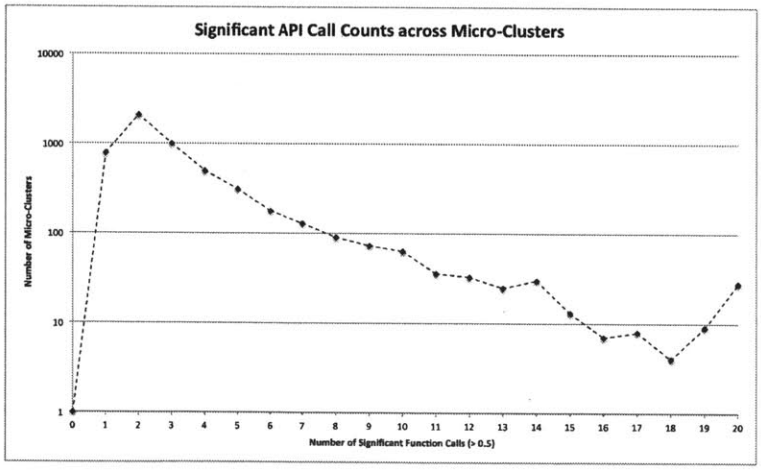

Figure 6-3: Significant API Call Counts across Micro-Clusters

Function Trends

I plotted the number of application methods per macro-cluster in Figure 6-4. The figure shows that most of the macro-operations are associated with 1-10 application methods. The number of macro-operations drops exponentially with respect to the number of functions. From the figure I can also infer that the macro-operations with large numbers of associated clusters are most likely bad micro-operations since many of the functions are not strongly correlated to the clusters. Finally, the figure also shows that no clusters have more than 300 strongly correlated functions.

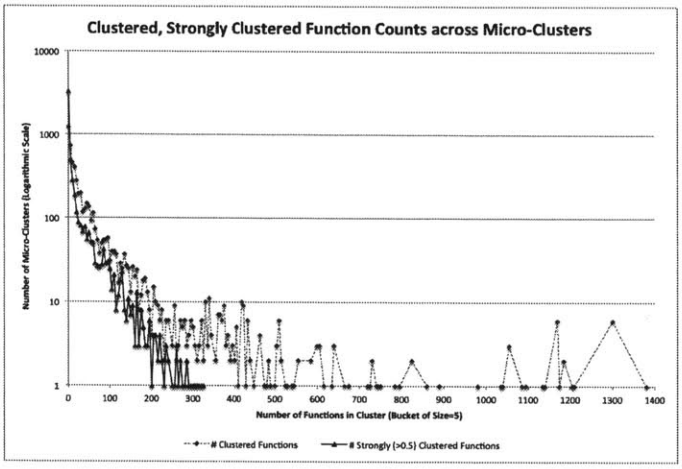

Figure 6-4: Clustered, Strongly Clustered Function Counts across Micro-Clusters

6.4.2 Qualitative Results

I analyzed the results qualitatively by identifying the clusters with five or more highly correlated calls. I labeled their macro-operation based on their API call mix and verified their macro-operations validity by analyzing the clustered application methods. Many macro-operations were instantly identifiable just from glancing at the API call mix. Furthermore, they were found in functions across different apps, which suggests they are true macro-operations.

Consider two case studies that have been verified against the function source code:

1. **Create a Rectangle with Rounded Corners:** Based on the instruction

mix, I infer this macro-operation creates a bitmap with rounded, transparent corners. Though this macro-operation is obscure, it appears across a wide range of - including dictionaries, contact managers and SMS clients. I traced the similar code back to a single "stackoverflow.com" thread. This case study suggests that macro-operation clusters can be used for cursory duplicate code detection and identifying commonly used, but esoteric procedures.

```
1.04: init(Bitmap)
0.87: setXfermode(Xfermode)
0.84: setColor(int)
0.83: init(PorterDuff$Mode)
0.82: init(int,int,int,int)
0.79: setAntiAlias(boolean)
0.79: drawRoundRect(Rect,float,float,Paint)
0.79: drawARGB(int,int,int,int)
0.76: drawBitmap(Bitmap,Rect,Rect,Paint)
0.73: init(Rect)
```

Figure 6-5: Example of application methods which belong to Rounded Corners Cluster

2. **WebGL Image-to-Texture Binding:** Based on the instruction mix, I infer this macro-operation binds an image to a WebGL texture. This macro-operation has been found across multiple classes of apps - including art galleries, file managers, photo managers, and special effects apps. WebGL procedures such as this one are typically very verbose and require the user execute long sequences of API calls. For cases where the macro-operation is a predictable set of calls, I can use the macro-operation to assist programmers. as the programmer is typing I can predict the macro-operation being used and provide API call recommendations to assist programmers with their coding task.

6.5 Summary

In this chapter I demonstrated that it is possible to extract common Android API usage patterns out of a large set of Android applications. Overall, I extracted more than 1,000 APKs from Google Play, processed the data, and ran Mahout's clustering

```
2.26: glTexParameterf(int,int,float)
1.10: glBindTexture(int,int)
0.98: glGenTextures(int,int[],int)
0.84: texImage2D(int,int,Bitmap,int)
0.77: recycle()
0.67: glTexParameteri(int,int,int)
0.48: glTexParameteriv(int,int,int[],int)
0.31: glTexParameterx(int,int,int)
0.21: requestRender()
```

Figure 6-6: Example of application methods which belong to WebGL Cluster

algorithms to group the data into macro-operations. I also manually analyzed the resulting clusters and were able to label several of the macro-operations.

Chapter 7

Conclusion

In this thesis I presented a design for an efficient and sound abstract interpretation-based Value Analysis which calculates the values of important fields of objects of security-sensitive Android API classes. The analysis is an important component of DroidSafe, an Android malware detection system designed to prove important properties of sensitive program behaviors before the programs appear in an application marketplace. The resolved program values provide important context for other DroidSafe analyses and the generated application summary, improving their precision. This in turn helps a trusted analyst avoid false positives and determine whether a particular application is malicious in a shorter amount of time.

Appendix A

PickContact Source Code

```
1  public class PickContact extends Activity {
2      Toast mToast;
3      ResultDisplayer mPendingResult;
4
5      class ResultDisplayer implements OnClickListener {
6          String mMsg;
7          String mMimeType;
8
9          ResultDisplayer(String msg, String mimeType) {
10             mMsg = msg;
11             mMimeType = mimeType;
12         }
13
14         public void onClick(View v) {
15             Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
16             intent.setType(mMimeType);
17             mPendingResult = this;
18             startActivityForResult(intent, 1);
19         }
20     }
21
22     @Override
23     protected void onCreate(Bundle savedInstanceState) {
24         super.onCreate(savedInstanceState);
25
26         setContentView(R.layout.pick_contact);
27
28         // Watch for button clicks.
29         ((Button)findViewById(R.id.pick_contact)).setOnClickListener(
30                 new ResultDisplayer("Selected␣contact",
```

```
31                          ContactsContract.Contacts.CONTENT_ITEM_TYPE));
32          ((Button)findViewById(R.id.pick_person)).setOnClickListener(
33                  new ResultDisplayer("Selected␣person",
34                          "vnd.android.cursor.item/person"));
35          ((Button)findViewById(R.id.pick_phone)).setOnClickListener(
36                  new ResultDisplayer("Selected␣phone",
37                  ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE));
38          ((Button)findViewById(R.id.pick_address)).setOnClickListener(
39           new ResultDisplayer("Selected␣address",
40           ContactsContract.CommonDataKinds.StructuredPostal.CONTENT_ITEM_TYPE));
41      }
42
43      @Override
44      protected void onActivityResult(int requestCode, int resultCode,
45                                      Intent data) {
46          if (data != null) {
47              Uri uri = data.getData();
48
49              if (uri != null) {
50                  Cursor c = null;
51                  try {
52                      c = getContentResolver().query(uri,
53                              new String[] { BaseColumns._ID }, null, null, null);
54                      if (c != null && c.moveToFirst()) {
55                          int id = c.getInt(0);
56                          if (mToast != null) {
57                              mToast.cancel();
58                          }
59                          String txt = mPendingResult.mMsg + ":\n" + uri
60                                                           + "\nid:␣" + id;
61                          mToast = Toast.makeText(this, txt, Toast.LENGTH_LONG);
62                          mToast.show();
63                      }
64                  } finally {
65                      if (c != null) {
66                          c.close();
67                      }
68                  }
69              }
70          }
71      }
72  }
```

Bibliography

[1] 40 staffers, 2 reviews, 8500 iphone apps per week, August 2009. http://brainstormtech.blogs.fortune.cnn.com/2009/08/21/40-staffers-2-reviews-8500-iphone-apps-per-week/.

[2] Mobile threat report. Technical report, F-Secure Labs, 2013. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf.

[3] AppBrain. Number of available android applications, January 2014. http://www.appbrain.com/stats/number-of-android-apps.

[4] INC Apple. Apples app store downloads top three billion. Press Release, January 2010. http://www.apple.com/pr/library/2010/01/05Apples-App-Store-Downloads-Top-Three-Billion.html.

[5] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from http://www.brics.dk/JSA/.

[6] comScore MobiLens. Top smartphone platforms. Press Release, October 2013. http://www.comscore.com/Insights/Press_Releases/2013/12/comScore_Reports_October_2013_US_Smartphone_Subscriber_Market_Share.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[8] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[9] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[10] Asger Feldthaus and Anders Møller. *The Big Manual for the Java String Analyzer*. Department of Computer Science, Aarhus University, November 2009. Available from `http://www.brics.dk/JSA/`.

[11] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] Google. Jelly bean. Documentation. `http://developer.android.com/about/versions/jelly-bean.html#android-42`.

[14] X. Jiang. An evaluation of the application ("app") verification service in android 4.2. Dec 2012. `http://www.cs.ncsu.edu/faculty/jiang/appverify/`.

[15] Hiroshi Lockheimer. Android and security. Blog Post, February 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[16] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[17] Dan Moren. Retrievable iphone numbers mean potential privacy issues. Article, September 2009. `http://www.macworld.com/article/1143047/phone_hole.html`.

[18] Nielsen. State of the appnation - a year of change and growth in u.s. smartphones. Press Release, May 2012. `http://www.nielsen.com/us/en/newswire/2012/state-of-the-appnation-%C3%A2%C2%80%C2%93-a-year-of-change-and-growth-in-u-s-smartphones.html`.

[19] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[20] Jacques Sakarovitch. The language, the expression, and the (small) automaton. In *CIAA*, pages 15–30, 2005.

[21] tienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192 – 1201, 2012.

[22] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[23] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 559–572, Berkeley, CA, USA, 2013. USENIX Association.

[24] N. Wingfield. 'IPhone Software Sales Take Off: Apple's Jobs'. *Wall Street Journal*, Aug 2008. `http://online.wsj.com/news/articles/SB121842341491928977`.

[25] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.

[26] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109. IEEE Computer Society, 2012.