



AFRL-RI-RS-TR-2018-159

AUTOMATED ANDROID APPLICATION PARTITIONING (A3P)

RAYTHEON BBN TECHNOLOGIES

JUNE 2018

INTERIM TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2018-159 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

SCOTT F. ADAMS
Work Unit Manager

/ S /

WARREN H. DEBANY, JR
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) JUN 2018		2. REPORT TYPE INTERIM TECHNICAL REPORT		3. DATES COVERED (From - To) AUG 2016 – JUL 2017	
4. TITLE AND SUBTITLE AUTOMATED ANDROID APPLICATION PARTITIONING (A3P)				5a. CONTRACT NUMBER FA8750-15-C-0108	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) Alex Jordan, Michelle Spina, Benjamin Clough, Karen Sirois, Maria Zakhalyavko				5d. PROJECT NUMBER STAC	
				5e. TASK NUMBER BB	
				5f. WORK UNIT NUMBER NT	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Raytheon BBN Technologies Corp. 10 Moulton St. Cambridge MA 02138				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGB 525 Brooks Road Rome NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-159	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. PA# 88ABW-2018-2931 Date Cleared: 11 JUN 2018					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Mobile applications have become ubiquitous and a critical part of many peoples' every-day life. However, mobile operating systems provide only coarse control over how private data is accessed, transformed, or used, and users have remarkably little understanding of how this can affect their privacy. Complicating matters further are advertising libraries or other third-party software that require odd permissions in order to satisfy their particular needs, but are tangential to the main application. To address these issues, we present a system for analyzing an Android application's control flow, information flow, and security profile to partition it into a set of cooperating reduced-privilege micro-apps that reproduce the original application's functionality. Our experiments on real-world applications indicate that this approach provides significant value in reducing the security requirements of an individual micro-app, and that the original application functionality can be reproduced by the collection of micro-apps.					
15. SUBJECT TERMS Android, computer security, distributed computing, mobile applications, permission, reasoning about programs, software reusability					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 34	19a. NAME OF RESPONSIBLE PERSON SCOTT F. ADAMS
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code)

TABLE OF CONTENTS

1.	Summary	1
2.	Introduction.....	3
2.1	Related Work	3
2.2	Background.....	4
3.	Methodology, Assumptions, and Procedures.....	6
3.1	Assumptions.....	6
3.2	Application Partitioning.....	6
3.3	Experimental Evaluation.....	12
4.	Results and Discussion	15
4.1	IPC Test Applications	15
4.2	Realistic Applications	16
4.3	Google Play Store Applications.....	20
5.	Concluding Remarks.....	25
5.1	Potential Future Work.....	25
6.	References.....	26
	Glossary/Acronyms.....	28

LIST OF FIGURES

Figure 1 - A3P takes an Android application binary and produces a collection of cooperating micro-apps with limited permissions that replicate the original application functionality; APK - Android Application Package, Spec – specification, RPC- Remote Procedure Call.....	1
Figure 2 - Partitioning occurs in three passes, each of which handle a particular type of class call graph node.....	9
Figure 3. Micro-app Packaging Process	10
Figure 4 - A direct method call is transformed into an AIDL invocation when the call crosses micro-app boundaries.....	12
Figure 5 - CallBlocker is split into nine partitions, most of which contain a single class requiring a dangerous permission.....	18
Figure 6 - CryptoPic is partitioned into two micro-apps with disjoint permissions	19
Figure 7 - MyLibrary is split into three partitions; two have dangerous permissions, and the third manages the database layer	20
Figure 8 - Play Store Apps are partitioned based on required permissions and use of the application sandbox	21

LIST OF TABLES

Table 1 - A3P Assumptions	6
Table 2 - IPC Test Applications.....	13
Table 3 - Realistic Test Applications.....	14
Table 4 - Google Play Store Applications	14
Table 5 - CallBlocker Partitioning Results	17
Table 6 - CryptoPic Partitioning Results	18
Table 7 - MyLibrary Partitioning Results	19
Table 8 - ColorPicker Partitioning Results	20
Table 9 - Google Play Store Application Partitioning Results	21

1. SUMMARY

This report resulted from research sponsored by the Defense Advanced Research Projects Agency (DARPA) for the Space/Time Analysis for Cybersecurity (STAC) program, under Air Force Research Laboratory (AFRL) agreement number FA8750-15-C-0108. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Mobile applications (apps) have become ubiquitous and a critical part of many peoples' everyday life. However, mobile operating systems provide only coarse control over how private data is accessed, transformed, or used, and users have remarkably little understanding of how this can affect their privacy. Complicating matters further are advertising libraries or other third-party software that require odd permissions in order to satisfy their particular needs, but are tangential to the main application. To address these issues, we present Automated Android Application Partitioning (A3P), a system for analyzing a binary Android application's control flow, information flow, and security profile to partition it into a set of cooperating reduced-privilege micro-apps that reproduce the original application's functionality. This is performed on the application binary, and does not require assistance from the application developer. Our experiments on real-world applications indicate that this approach provides significant value in reducing the security requirements of an individual micro-app, and that the original application functionality can be reproduced by the collection of micro-apps. While this effort focuses on Android applications, the technique is generalizable to Windows, OSX, and GNU/Linux programs that use privilege bracketing to minimize potential damage from an exploit.

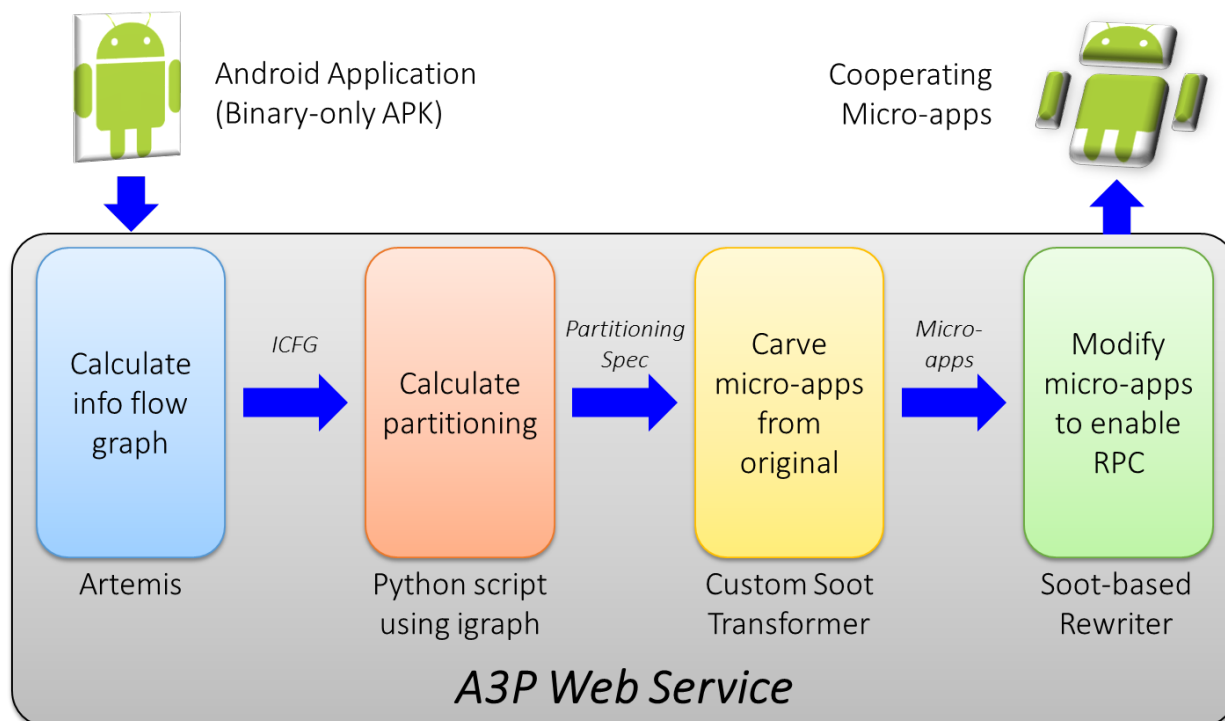


Figure 1 - A3P takes an Android application binary and produces a collection of cooperating micro-apps with limited permissions that replicate the original application functionality; APK - Android Application Package, Spec – specification, RPC- Remote Procedure Call

The A3P system uses four major processes to accomplish its goals, summarized in Figure 1. An Android application binary is provided as an Android Application Package (APK) and analyzed using a tool that statically recovers the application control flow and information flow, identifies security-relevant features of the application, and creates an Information and Control Flow Graph (ICFG) to hold the data. The next stage analyzes the graph using a set of heuristics to determine a partitioning that minimizes the permissions required by each partition. The third stage carves the original application along the lines suggested by the partitioner into a set of micro-apps. The fourth stage performs a fixup pass that modifies the micro-apps to make method calls across partitions into remote procedure calls.

In order to evaluate the partitioning algorithm, we wrote a set of applications that employ a variety of Android mechanisms for invoking other application components. We also wrote a set of realistic test applications with known ground-truth and collected a small set of apps from the Google Play Store. The A3P system was able to successfully calculate a partitioning for all of the applications on which we tested it. For 19 of the 28 applications we tested, we were able to create two or more micro-apps that duplicated the original applications' functionality. For the 9 applications that failed, all of the failures are reducible to two missing features of the system that were out of scope for our effort.

There are a variety of potential avenues for future research; the most useful would be to modify the system to handle the two failure modes that were observed in the A3P effort. There is also interesting work to be done in hardening the generated micro-apps and in leveraging the partitioning capability to introduce new security controls. In addition, while the specific tactics we used were specific to Android, the concepts are generalizable to other computing environments including the iOS operating system and desktop applications.

2. INTRODUCTION

The Android application marketplace is filled with applications that require seemingly odd combinations of permissions: flashlights that access your contacts [1], games that require your location [2], and messaging apps that get a list of other apps on your phone [3]. Frequently, these required permissions are due to external resources like advertising libraries that use personal information to tailor ads to users. Sometimes, they are due to an app's need to process multiple types of data. Occasionally, they are due to malicious code that has been inserted into otherwise benign applications. Regardless of why the application needs them, the mix of required permissions introduces significant risk to the user, her device, and her data. The fact that users cannot control *how* an application uses the data to which it has access only compounds the issue.

To address this problem, we have developed an automated process for dividing an application into multiple cooperating *micro-apps* that replicate the original application's behavior while maximizing a given separation metric. The initial metric we implemented was privilege separation. The metric is improved when permission per application is minimized. This privilege separation can prevent malicious code or exploited vulnerabilities in the resulting applications from gaining access to more information than the host code normally requires. Each micro-app is modified to replace calls across separation boundaries with remote procedure calls using standard Android inter-process communication (IPC) mechanisms.

While this effort focuses on Android applications, the technique is generalizable to Windows, OSX, and GNU/Linux programs that use privilege bracketing to minimize potential damage from an exploit. By focusing on Android we begin with a platform of interest to which we can apply existing program analysis tools. We also avoid some of the difficulties of disassembly and semantic recovery that are present in other platforms, such as x86 or ARM binaries.

2.1 Related Work

Program partitioning (also known as “slicing”) as a distinct technique was first described, to our knowledge, in a 1984 paper by Mark Weiser [4] as a method for decomposing a program into subprograms by preserving only those statements that affected a particular output of the program. Slicing was initially proposed as a way to aid in debugging of applications, but has been extended to aid in program differencing and integration [5], regression testing [6], automatic parallelization [7], and to cover dynamic behavior such as concurrency [8].

A significant early application of partitioning was in automatic creation of distributed programs from a traditional single-host application. Later efforts used a graph-cutting algorithm [9] that provided major inspiration for our partitioning technique.

The concept of isolating dangerous code from the host application is a natural desire that has been considered at length in the literature. AdSplit [10] appears to be the earliest approach for Android. It recognizes specific preselected advertising libraries, replaces their implementation in the host app with a remoting stub, runs the ad library in its own process, and connects them together through standard Android IPC mechanisms. Later work, including AFrame [11] and FlexDroid [12] rely on developer effort and modifications to the Android OS (respectively), but accomplish largely the same capabilities. These approaches all have required *a priori* identification and characterization of libraries to be extracted. Other approaches such as NativeGuard [13] are capable of creating remote procedure call interfaces, but are limited to pre-existing native libraries with known interfaces.

Cloud offload is another related problem domain from the mobile computing literature. Efforts in this area include CloneCloud [14] and others that attempt to automatically identify portions of a mobile application that can be offloaded to cloud computing resources to make mobile applications more performant in terms of power utilization [15] or execution speed [16], or to make use of specialized computing resources [17]. The approaches described in the literature focus on specific computing tasks and tend to use heuristics or dynamic code analysis for deciding which code should be offloaded. This offload capability is a special case of the more general program separation problem we address in this work.

In contrast, a general purpose partitioning capability needs to be able to determine partitions to separate arbitrary components of the program given a variety of separation metrics. A3P uses static code analysis techniques to find optimal cut points for partitioning the target application into distinct functional areas based on the kinds of data each area processes, without any *a priori* knowledge of the application. These cut points maximize a separation metric for application code sharing or data sharing across components. We apply information flow analysis techniques to analyze and partition a fully packaged binary application and implement a proof-of-concept automated system that works with applications selected from a variety of test corpora and real-world sources.

2.2 Background

2.2.1 Android application model

By default, each executing Android application runs in its own Dalvik virtual machine instance, running as a per-process user on a heavily modified form of Linux. Android applications do not have a single entry point as in most traditional applications. This is an important detail, as program analysis tools designed for traditional applications normally only need to account for a single entry point that is called in predictable situations. Instead, each Android application is a set of components that the system invokes as necessary. These components include *Activities*, which present a visual user interface for one focused application function that the user can undertake; *Services*, which are like activities, but do not present a visual user interface; *Broadcast receivers*, which exist to receive and process broadcast announcements; and *Content providers*, which provide an interface to some set of structured data. These components communicate primarily through *Intents*, which are an asynchronous message that defines an action to be undertaken and data associated with that action stored in key-value pairs.

Android applications are described in the `AndroidManifest.xml` file packaged within the APK file. This manifest describes the permissions the application requests at install time, the application components present, and the mapping from intents to components that handle them. This includes the *launch activity* that is triggered when the application is launched by a user or another application.

2.2.2 Android permissions

On Android, access to sensitive system resources such as the camera or the network is mediated through a system built on top of the Linux user and group mechanism, amended with Security-Enhanced Linux (SELinux) policies to strengthen application isolation. Each Android application is installed and run under a unique user identification (ID) that is added to a set of groups depending on the permissions listed in the `AndroidManifest.xml` file. In older versions of Android, these permissions were fixed at install time after being approved by the user. As of Android 6.0 (Application Programming Interface [API] Level 23), these permissions can be

Approved for Public Release; Distribution Unlimited

requested at runtime and revoked by the user at will. While dynamic permissions are an interesting development in the Android ecosystem, they do not present any new or unique challenges to our analysis and are therefore not discussed further.

2.2.3 Android IPC

Like most operating systems, Android provides a palette of IPC mechanisms that applications can use to send data and control messages to other applications. The key IPC mechanism on Android is the *Intent*, which is an asynchronous message that can be passed within an application or between applications. Intents are parameterized by the action they are requesting, the intended receiver specified by class name or by class object, and a collection of key-value pairs called *extras*. These messages are used to start applications, launch activities, communicate with services, and perform most other data-sharing between application components.

3. METHODOLOGY, ASSUMPTIONS, AND PROCEDURES

3.1 Assumptions

The A3P effort attempted to make as few assumptions as possible in order to maintain generality of the solution. The assumptions we did make are summarized in the table below, along with the rationale behind the assumption.

Table 1 - A3P Assumptions

Assumption	Rationale
Applications are provided as APK files, without access to the original application source code	The analyst may not always have access to the original application developer
The A3P user has skills of an average Android application developer	Android application development is a widely distributed skill, and if an organization needs to develop the skill, there are a huge number of training courses, books, and online courses that can assist in obtaining it.
Application developers do not make any special effort to make their applications “partitionable”	Application authors may not even be aware that A3P exists

We believe these assumptions to be reasonable, and they do not significantly limit the set of applications that we can analyze and partition.

3.2 Application Partitioning

In order to partition an application into coordinating micro-apps, we perform four major processes. We first perform a static control flow analysis and static information flow analysis in order to recover the application’s internal structure in the form of a hybrid ICFG in the style of [18]. We then apply a series of partitioning heuristics on the graph and determine partition points. Using these points, we divide the application into separate micro-apps. The final stage is to rewrite each micro-app to restore control-flow and information-flow links that cross partitions.

3.2.1 Static Program Analysis

Prior to partitioning an Android APK a static analysis is performed to gather enough information about the classes in the APK and the interactions between the classes. The information is used for both the partitioning of the APK and the repackaging of each partition into *micro-apps*. The tool used to perform the static analysis is the Artemis system.

Artemis is a program analysis system devised by Raytheon BBN Technologies to assess Android applications. Artemis extends the DroidSafe tool [19] to perform a static analysis of the information flow paths within an application. These results are saved to a database, allowing a single analysis to be queried many times in the future. The APK partitioning process uses Artemis’ query language to retrieve the data necessary for the partitioning and subsequent repackaging. This includes calculating Android permissions required per call site, Android component classes present in the app, resources used by each class, extracting the full program call graph, determining sandbox tags per class, and finding destinations for each Intent.

3.2.2 Partitioning Strategy

Applications can be partitioned based on a wide variety of metrics. For example, existing work such as NativeGuard [13] separated native code from the rest of the Android application in order to minimize the damage that could be inflicted by rogue native code. Other work such as AdSplit [10] and AdDroid [20] focused on separating third-party advertising libraries from the core application.

In the A3P effort, we chose to partition applications based on the permissions needed by each region of functionality. By separating unrelated permissions, such as access to the network and reading from the contacts database, software vulnerabilities and malicious code are limited in the information to which they have access and what they can do with what information they do have. It also gives us the ability to introduce filters or controls on the data that is passed between micro-apps, although this is not addressed in our current effort.

More formally, the goal of our partitioning process is to minimize the number of Android classes that have access to each permission, while at the same time minimizing the number of partitions and subsequent micro-apps. As an example, an application that has ten classes, one of which x uses the network and one of which y uses the filesystem, will be divided into three partitions: one containing x , one containing y , and one containing the other 8 classes. In this analysis, we did not attempt to quantify the impact of adding remote procedure calls, but as a proxy for this cost gave each cross-partition method call the same penalty against the partition metric.

The first step in partitioning is to transform the program's ICFG into a graph data structure amenable to processing. We chose the Python igraph package [21], an open source library for creating and manipulating graphs, as an initial representation due to the large number of graph analysis algorithms it provides. We create a class call graph (CCG) where each node is a class implemented by the application, and a directed edge from A to B represents a call from some method in A to some other method in B. Calls within a class are not represented in this structure.

Each node is labeled with information about the class that will aid in the partitioning process, including the permissions used by the class, specific interactions with the Android application sandbox, the Android application components implemented in the class, and the application resources used by each class. Each directed edge is assigned a weight, where the weight is the number of calls made between the two classes in the direction specified.

Android application sandboxing introduces an additional partitioning constraint. Android isolates application data by providing a portion of the filesystem that is only accessible to the application itself and providing a variety of other protections on application-private data (e.g., data providers, databases, application resources, etc.). When partitioned into multiple micro-apps, this sandboxing prevents trivial data sharing between micro-apps. While there are ways on Android to create data accessible to all applications, we felt that this introduced unacceptable risk given the goal of making the application more safe, as any application on the device would be able to access the data. Given the difficulty of automatically rewriting applications to work around the sandbox, we instead chose to put all classes that require a particular kind of sandbox access within the same partition. We intend to address this shortcut in future work in this area.

We also note the use of native code. Native code is a fundamentally dangerous feature [22] in Android, as it runs as the same Linux user as the Dalvik interpreter. This allows for a wide variety of attacks, including code injection, code modification, or even a compromise of the operating system itself. To try and mitigate attacks on the Dalvik interpreter, we isolate any class using native code within its own micro-app. This limits the damage native code can do to the

core application, as it will be running as a separate user. Such partitions will also get the permission tags they require to operate. Note that this does not address the issue of attacks against the operating system.

Experimenting with the general-purpose graph partitioning algorithms included in the graph library did not prove successful, so we began a literature search for algorithms for partitioning applications. While this space has seen recent renewed interest, it is in fact a mature problem dating back to scheduling on multiple processors in the late 1970s [23]. We looked at a wide variety of work in the area and took major inspiration from graph cutting algorithms for distributed applications e.g., [9] which partition applications for optimal processing for distributed applications.

The partitioning algorithm we developed can best be described as an agglomerative partitioning scheme. We begin with the labeling described above, decorating each node in the class call graph with the permissions and sandbox features used by the methods it contains. The partitioning then occurs in three passes: sandbox features, permissions, and cloning, summarized in the figure below. When these three passes are complete, we merge all partitions that have no permissions or sandbox tags.

The first **sandbox features** pass makes a partition for each sandbox feature, and then assigns each class that uses that feature to that partition. If a class uses two sandbox features, we combine the sandbox-based partitions into a single partition. If a class does not use any sandbox features, we leave it unassigned. For each node that was assigned to a partition, we examine its outgoing edges. If the target is a leaf node not in a partition and the node does not have a permission label, we pull it into this partition. This has the effect of pulling in any classes that are “behind” the partition point.

In the **permissions** pass, we iterate over nodes that have a permission label. If the node is already in a partition, we add the required permission to the partition and continue. If the node is not in a partition, we either create a new partition for classes with that permission or add it to the existing partition for that permission. We repeat the behind check described above and add any behind nodes to the appropriate partition.

In the **cloning** pass, we iterate over all nodes that do not have a partition or sandbox label. If the node is already in a partition, we continue. If it is not, we add it to a special “others” partition that will hold all unpermissioned code. For each outgoing edge from that node, we check to see if the target is in a partition. If it is not, we add the target to the “others” partition as well. If the target node is in a partition and is a leaf node without a permission tag, we clone the target class and add the clone to the “others” partition. This cloning scenario is relatively common in computational library code, as the code is reused across the application, but does not require any

special permissions to run. By cloning the code, we eliminate the need to introduce an IPC channel and thus use extra space to pay for simplicity.

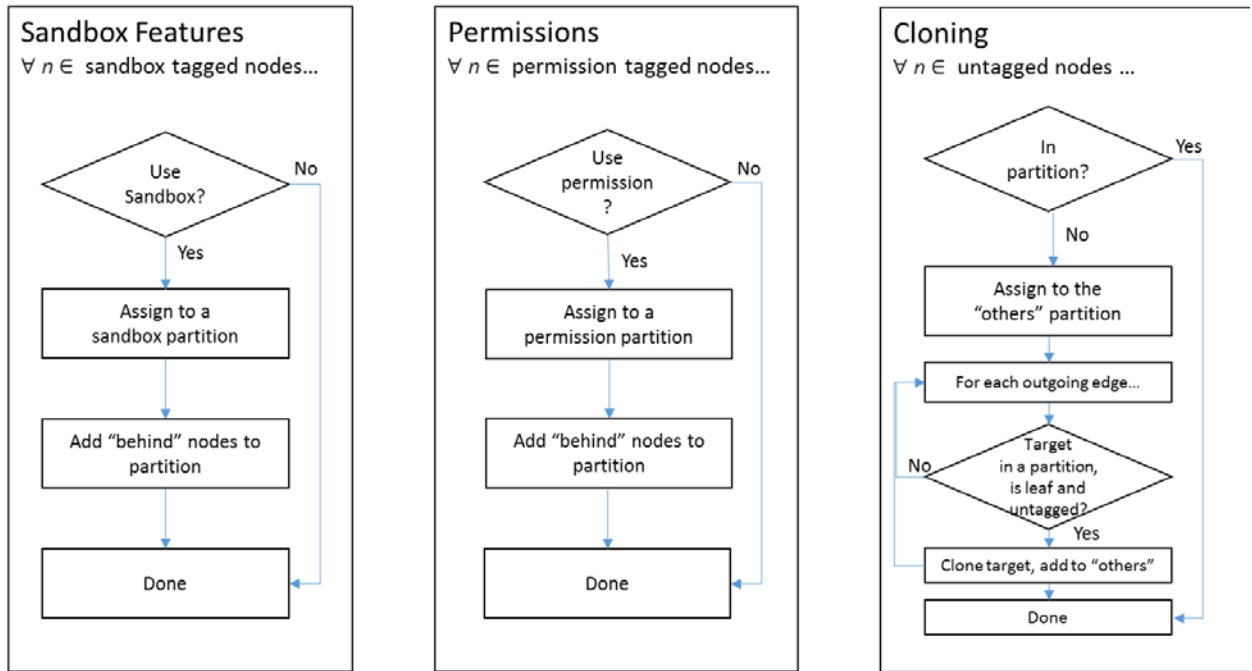


Figure 2 - Partitioning occurs in three passes, each of which handle a particular type of class call graph node.

3.2.3 Micro-app Construction

The result of the partitioning process is a partitioning strategy document that describes which classes belong in which micro-app, which original application resources are required, and what components are required in the Android application manifest file. This is passed on to the micro-app construction component of the pipeline, described in this section.

Every micro-app retains the title of the original app but receives a new unique package name as required by the Android operating system. To create manifest files, the partitioning specification may also include Extensible Markup Language (XML) chunks later added in the manifest build process and, to provide inter-app communication, may contain additional statements needed to add or modify micro-app IPC.

We automate the process of building micro-apps in a multi-stage process. First, `apktool` [24] extracts the original app so the unmodified manifest file, Dalvik executable (DEX) file, resource files and libraries can be manipulated. The partitioning specification provides the list of resource files needed by each micro-app from the original app which require no modification besides renaming some references to reflect the new app package name. We then copy over the necessary resource files as well as all libraries and assets (if any) to the respective micro-app. Note that at this stage, we could perform a more complex analysis to determine which application assets and libraries are used by the new micro-app application code, but this optimization was not a part of the A3P effort.

A second stage handles the creation of a micro-app's manifest and DEX file. We parse attributes from the original application manifest and combine them with information from the partitioning specification, such as the new app package and XML chunks, to write out a new manifest file for

each micro-app. With the partitioning of the app, support for communication between micro-apps must be added to maintain the original app's functionality. In general, this requires the addition of new intent filters to the manifest file which the XML chunks provide. However, significant modification of the application code may also be necessary in the event of classes being partitioned out. Our approach for performing this modification is described in the following section.

The original DEX file, in the simplest case, must be modified in two ways. First, the classes contained within must be filtered with the JavaScript Object Notation (JSON) acting as a whitelist such that every micro-app receives only the classes given by the chosen partitioning strategy. Second, all application classes must be re-named to reflect the new app package of the micro-app. Communication between micro-apps complicates this process somewhat as the classes themselves may require modification or the addition of new classes to provide IPC between micro-apps.

We begin by using dex2jar to covert the DEX file before extracting the individual Java class files for copying and further modification. Each micro-app receives a copy of the class files from the original app, ignoring classes which do not belong in the partition. We add classes for providing micro-app IPC in this step where necessary which we hereafter refer to as Android Interface Definition Language (AIDL) classes. These AIDL classes overwrite the class file(s) they replace in a partition and also introduce new classes to the micro-app (see below for further explanation).

Once we collect and filter the correct class files for the micro-app, the classes need to be re-named, receive new or modified statements for IPC and be written out to a new DEX file. We use Soot, a Java transformation framework that supports Android, to handle code modification [25]. This is implemented as a Transformer, which is a component of the Soot framework used to modify application code in-place.

Completion of the packaging process involves running `apktool` over the new micro-app to build it into an APK before signing it with `jarsigner`. We sign the application with the Android debug certificate for testing.

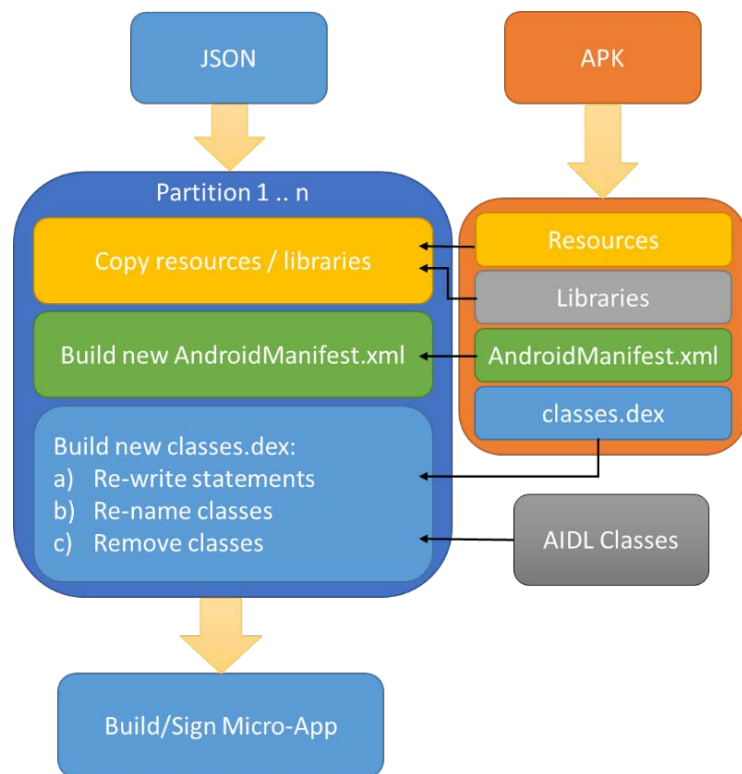


Figure 3. Micro-app Packaging Process

3.2.4 Inter-application Communication

Once the original app has been split, the resulting micro-apps still need to communicate with one another. There are two different types of communications used in the original app that must be maintained between the micro-apps: those that were Android Intents, and those that were direct method calls.

Intents can largely stay as they originally were, as they are Android's built-in method for inter-component communication and designed for inter-application communication. However, depending on the way the Intent is declared, it may need to be modified. In particular, we must modify any explicit Intent that specifies the callee using a Class object for which the class would no longer exist in the same app, or a fully-qualified class name string for which the package name has changed. There are a variety of methods that can be used to declare an Intent, and each invocation of those methods must be updated to specify the callee correctly. It is also worth noting that even when an Intent's destination remains within the micro-app, it may still be necessary to change the Intent declarations when the package name has been modified.

In order to change these explicit Intents, we first need to identify the destination of the Intent in the original app. DroidSafe implements this as part of its information flow analysis, and we reuse those results to retrieve the destination of the Intent. We then need to find the statements where the Intent is declared so that we can modify them. The identified statements can be replaced with a simple Intent initialization and a call to `Intent.setClassName()` with the updated fully-qualified class name of the destination.

Method calls between micro-apps are more difficult to handle. They are necessary because applications have rarely been designed with Intents exactly along permission boundaries. To work around this problem, we used the Android Interface Definition Language (AIDL). AIDL is an Android-provided way of defining an interface between two processes [26]. AIDL defines the interface between processes, and Android uses it to generate the code necessary to serialize objects across the application boundary. AIDL has a “server” side that provides the service, and a “client” side that uses the service. In A3P terminology, the micro-app where the functionality resides is the server, and the caller of that functionality is the client.

To use AIDL to handle inter-application method calls, we first need to identify all inter-app call sites and group them by callee class. An AIDL interface is generated for each callee class by including the declaration for each method that will be accessed from outside the micro-app. The AIDL interface is shared between micro-apps and transformed by Android's `aidl` tool into a stub library. Depending on whether the micro-app is the server or the client, the `aidl` tool generates a set of classes and interfaces that can be used for the remote invocation. We then rewrite each direct invocation of the original functionality with a call to the AIDL interface representing the remote server. The Android runtime handles all serialization, transport, and deserialization, and invocations of an AIDL interface block until the method returns.

Original invocation:

```
public class ProcessFile {  
    ...  
    String md5 = Checksum.generateMD5("abc");  
    ...  
}
```

Becomes in the caller:

```
public String generateMD5(String arg0) {  
    IChecksum iface = AidlInterfaces.mIChecksum;  
    return iface.generateMD5(arg0);  
}
```

Becomes in the callee:

```
public String generateMD5(String arg0) {  
    return Checksum.generateMD5(arg0);  
}
```

Figure 4 - A direct method call is transformed into an AIDL invocation
when the call crosses micro-app boundaries

There are a few complications. If the target of the method call is not static, we must track which instance is being referenced. To do this, an injected `RemoteService` keeps a map of IDs to callee object instances. The ID is the instance's hashcode. The mapping is populated during the constructor call and returned, so that the callee wrapper class in the caller micro-app knows it as well. The non-static AIDL method calls are given an extra parameter to specify which instance to act on. A second complication is that AIDL, by default, only handles primitive types and strings as arguments and return values. Although we have added a special case to allow Contexts, we have not solved the general case of this problem, and intend to address it in future work.

In addition to the rewriting of method invocations, we also inject a `LaunchActivity` into each micro-app. This activity replaces the original application's main activity and binds the `RemoteService` to create a connection between the caller and callee micro-apps. It then launches the application's original main activity. This ensures that the communication channel is open and ready by the time the micro-app needs the functionality.

3.3 Experimental Evaluation

In order to test the A3P partitioning system, we performed three sets of tests. The first was on a small number of test applications written to exercise Android features that cover the space of IPC mechanisms. The second was on a set of more realistic applications we developed specifically to test partitioning on a full application. The third was on a set of applications downloaded from the Google Play Store. Each of these corpora and the results of our testing are described in the following sections.

3.3.1 Test Applications

In order to test a wide variety of mechanisms for communicating between applications, we developed 18 test applications that each use a single type of IPC mechanism. These application are designed to be as simple as possible from a program analysis standpoint in order to minimize

complexity in the underlying program analysis, and they exhibit no useful behavior visible to the user. To simplify execution testing, each side of the IPC was responsible for logging a message with its class name to the Android system log using the `android.util.Log` class. We have published these applications on GitHub in the hope that they will be used by others to test similar program analyses.

In order to test on realistic applications, we developed five more complex full applications. These applications are more complex than the IPC test apps, but less sophisticated than a real-world Android application. In particular, they do not contain advertising libraries or native code, and they are relatively small. However, since we wrote the applications, we had a significantly better understanding of how the applications functioned. This “ground truth” helped us understand and debug the A3P system’s results.

We also sampled five applications from the Google Play Store. These applications were selected at random from a larger set on which we were able to complete the DroidSafe static program analysis step in a reasonable amount of time. Note that these applications are not representative of the full spectrum of apps available on the app store, but the results discussed below clearly illuminate the future work required to support real-world applications.

The application sizes are listed in the table below, along with a brief description of the app. Note that the Standard Lines of Code (SLoC) metric listed in the first two tables only includes the Java application sources for the application. Libraries and the operating system itself are not included, although portions of them are analyzed as part of the information flow analysis. In general, SLoC is a very weak predictor of analysis time, but it serves to give a feel for the complexity of the applications.

Table 2 - IPC Test Applications

IPC Test Application	SLoC	Description
StartActivityClassConst	25	Starts an activity via a .class constant
StartActivityStringConst	27	Starts an activity via fully qualified class name (FQCN)
StartActivityImplicit	24	Starts an activity by action, without specifying the class
StartActivityFromService	44	Starts an activity from within a Service
StartActivityFromReceiver	39	Starts an activity from within a BroadcastReceiver
StartActivityForResultClassConst	39	Starts an activity and waits for result, using .class
StartActivityForResultStringConst	40	Starts an activity and waits for result, using FQCN
StartActivityForResultImplicit	40	Starts an activity and waits for result, implicit intent
StartActivitiesFromActivity	40	Starts multiple activities using an array of intents
PendingIntentActivity	33	Create an activity via a PendingIntent
StartServiceClassConst	35	Starts a service via its .class constant
StartServiceFromService	50	Starts a service from within a Service
StartServiceFromReceiver	45	Starts a service from within a BroadcastReceiver
BindServiceClassConst	110	Binds a service via its .class constant
SendBroadcast	24	Sends a broadcast intent to start an activity
SendBroadcastFromService	42	Sends a broadcast to start an activity from a Service

SendBroadcastFromReceiver	39	Sends a broadcast to start an activity from a BroadcastReceiver
SendOrderedBroadcast	24	Sends an ordered broadcast that allows for prioritized responders to a broadcast Intent

Table 3 - Realistic Test Applications

Realistic Application	SLoC	Description
CallBlocker	1028	Block incoming calls based on caller ID blacklisting
CryptoPic	1081	Take and view pictures encrypted with a passphrase
MyLibrary	916	Inventory management for personal libraries, including scanning bar codes, information lookup, etc.
ColorPicker	184	Simple multi-activity application for testing IPC

Table 4 - Google Play Store Applications

Play Store Applications	SLoC	Description
app.juri.ukulele-12.apk	n/a	App for ukulele chords, tuning, and other guides
bingle.min.games.csaw-3.apk	n/a	Puzzle game
br.com.somzoom-4.apk	n/a	Radio station app
b0bby.morsecode-1.apk	n/a	Morse code trainer
br.com.netfilter.avioes-3.apk	n/a	App for reading information about different aircraft and marking ones you have flown

4. RESULTS AND DISCUSSION

The results per application are summarized in the tables below.

4.1 IPC Test Applications

Application	Processing Time (sec)	Result Functional?
StartActivityClassConst	364	✓
StartActivityStringConst	375	✓
StartActivityImplicit	368	✓
StartActivityFromService	375	✓
StartActivityFromReceiver	388	✓
StartActivityForResultClassConst	364	✓
StartActivityForResultStringConst	391	✓
StartActivityForResultImplicit	372	✓
StartActivitiesFromActivity	392	✗
PendingIntentActivity	364	✓
StartServiceClassConst	371	✓
StartServiceFromService	390	✓
StartServiceFromReceiver	377	✓
BindServiceClassConst	359	✓
SendBroadcast	378	✓
SendBroadcastFromService	383	✓
SendBroadcastFromReceiver	383	✗
SendOrderedBroadcast	378	✓

For the IPC test applications, we put the A3P system into a special mode that attempted to isolate each **Activity** or **Service** in its own partition. This forced the system to perform the partitioning without having to introduce permissions or other complexities into the applications that would induce partitioning via our existing heuristics. Without it, the system would have left the apps intact.

Of the 18 IPC test applications, A3P is able to automatically partition 16 of them into micro-apps that maintain the original application functionality. There were two failures in this set, both of which are due to imprecision in the information flow analysis that underlies A3P.

For each of the 16 applications that we were able to partition successfully, the system correctly split the application into two or more micro-apps, rewrote the IPC mechanisms correctly, and repackaged the micro-apps into installable and runnable APK files. We were able to install the micro-apps, run the “main” micro-app, and see all of the diagnostic log messages we were expecting.

In the case of `StartActivitesFromActivity`, the main part of the application declares an array of `Intent` objects, each of which will trigger the launch of an activity. However, the DroidSafe information flow analysis is not sensitive with respect to arrays. This allows for scaling to larger applications at the cost of confusing values within an array. Our analysis loses track of which intent connects to which target activity and prevents proper reconstruction of the application workflow. The A3P system thus cannot determine how to partition the application and reports a failure. To fix this issue, we would need to extend the information flow analysis in DroidSafe to cover values in arrays. This is a non-trivial change to the analysis and would likely cause large increases in processing times, perhaps to the level of intractability for real-world applications.

In the case of `SendBroadcastFromReceiver`, the main application sets up an activity to receive broadcast messages. This pattern of dynamic receiver registration is not uncommon in Android applications, but is not modeled in the DroidSafe information flow analysis. This makes the A3P system unable to determine who the receiver of the broadcast message will be. In this case, A3P must be conservative and assume the broadcast can be sent anywhere, which makes the partitioning algorithm unable to determine an optimal partition and report a failure. To fix this issue, we would need to add dynamic broadcast receiver registration to the DroidSafe Android system model.

4.2 Realistic Applications

Application	Processing Time (sec)	Result Functional?
CallBlocker	414	✗
CryptoPic	626	✓
MyLibrary	7703	✓
ColorPicker	377	✓

For the four realistic applications we developed in-house and analyzed, we were able to successfully construct working micro-apps for three of the four applications. In the CallBlocker application, we were able to partition the application, but unable to generate micro-apps due to limitations in AIDL. Results for each of the applications are given below, along with a brief discussion.

4.2.1 CallBlocker

The CallBlocker application was partitioned into nine micro-apps. Of these nine micro-apps, eight had permissions, and one was responsible for handling the interactions with the app’s

database. There is one class that is cloned and shared between slices 1, 7, and 9 – this class is a data-only class that represents a contact as a pair of strings.

Table 5 - CallBlocker Partitioning Results

Slice	Permissions	Sandbox tags	Classes
1	RECEIVE_BOOT_COMPLETED, READ_PHONE_STATE	SharedPreferences	4
2	ACCESS_NETWORK_STATE	n/a	1
3	WRITE_CALL_LOG, READ_CALL_LOG, ADD_VOICEMAIL	n/a	1
4	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION	n/a	1
5	RECEIVE_BOOT_COMPLETED, RECEIVE_SMS	n/a	1
6	READ_CALENDAR, WRITE_CALENDAR	n/a	1
7	READ_SOCIAL_STREAM, WRITE_SOCIAL_STREAM, READ_SYNC_SETTINGS, READ_CONTACTS, WRITE_CONTACTS	n/a	2
8	INTERNET	n/a	1
9	n/a	Database	9

The diagram below shows the partitioning by classes, where each node is a class, and an edge represents a call from a method in the source class to a method in the target class. Most of the partitions contain a single class, but partitions with multiple classes are grouped within a shaded

polygon. Nodes that appear within multiple polygons are cloned between those partitions. In CallBlocker, the Contact class described above is an example of this.

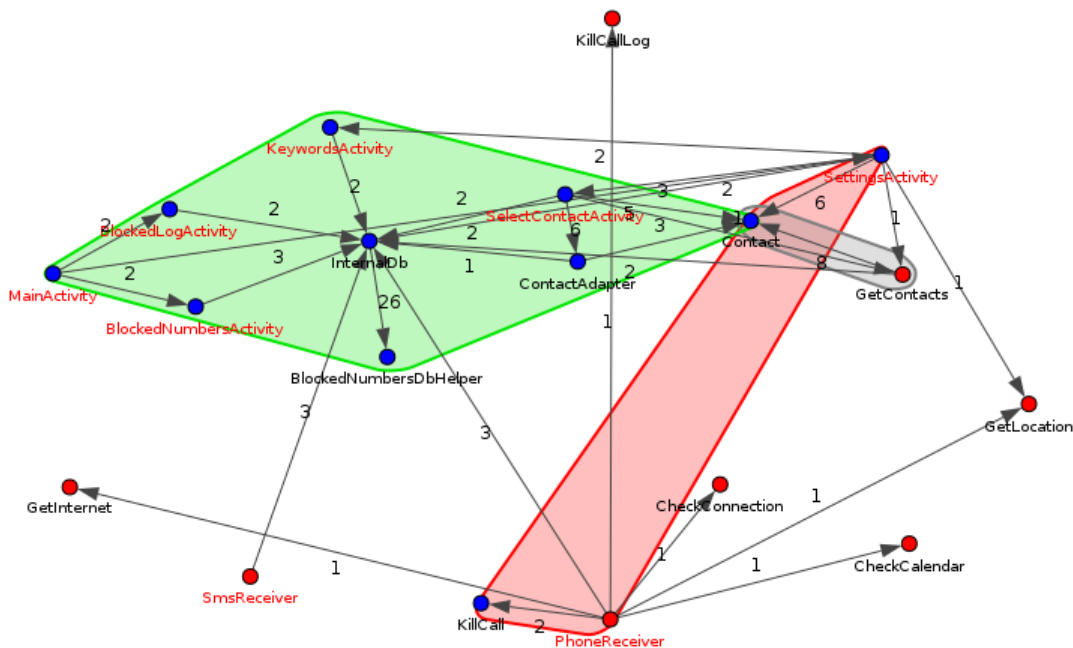


Figure 5 - CallBlocker is split into nine partitions, most of which contain a single class requiring a dangerous permission

The CallBlocker micro-apps are unable to reproduce the original application’s functionality due to a single issue: AIDL only handles Java primitive types (int, char, long, etc.), strings, character sequences, lists, and maps. The partitioning chosen by A3P for the CallBlocker application would have required the AIDL interface to ship a complex object across partitions. Since the generated AIDL interface cannot do this natively, we were unable to create the appropriate micro-apps. To solve this, we would need to replace AIDL with a general-purpose object serialization/deserialization capability. This is discussed further in the Future Work section to follow.

4.2.2 CryptoPic

For the CryptoPic application, A3P generates two partitions, each of which contain a handful of application classes and a larger number of Android Support Library classes. The class that can take pictures is isolated from the classes that manage the data store, limiting the potential attack surface.

Table 6 - CryptoPic Partitioning Results

Slice	Permissions	Sandbox tags	Classes
1	CAMERA	File	4
2	n/a	Database	2

The resulting micro-apps are able to function properly – the user can take a picture, find it through the database explorer, and then open and view or share the picture, as per the original application’s functionality.

The figure below shows the partitioning of CryptoPic. The large number of cloned classes in the middle of the diagram are Android Support Library classes whose implementations are provided by the operating system.

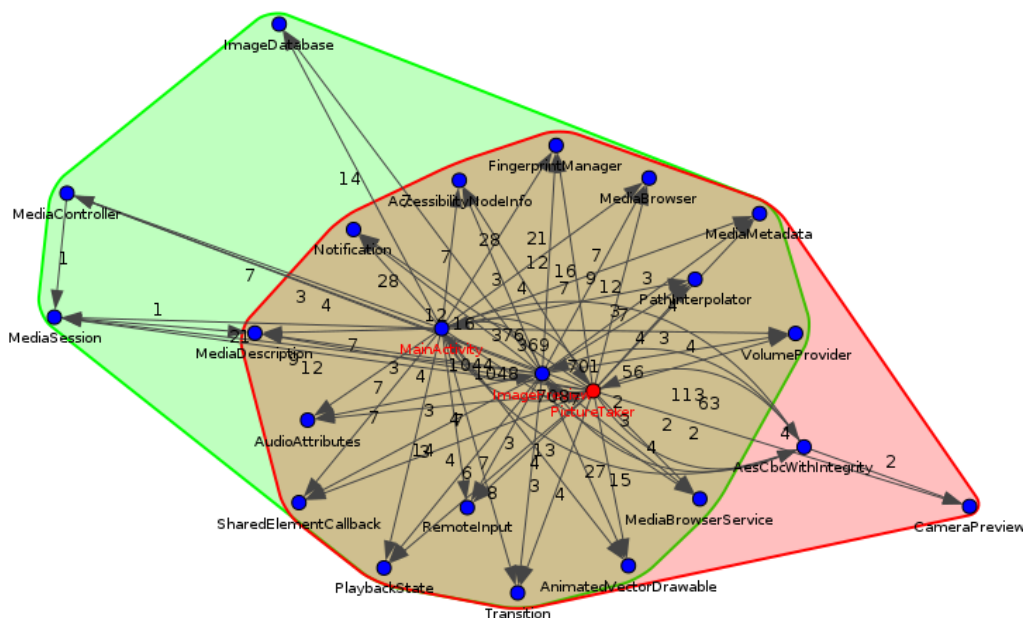


Figure 6 - CryptoPic is partitioned into two micro-apps with disjoint permissions

4.2.3 MyLibrary

The MyLibrary application is divided into three partitions, two of which require dangerous permissions, and one of which manages the database layer. The large number of classes in the third partition are mostly from the barcode scanning library that the application uses. There is a single cloned class called **BookInfo** shared between slices 2 and 3 – it is a data-only object that stores a book’s metadata.

Table 7 - MyLibrary Partitioning Results

Slice	Permissions	Sandbox tags	Classes
1	CAMERA	n/a	2
2	INTERNET	n/a	2
3	n/a	Database	251

Table 9 - Google Play Store Application Partitioning Results

Application	Processing Time (sec)	Result Functional?
app.juri.ukulele-12.apk	759	X
bingle.min.games.csaw-3.apk	405	X
br.com.somzoom-4.apk	426	X
b0bby.morsecode-1.apk	360	X
br.com.netfilter.avioes-3.apk	388	X

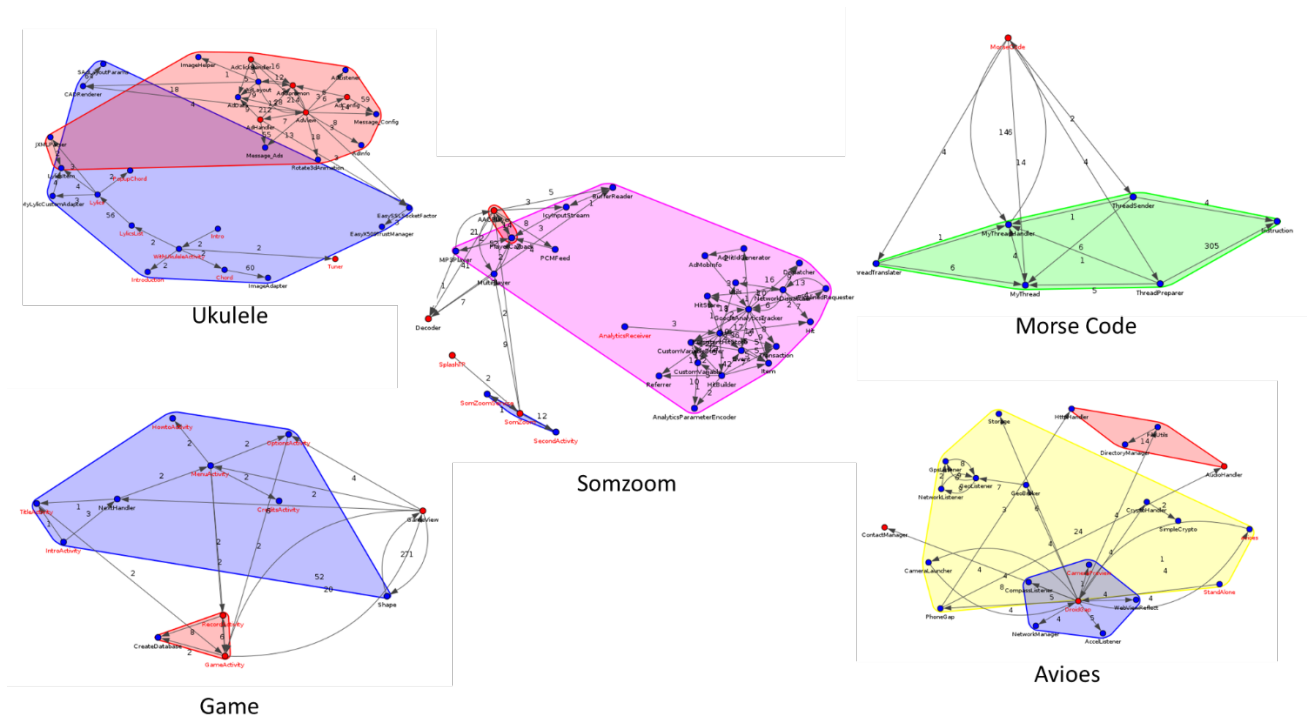


Figure 8 - Play Store Apps are partitioned based on required permissions and use of the application sandbox

4.3.1 Ukulele

Slice	Permissions	Sandbox tags	Classes	Notes
1	INTERNET, ACCESS_NETWORK_STATE, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION	File	15	Advertisement library, as well as the JXMLParser class because it contains a download method

2	WAKE_LOCK	n/a	1	Tuner class, so screen doesn't turn off while tuning
3	n/a	SharedPreferences	14	Rest of classes, which don't use permissions

There is one class cloned across slices 2 and 3, called `LyricsItem`, which is a simple object containing only a chord string and a text string.

4.3.2 Game

Slice	Permissions	Sandbox tags	Classes	Notes
1	ACCESS_NETWORK_STATE, INTERNET, READ_PHONE_STATE	Database, SharedPreferences	3	All of the classes that use the database
2	WAKE_LOCK, VIBRATE	n/a	1	The longest class; sets up the game board and handles the game events
3	n/a	SharedPreferences	8	The rest of the classes

There are no cloned classes in this application.

4.3.3 SomZoom

Slice	Permissions	Sandbox tags	Classes	Notes
1	INTERNET	File	2	AACPlayer and the interface it implements
2	GET_TASKS	n/a	1	Checks task list to prevent opening more than one instance of the app
3	VIBRATE	n/a	3	One class uses a permission and the other two are called by it and are easily

				separable from the rest of the graph
4	[Native code]	n/a	1	Audio decoder
5	n/a	Database	25	The rest

There is a single class cloned across slices 1 and 5 called **PlayerCallback**, which is an interface containing methods such as **playerStarted** and **playerStopped**. Implementations of these interface methods are called by other parts of the application when the music player starts and stops, respectively.

Interestingly, SomZoom uses native code to perform its audio decoding tasks. A3P is able to handle this as per its native code policy, which isolates the audio decoder within its own process space.

4.3.4 Morse code Trainer

Slice	Permissions	Sandbox tags	Classes	Notes
1	WAKE_LOCK	n/a	1	Main class, needs wake lock so screen doesn't turn off during input/output
2	n/a	n/a	6	The rest (translation logic, input validation, etc.)

4.3.5 Avioes

Slice	Permissions	Sandbox tags	Classes	Notes
1	WAKE_LOCK	File	4	All 4 classes involve file operations (reading, writing, deleting, moving)
2	READ_SYNC_SETTINGS, READ_SOCIAL_STREAM, WRITE_SOCIAL_STREAM, READ_CONTACTS, WRITE_CONTACTS	n/a	1	Searching contacts (presumably for sharing purposes?)
3	INTERNET	n/a	6	One of the classes needs the Internet and the other 5 are called by it and are

				the most easily separable from the rest of the graph
4	n/a	Database	11	The rest (only 1 class uses the database)

In analyzing the Avioes application, we noticed that the application is built using the PhoneGap [28] app construction tool. A3P analyzes this application correctly, but cannot partition it due to the need to send complex objects across AIDL interfaces.

5. CONCLUDING REMARKS

The A3P approach shows great promise for separating applications into cooperating micro-apps. By combining information about the program's control structure, its security properties, and its information flow, we are able to calculate a partitioning of the application that divides regions of critical functionality into separable micro-apps that replicate the original application's functionality. This partitioning introduces new security boundaries that protect user information and create distinct failure domains for critical functions. We are also able, in many cases, to automatically turn this partitioning into a collection of installable and runnable micro-apps that replicate the original application functionality.

5.1 Potential Future Work

There are a variety of interesting opportunities to extend and expand the capabilities we have developed to cover a larger part of the Android application space, to protect the micro-apps we create, and to further control the application's security surface. In addition, while the particular tactics we have developed are tailored for Android applications, there is no fundamental reason why the approach cannot be applied to other environments, such as iOS apps or desktop applications.

The most obvious shortcoming in our system is our inability to handle applications that must send complex stateful objects across application boundaries. This is a critical issue to address in order to function on real-world applications, as shown by our experiments on real-world apps. The limitation is primarily due to our selection of AIDL as a remote procedure call interface. We believe that using another RPC library with a more complex object transfer capability would alleviate this problem.

The partitioning process also introduces opportunities to introduce other kinds of security measures. For example, our current system makes no concerted effort to protect the IPC interfaces that it introduces. In a real-world system, we would obviously want to ensure that the only IPC calls that could be made were the ones present in the original application. This would prevent abuse of the micro-apps by third parties.

In addition to protecting micro-apps from third parties, the partitioning could also allow users to exert finer-grained control over how data is used on their device. For example, one might consider a "data firewall" that could log or control what kinds of data can be sent between micro-apps. Such an interposition capability could also be used to introduce privacy-preserving transformation or encryption of data without modifying the original application code.

6. REFERENCES

- [1] Snoopwall, "Flashlight Apps Threat Assessment Report," 2014.
- [2] R. Paul, "Some Android apps caught covertly sending GPS data to advertisers," *Ars Technica*, 30 9 2010. [Online]. Available: <https://arstechnica.com/security/2010/09/some-android-apps-found-to-covertly-send-gps-data-to-advertisers/>. [Accessed 01 07 2017].
- [3] M. Braga, "What Exactly Is the Facebook Messenger App for iOS Tracking?," *Vice Motherboard*, 10 9 2014. [Online]. Available: http://motherboard.vice.com/en_uk/read/facebooks-messenger-app-is-tracking-a-lot-more-of-your-data-than-you-think. [Accessed 01 07 2017].
- [4] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vols. SE-10, no. 4, pp. 352-357, 1984.
- [5] S. Horowitz, J. Prins and T. Reps, "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345-387, 1989.
- [6] R. Gupta, M. J. Harrold and M. L. Soffa, "Program Slicing-Based Regression Testing Techniques," in *Proceedings of the Conference on Software Maintenance*, 1992.
- [7] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu and T.-f. F. J. Ngai, "New Slicing Algorithms for Parallelizing Single-Threaded Programs," in *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, Beijing, 2008.
- [8] J. Cheng, "Slicing Concurrent Programs," in *First International Workshop, AADEBUG*, Linkoping, Sweden, 1993.
- [9] K. Hogstedt, D. Kimelman, V. Rajan, T. Roth and M. Wegma, "Graph Cutting Algorithms for Distributed Applications Partitioning," *ACM SIGMETRICS Performance Evaluation Review*, pp. 27-29, 2001.
- [10] S. Shekhar, M. Dietz and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications," in *USENIX Security Symposium*, 2012.
- [11] X. Zhang, A. Ahlawat and W. Du, "AFrame: Isolating advertisements from mobile applications in Android," in *Computer Security Applications Conference*, 2013.
- [12] J. Seo, D. Kim, D. Cho, I. Shin and T. Kim, "FLEXDROID: Enforcing In-App Privilege Separation in Android," in *NDSS*, 2016.
- [13] M. Sun and G. Tan, "NativeGuard: protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014.
- [14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, 2011.
- [15] Y.-W. Kwon and E. Tilevich, "Reducing the Energy Consumption of Mobile Applications Behind the Scenes," in *ICSM*, 2013.

- [16] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon and W. Heinzelman, "Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture," in *IEEE Symposium on Computers and Communications (ISCC)*, Cappadocia, 2012.
- [17] S. Kosta, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of IEEE Infocom*, 2012.
- [18] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant and D. Song, "Hi-cfg: Construction by binary analysis and application to attack polymorphism," in *European Symposium on Research in Computer Security*, 2013.
- [19] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen and M. Rinard, "Information-Flow Analysis of Android Applications in DroidSafe," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2015.
- [20] P. Pearce, A. Porter Felt, G. Nunez and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.
- [21] G. Csárdi and T. Nepusz, "The igraph software package for complex network research.," *InterJournal Complex Systems*, p. 1695, 2006.
- [22] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. de Geus, C. Kruegel and G. Vigna, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," in *NDSS '16*, San Diego, CA, 2016.
- [23] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE transactions on Software Engineering*, vol. 1, pp. 85-93, 1977.
- [24] C. Tumbleson and R. Wisniewski, "Apktool," 13 June 2017. [Online]. Available: <https://ibotpeaches.github.io/Apktool/>. [Accessed 01 06 2017].
- [25] Sable, "Instrumenting Android Apps with Soot," [Online]. Available: <https://github.com/Sable/soot/wiki/Instrumenting-Android-Apps-with-Soot>. [Accessed 19 April 2017].
- [26] Google, Inc., "Android Interface Definition Language," [Online]. Available: <https://developer.android.com/guide/components/aidl.html>. [Accessed 01 07 2017].
- [27] N. Viennot, E. Garcia and J. Nieh, "A Measurement Study of Google Play," *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.
- [28] Adobe, Inc., "Adobe PhoneGap," [Online]. Available: <https://phonegap.com/>. [Accessed 01 07 2017].

GLOSSARY/ACRONYMS

Term	Definition
A3P	Automated Android Application Partitioning
AIDL	Android Interface Definition Language for specifying and generating remote procedure call interfaces
APAC	DARPA's Automated Program Analysis for Cybersecurity program - focused on detecting malicious Android applications through static program analysis
API	Application Programming Interface
APK	Android Application Package - the binary archive format in which Android applications are typically distributed
app	application
ARM	Advanced RISC Machines
Artemis	Static program analysis tool developed by BBN for the APAC program
CCG	Class Call Graph
Dalvik	Virtual machine used by Android to run applications
DEX	Dalvik executable - byte code format for Android
GNU	GNU's Not Unix operating system (recursive acronym)
ID	identification
ICFG	Information- and Control- Flow Graph - a directed graph representing control flow transfers and information flows within an application
Intent	Android inter-component communication message used to start activities, invoke method calls, etc.
iOS	A mobile operating system (formerly iPhone OS)
IPC	Inter-process communication
JSON	JavaScript Object Notation
Micro-app	A stand-alone portion of an input application that has been carved out of the input and transformed to operate independently
OS	Operating System
OS X, OSX	Version 10 of the Apple Macintosh operating system (OS)
Permission	Android system privilege primitive that allows access to a particular feature of the host device. Prior to Android 6.0 (API 23), permissions were granted at the time of installation. In API 23 and later, permissions can be granted at installation or granted dynamically by the user.

RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SELinux	Security-Enhance Linux
SLOC	Standard Lines of Code
Soot	General-purpose Java program analysis and rewriting tool
STAC	DARPA's Space/Time Analysis for Cybersecurity program
x86	Family of backward-compatible instruction set architectures
XML	Extensible Markup Language