June 2018

Precise, General, and Efficient Data-flow Analysis for Security Vetting of Android Apps

Fengguo Wei
Univerisity of South Florida, fwei@mail.usf.edu

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Precise, General, and Efficient Data-flow Analysis for
Security Vetting of Android Apps

by

Fengguo Wei

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Xinming Ou, Ph.D.
Jarred Ligatti, Ph.D.
Yao Liu, Ph.D.
Nasir Ghani, Ph.D.
Robby, Ph.D.

Date of Approval:
May 25, 2018

Keywords: Static Analysis, Mobile Security, Android App Security Analysis

Copyright © 2018, Fengguo Wei
DEDICATION

To my wife, and to my loving family.
ACKNOWLEDGMENTS

I would like to thank my advisor Xinming Ou for all his guidance and encouragement throughout my Ph.D. journey. His spirit inspires me to always start from real-world cybersecurity problems and always outside-the-box to seek solutions. I am also thankful for him to provide me the opportunity to work on research problems that I am interested in continually.

I would like to thank Sankardas Roy for always supporting me with faithful feedbacks. In retrospect, I feel that I have been fortunate to have someone who has such an experience and attitude in scientific research and guides me smoothly going through the first few years of my Ph.D. journey.

I would like to thank Robby for his guidance on the use programming analysis methods. His insightful suggestions helped immensely in carrying out the framework I built. This research would not have been possible without his continuous mentorship.

I would like to thank my other committee members, Jarred Ligatti, Yao Liu, Nasir Ghani, for spending the tremendous amount of time helping me improve the presentation of this dissertation.

At last, I would like to thank my fellow Argus Lab members for the many memorable times and fruitful discussions.

This research was partially supported by the U.S. National Science Foundation under Grants No. 0644288, No. 0954138, No. 1018703, No. 1717862, and No. 1718214, and the U.S. Air Force Office of Scientific Research under Award No. FA9550-09-1-0138.
TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

ABSTRACT vii

CHAPTER 1: INTRODUCTION 1
  1.1 Android Security Issues and Vetting System 1
  1.2 Previous Works on Android Security Analysis 2
    1.2.1 Android Static Analysis 2
    1.2.2 Dynamic and Hybrid Analysis 5
  1.3 Android Static Analysis Challenges 6
  1.4 Contributions 7

CHAPTER 2: ANDROID SYSTEM MODEL 10
  2.1 Runtime Environment Model 10
    2.1.1 Event-based System 10
    2.1.2 Component-level Environment Model 12
  2.2 System API Model 13
    2.2.1 Heap Manipulation Summary Language 14

CHAPTER 3: DATA-FLOW ANALYSIS 16
  3.1 Reaching Facts Analysis (RFA) 17
    3.1.1 Notations 17
    3.1.2 The Basic DFG Building Process 18
    3.1.3 Handle Library APIs 22
  3.2 Summary-based Bottom-up Data-flow Analysis (SBDA) 22
  3.3 Discussion 25

CHAPTER 4: COMPONENT-BASED ANALYSIS 27
  4.1 Motivating Example 27
  4.2 Building the Component-level Data-flow Graph 30
  4.3 Building the Component-level Data Dependence Graph 30
  4.4 Linking Inter-component Data Flows 32
    4.4.1 Intent 36
      4.4.1.1 ST Construction 36
      4.4.1.2 Stitching Intent Channels – Intent Destination Resolution 37
    4.4.2 RPC 38
      4.4.2.1 ST Construction 38
## CHAPTER 7: EVALUATION

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Performance and Scalability</td>
<td>68</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Component-based Analysis</td>
<td>68</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Inter-language Analysis</td>
<td>69</td>
</tr>
<tr>
<td>7.2</td>
<td>Intent Resolution Effectiveness</td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>Statistics of Native Library Usage</td>
<td>72</td>
</tr>
<tr>
<td>7.4</td>
<td>Application to Security Analysis</td>
<td>74</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Password Leakage Checker</td>
<td>74</td>
</tr>
<tr>
<td>7.4.2</td>
<td>OAuth Token Leakage Checker</td>
<td>75</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Hiding-icon Checker</td>
<td>76</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Crypto Library Misconfiguration Checker</td>
<td>76</td>
</tr>
<tr>
<td>7.4.5</td>
<td>SSL/TLS Misconfiguration Checker</td>
<td>77</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Communication Data Leakage Checker</td>
<td>77</td>
</tr>
<tr>
<td>7.4.7</td>
<td>Intent Injection Checker</td>
<td>78</td>
</tr>
<tr>
<td>7.4.8</td>
<td>Inter-language Security Checker</td>
<td>79</td>
</tr>
<tr>
<td>7.4.8.1</td>
<td>Case Study 1: Inter-language Data Leakage</td>
<td>79</td>
</tr>
<tr>
<td>7.4.8.2</td>
<td>Case Study 2: Stealthy Command Execution</td>
<td>80</td>
</tr>
<tr>
<td>7.4.8.3</td>
<td>Case Study 3: Stealthy Command and Control Communication</td>
<td>80</td>
</tr>
<tr>
<td>7.4.8.4</td>
<td>Case Study 4: Malicious Identity Hiding</td>
<td>80</td>
</tr>
<tr>
<td>7.5</td>
<td>What it Takes to Build a New Analysis</td>
<td>81</td>
</tr>
<tr>
<td>7.6</td>
<td>Comparison with Existing Tools</td>
<td>81</td>
</tr>
</tbody>
</table>

## CHAPTER 8: CONCLUSIONS

88

## REFERENCES

90

## APPENDIX A: COPYRIGHT PERMISSIONS

101
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Communication points of an app component</td>
<td>35</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Intent resolution precision</td>
<td>72</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Native library statistics for datasets</td>
<td>73</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>Password leakage checker report</td>
<td>75</td>
</tr>
<tr>
<td>Table 7.4</td>
<td>OAuth token leakage checker report</td>
<td>76</td>
</tr>
<tr>
<td>Table 7.5</td>
<td>Crypto library misconfiguration checker report</td>
<td>77</td>
</tr>
<tr>
<td>Table 7.6</td>
<td>SSL/TLS misconfiguration checker report</td>
<td>78</td>
</tr>
<tr>
<td>Table 7.7</td>
<td>Communication data leakage checker report</td>
<td>78</td>
</tr>
<tr>
<td>Table 7.8</td>
<td>Intent injection checker report</td>
<td>79</td>
</tr>
<tr>
<td>Table 7.9</td>
<td>Code size and running time (checkers and core)</td>
<td>81</td>
</tr>
<tr>
<td>Table 7.10</td>
<td>Results on DroidBench</td>
<td>83</td>
</tr>
<tr>
<td>Table 7.11</td>
<td>Results on ICC-Bench</td>
<td>84</td>
</tr>
<tr>
<td>Table 7.12</td>
<td>Results on NativeFlowBench</td>
<td>86</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Activity lifecycle</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Lifecycle example</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Library API call example</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Building the DFG for foo: The intra-procedural control flow graph (CFG) of foo is extended to a callee, bar</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>SBDA workflow</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Heap manipulation summary example</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>The inter-component-leak App: The arrowed lines among the app components highlight some of the inter-component communication</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>DFGs and STs of the components in App &quot;inter-component-leak&quot;: An excerpt</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Data flow between app components via RPC and static field</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>The inter-language-leak App: The arrowed lines among the app components highlight some of the inter-language-communication</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>JNINativeMethod structure</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>JNINativeInterface and JNIInvokeInterface structures</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>GetStringUTFChars function model</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>getCharFromString function source code and assembly</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>native_activity.h example</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>android_native_app_glue.h example</td>
<td>57</td>
</tr>
<tr>
<td>6.1</td>
<td>The Argus-SAF analysis pipeline</td>
<td>59</td>
</tr>
</tbody>
</table>
Figure 6.2  APK preprocess and environment model 60
Figure 6.3  RFA-based analysis 62
Figure 6.4  SBDA-based analysis 63
Figure 6.5  Inter-component analysis 65
Figure 6.6  Argus-SAF actor model 66
Figure 7.1  Time to build $DFG$ 69
Figure 7.2  Time to run $SBDA$ 70
Figure 7.3  Native code analysis performance 71
Figure 7.4  Hiding-icon code snippet 76
This dissertation presents a new approach to static analysis for security vetting of Android apps, and a general framework called Argus-SAF. Argus-SAF determines points-to information for all objects in an Android app component in a flow and context-sensitive (user-configurable) way and performs data-flow and data dependence analysis for the component. Argus-SAF also tracks inter-component communication activities. It can stitch the component-level information into the app-level information to perform intra-app or inter-app analysis. Moreover, Argus-SAF is NDK/JNI-aware and can efficiently track precise data-flow across language boundary. This dissertation shows that, (a) the aforementioned type of comprehensive app analysis is utterly feasible in terms of computing resources with modern hardware, (b) one can easily leverage the results from this general analysis to build various types of specialized security analyses – in many cases the amount of additional coding needed is around 100 lines of code, and (c) the result of those specialized analyses leveraging Argus-SAF is at least on par and often exceeds prior works designed for the specific problems, which this dissertation demonstrate by comparing Argus-SAF’s results with those of prior works whenever the tool can be obtained. Since Argus-SAF’s analysis directly handles inter-component and inter-language control and data flows, it can be used to address security problems that result from interactions among multiple components from either the same or different apps and among java code and native code. Argus-SAF’s analysis is sound in that it can assure the absence of the specified security problems in an app with well-specified and reasonable assumptions on Android runtime system and its library.
CHAPTER 1: INTRODUCTION

1.1 Android Security Issues and Vetting System

The Android smart-phone platform is immensely popular and has by far the most significant market share among all types of smartphones worldwide. However, there have been widely reported security problems due to malicious or vulnerable applications running on Android devices.

Fuchs et al. [1], Gibler et al. [2] described the sensitive data leakage issue frequently occurred for Android applications.

Lu et al. [3], Wang et al. [4] discovered a particular type of data injection problem in Android domain, called intent injection vulnerability. This type of vulnerability allows the attacker to abuse capability provided by Android component and escalate their privilege.

Poeplau et al. [5] systematically studied the dynamic code loading functionality in an Android application. They revealed this capability often time being abused by benign apps, as well as leveraged by malware to execute the malicious payload stealthily.

Felt et al. [6], Zhou et al. [7, 8], Wei et al. [9] profiled the malicious behavior and technology evolution for Android malware over the past years. Based on their description, the majority of Android malware targeted on malicious behaviors, such as, stealing device user’s sensitive information, damaging the device, or annoying the user.

Recent studies [8, 9, 10, 11, 12, 13, 14, 15] have shown that native code is a continuous threat which might stealthily leak sensitive information or utilize Android malware to evade AV detection.

The current solutions to those security problems are mostly reactive (e.g., pulling an app off the market after potential damage may have already been done). There have not been effective vetting methods that market operators can rely upon to ensure apps entering a market (e.g., Google Play) are free of certain types of security problems. Often, they have to resort to dynamic analysis
— running an app in a testing environment with the hope of identifying the problematic behaviors, if any, during the test run (e.g., Google Bouncer [16]).

Many security problems of Android apps can be discovered by static analysis on the Dalvik bytecode of the apps, and there have been a number of earlier efforts along this line [3, 8, 10, 13, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Compared with dynamic analysis, static analysis has the advantage that a malicious app cannot easily evade detection by changing their behaviors in a testing environment, and it can also provide a comprehensive picture of an app’s possible behaviors as opposed to only those that manifest during the test run. Due to the inherent undecidability nature of determining code behaviors, any static analysis method must make a trade-off between computing time and the precision of analysis results. Precision can be characterized as metrics on:

1) missed behaviors (app behaviors missed by the analyzer that may present security risks, also referred to as false negatives), and

2) false alarms (behaviors that an app does not possess, but the analyzer fails to rule out, also referred to as false positives).

1.2 Previous Works on Android Security Analysis

1.2.1 Android Static Analysis

There has been a long line of works on applying static analysis for Android security problems [3, 17, 18, 19, 25, 26, 28, 29, 30, 31, 32, 33, 34]. Most closely related works are described below.

The design of Argus-SAF leverages a number of approaches from FlowDroid [19, 30] (e.g., callback collection algorithm during environment generation), but the two also have significant differences. FlowDroid does not handle ICC and as such cannot address security issues involving intent passing among multiple components. FlowDroid builds a call graph based on Spark/Soot [35], which conducts a flow-insensitive points-to analysis. FlowDroid then conducts a taint and on-demand alias analysis based on the above call graph, using IFDS [36, 37] which is flow and context-sensitive. The flow-insensitivity in the call graph construction may introduce spurious call edges (false positives), which could impact the analysis precision of the subsequent IFDS analysis. Argus-SAF computes the call graph at the same time as the data-flow analysis by computing the flow and
context-sensitive points-to facts; thus its call graph is more precise, which could lead to fewer false positives in the final analysis results. *FlowDroid* does not calculate alias or points-to information for *all* objects in both flow and context-sensitive way. This is a design decision from computing cost concerns [30]. Argus-SAF calculates all objects’ points-to information in both context and flow-sensitive way, with reasonable computing cost (ref. Section 7.1). This capability enables us to build the generic framework supporting multiple security analyses. Moreover, *FlowDroid* avoids to handle native method invocation and applies a comprehensive model for native method calls. Argus-SAF provides a comprehensive analysis scheme for inter-language data-flow analysis.

*Epicc* [18] computes Android Intent call parameters using the same IDE framework as FlowDroid, by modeling the intent data structure explicitly in the flow functions. To the best of our knowledge, *Epicc* does not use the Intent parameter analysis result to resolve the Intent call targets in the general case and has not used the result to perform the inter-component data-flow analysis. Argus-SAF’s approach to deriving Intent parameters is to simply use the flow and context-sensitive points-to information (including that for string objects) already computed in the *DFG*, without the need for a separate data-flow analysis just for Intent. Argus-SAF also uses the Intent call parameter information to link Intent call sites to call targets, resulting in a *DFG* that includes data flow paths both within and across components. Moreover, recent work [38] shows that domain knowledge and probabilistic models can be leveraged in Intent destination resolution, which Argus-SAF could adopt. Moreover, *Epicc* cannot resolve Intent call parameters if it presents in the native code. Argus-SAF has the solution to capture intent calls in native code.

Recently, *IccTA* [26] and *DroidSafe* [25] made advancement in the state-of-the-art of Android app static analysis. *IccTA* extends *FlowDroid* and uses *IC3* [33] as the Intent resolution engine, which can now track data flows through regular Intent calls and returns. However, *IccTA* is yet to track the information flow through remote procedure call (RPC). *DroidSafe* [25] tracks both Intent and RPC calls but does not capture data flows through “stateful ICC” nor inter-app analysis. None of *IccTA* and *DroidSafe* handles native method call.
Recent works [32,34] explored various approaches to tracking Android inter-app communication for security vetting. The new component-based analysis of Argus-SAF as discussed in this paper supports inter-app analysis naturally (see Section 4.6).

Lu et al. [3] use a static-analysis scheme called CHEX to detect *component hijacking* problem in Android, which is reduced to finding information flows. CHEX first constructs *app-splits*, each of which is a code segment reachable from an entry point. It then computes the data-flow summary for each split using Wala [39]. The split summaries are linked in all permutations that do not violate the Android system call sequences and could result in transitive information flow. Argus-SAF computes information flow differently – through the usage of an environment method for each component that calls the relevant callbacks in the right order (per Android system specification), and by building the *DFG* and *DDG* for the complete app. CHEX does not have the provision to track data flow through the ICC channels, which Argus-SAF does.

Chin et al. [17] first systematically studied the attack surface related to Intent. In particular, they identified problems such as *unauthorized intent receipt* and *intent spoofing*. They also developed a static analysis tool which can raise warnings for the above problems in an over-conservative manner. Their tool ComDroid performs flow-sensitive, intra-procedural static analysis, and the paper states that there is a limited inter-procedural analysis that “follows method invocations to a depth of one method call.” Argus-SAF performs a full-fledged inter-procedural data-flow analysis in a flow and context-sensitive way, and also tracks the data flows over the ICC channels. While the author would like to conduct comparison study between ComDroid and Argus-SAF, the link to the ComDroid tool (used to be http://www.comdroid.org) is no longer there.

Most of the existing works that leverage static analysis are focused on finding specific Android security problems, and the static analyses used do not seem to address some critical issues such as the inter-component and inter-language nature of Android app’s execution and the precise modeling of Android’s callback sequences. In contrast, Argus-SAF is a precise, general and efficient inter-component NDK/JNI-aware static analysis framework which can address a broad range of security issues in Android apps.
1.2.2 Dynamic and Hybrid Analysis

*TaintDroid* [21] is a dynamic (runtime) taint-tracking and analysis system to find potential misuse of the user’s private information.

*DroidScope* [10] is an Android application dynamic analysis tool that reconstructs OS level and *DVM* level information. *DroidScope* collects detailed native and *Dalvik* instruction traces, profile API-level activity, and track information leakage through both the Java and native components using dynamic taint analysis.

*NDroid* [11] performs dynamic taint analysis based on *QEMU* and tracks information flows through *JNI*. *NDroid* instruments important related JNI functions to resolve information flows, such as JNI entry, JNI exit, object creation. Moreover, It models the system library instead of instrumenting those standard functions to reduce overhead. However, similar to all dynamic analysis systems, *NDroid* has the path coverage issue, and it does not track control flows.

*TaintART* [40] applies dynamic taint tracking by instrumentation the *ART* compiler and runtime. *TaintART* follows *NDroid’s* method to handle JNI calls.

*Harvester* [27] employs hybrid analysis for extracting runtime values. When encountered with native methods, *Harvester* monitors them as logging points to extract runtime values instead of stepping into the native code to conduct the analysis.

*Going Native* [14] conducts static analysis to filter apps containing native code firstly and then perform dynamic analysis to study the native code usage of real-world Android apps. Then it generates native code sandboxing security policy.

*Malton* [15] is a dynamic analysis platform aimed to do malware detection that runs on *ART* runtime. *Malton* conducts multi-layer monitoring including native layer and information flow tracking to provide a comprehensive view of the Android malware behaviors.

*DroidNative* [41] utilizes specific control flow patterns to reduce the impact of obfuscations and use it as semantic-based signatures to detect malware in *ART* runtime.

All dynamic analyses are subject to evasion attacks. For example, researchers have shown [42] that Google’s Bouncer [16] can be fingerprinted and hence evaded by a well-crafted app. On the other hand, static analysis investigates the code of the app (such as, along with the app’s manifest),
which determines the runtime behaviors of the app; this makes it attractive for security vetting. Recently Sounthiraraj et al. [22] showed that static and dynamic analysis could be combined to achieve more effective detection/confirmation of security problems. Our approach provides a precise and general static analysis framework that can complement dynamic analyses.

1.3 Android Static Analysis Challenges

A practical challenge in static analysis is to control the rate of false alarms while not missing any (potentially dangerous) behaviors of apps. This is especially significant due to a number of features of Android.

1) Android is an event-based system. The control flow is driven by events from an app’s environment that can trigger various method calls. How to capture all the possible control flow paths in this open and reactive system while not introducing too many spurious paths (false alarms) is a significant challenge. (Solution at Chapter 2.)

2) The Android runtime consists of a large base of library code that an app depends upon. The event-driven nature makes a large portion of the control-flow involve the Android library. While fully analyzing the whole library code could improve the analysis’ faithfulness, it may also be prohibitively expensive (or imprecise). (Solution at Chapter 2.)

3) Android is a component-based system and makes extensive use of inter-component communication (ICC). For example, a component can send an Intent to another component. The target of an Intent could be specified explicitly in the Intent or be implicit and decided at runtime. Both control and data can flow through the ICC mechanism from one component to another. Capturing all ICC flows accurately is a major challenge in static analysis. (Solution at Chapter 4.)

4) Android provides a Native Development Kit (NDK) [43] which allows the developer to design app in native language (C/C++). NDK enables native Activity component, provides a set of native libraries to assist native code to access Android-specific features and uses Java Native Interface (JNI) as the communication bridge. Precisely tracking data flows in native Activity
component and modeling NDK libraries and JNI data structures are significant challenges. (Solution at Chapter 5.)

Prior research has attempted to address some of the above challenges. For example, Flow-Droid [19, 30] formally models the event-driven lifecycle of an Android app in a “dummyMain” method, but it does not address ICC. Epicc [18] statically analyzes Intent and uses an IDE [37] framework to solve for Intent call parameters, but does not link the Intent call sources to targets and does not perform data-flow analysis across component-boundaries. CHEX [3] uses a different approach to the modeling of the Android environment, by linking pieces of code reachable from entry points (called splits) as a way to discover data flows between the Android application components, but it does not address data flow through Intent channels. IC3 [33] is a composite constant propagation engine to solve Intent values in the whole application. IccTA [26] extends FlowDroid and uses IC3 as the Intent resolution engine, which can track data flows through regular Intent calls and returns. However, IccTA is yet to track a special category of ICC named remote procedure call (RPC) that invokes a method in a bound service component. DroidSafe [25] attempts to track both Intent and RPC calls. It performs an app-level analysis with flow-insensitive points-to information. None of the works mentioned above can capture data flows through “stateful ICC,” where component A sends data to B through one ICC, and later component A retrieves that same data from B through another ICC. Moreover, none of the works mentioned above can capture inter-language data-flow.

1.4 Contributions

I designed and built Argus-SAF—a precise, general and efficient data-flow analysis framework tailored for Android apps. The executable and source of Argus-SAF are publicly available.1 The main contributions from Argus-SAF are:

1) Argus-SAF computes points-to information for all objects and their fields at each program point and calling context. The points-to information is extremely useful for analyzing a number of security problems that have been addressed in prior works using customized methods.

---

1Argus-SAF is available at http://pag.arguslab.org/argus-saf.
Argus-SAF can be used to address these wide-range security problems directly with very little additional work. This dissertation also shows that such comprehensive analysis scales to large apps.

2) As part of the computation of object points-to information, Argus-SAF can build a highly precise inter-procedural control flow graph (ICFG) of an app component, which is both flow and context-sensitive [44]. This is a side benefit of our approach compared to prior works that have adopted existing static analysis frameworks (e.g., Soot [35] and Wala [39]), which build ICFG with less precision [45, 46].

3) Argus-SAF adopts a summary-based bottom-up data-flow analysis (SBDA) approach to compute flow and context-sensitive inter-language data-flow information efficiently. The summary-based nature of SBDA enables us to design unified heap manipulation summary representation for both java world and native world data-flow analysis. The bottom-up approach allows us only to visit each method exactly once to compute summary Δ and reuse Δ when a caller method invokes it.

4) Argus-SAF comprehensively models control and data-flow behavior for the Native component, NDK libraries, and JNI data structures to enable existing binary analysis tool, such as Angr [47] to understand Android-specific data flows.

5) For each app component, Argus-SAF builds a Data-flow Graph (DFG), which consists of the component’s ICFG together with each node’s (in ICFG) reaching (points-to) fact set. Then, Argus-SAF builds the data dependence graph (DDG) for each app component from its DFG. Furthermore, for each app component Argus-SAF builds a summary table (ST) listing its inter-component communication (control and data flow) activities over multiple channels, such as Intent, RPC, and static fields. Argus-SAF can conduct an elementary string analysis (due to its object-sensitivity) for inferring Intent/RPC call parameters and finds the correspondence between an ICC source and the ICC targets based on flow and context-sensitive matching algorithm. Using STs of multiple components, Argus-SAF can
stitch the component-level DDGs into an app-level inter-component DDG that supports both intra-app and inter-app analysis.

6) An analyst can add a security checker on top of Argus-SAF to detect the specific security problem he/she is interested. Through extensive experimentation, this dissertation demonstrates that a variety of security problems can be reduced to querying DFGs and DDGs.

Argus-SAF is evaluated extensively on real-world apps. The experimental results show that Argus-SAF scales well. I used Argus-SAF to address security problems such as data leakage (e.g., SMS message leakage), injection (e.g., intent injection), and misuse/abuse of APIs (e.g., to hide app icon). The core framework of I evaluated takes several minutes to analyze one app on average. All the specialized security checkers require minimal additional coding effort (around 100 LOC) to leverage Argus-SAF’s DFGs and DDGs to address the specific problem, and the additional running time is negligible (typically in the order of tens of milliseconds).

Then, an experiment is conducted to compare Argus-SAF with three state-of-the-art static analyzers for Android apps: FlowDroid [19], IccTA [26] and DroidSafe [25], and show that Argus-SAF can address a broader range of security problems due to inter-component communications. Argus-SAF also found multiple crucial security problems in Android apps that were never reported before in the literature.

Some materials presented in this dissertation were published in 21st ACM Conference on Computer and Communications Security (CCS) in 2014 [48] and ACM Transactions on Privacy and Security (TOPS) in 2018 [49].
CHAPTER 2: ANDROID SYSTEM MODEL ¹

2.1 Runtime Environment Model

2.1.1 Event-based System

An Android app is not a closed system; the Android system provides an environment in which the app runs. The code that may execute during the lifetime of an app is not all present in the app’s package. The Android system (which includes the Android runtime) does a bulk of the work in addition to that by the app’s code.

![Activity Lifecycle Diagram](image)

**Figure 2.1: Activity lifecycle**

In Android, numerous types of events (such as system events, UI events) can trigger callback methods defined in an app. As an example, while an Activity A is running, if another Activity B comes to the foreground, it is considered an event. This event could trigger A.onPause(), which

¹This chapter is partially based on the authors CCS 2014 [48] and TOPS 2018 [49] papers. See Appendix A for permission.
is either defined in the app’s code, or in the Android framework if the developer did not override the default method. Figure 2.1 depicts the lifecycle of an Activity. There are seven lifecycle methods of an Activity, such as `onCreate()`, `onPause()`, `onResume()`; they each represent a state in the transition diagram of the lifecycle. Android documentation specifies other states such as *Activity running* and *Activity shut down*. At *Activity running* state, the Activity is capable of responding other types of events, such as button click, GPS location update. Similarly, other types of components have a well-defined lifecycle involving multiple lifecycle methods and can handle different kinds of events.

```
public class MainActivity extends Activity {
    String str;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TelephonyManager tel = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        str = tel.getDeviceId(); // Source
    }

    @Override
    protected void onStart() {
        super.onStart();
        Log.i("imei", str); // Sink
    }
}
```

Figure 2.2: Lifecycle example

Take Figure 2.2 as an example. The *MainActivity* has field `str`. The `onCreate()` method retrieves the device ID (sensitive information) and sets it to the `str` field, whereas the `onStart()` method leak the `str` field into the log. If `onCreate()` and `onStart()` execute independently, this example won’t have any information leakage problem. However, those two methods are running under the same *MainActivity*’s context and executed following the system-defined order. Therefore, to understand the data-flow between `onCreate()` and `onStart()` the static analyzer need to understand and model how the system works.
2.1.2 Component-level Environment Model

As demonstrated in Figure 2.2, a static analyzer needs to model the Android system to analyze the system-defined control flows in the app.\(^2\)

Argus-SAF introduces a component-level model that captures the system-defined flows. The environment of a component \(C\) represents the main method, \(Env\_C\), which takes as parameter an incoming intent \(i\) and invokes \(C\)'s lifecycle methods (e.g., onCreate, onBind, or onReceive) based on \(C\)'s type (Activity, Service, Broadcast Receiver, etc.) and other callback methods (e.g., onLocationChanged) so that all possible paths are included. This component-level model is useful in capturing the impact of the Android system on both the control sequence and data flow of an app’s execution. Argus-SAF has a dedicated environment for each component that invokes the set of callback methods implemented in the component; this is the control part of modeling Android’s environment. Besides, the environment also keeps tracks of the intents received by the component; this is the data part of modeling Android’s environment. \(Env\_C\) also passes the intent parameter when necessary for other relevant methods (e.g., onReceive() of a Broadcast Receiver).

**Algorithm 1 Generating the environment method of component \(C\)**

| Input: The name of the component \(C\), manifest file, resource files, IR of \(C\). |
| Output: \(C\)'s environment method, \(Env\_C\). |

1. procedure \(GEN\_ENV(C)\)
2. create a method \(Env\_C\) having one parameter Intent \(i\), and an empty body;
3. \(callBacks \leftarrow collectCallbacks(C)\);
4. add \(callBacks\) into the body of \(Env\_C\) in the proper sequence emulating the reality;
5. return \(Env\_C\);
6. end procedure
7. procedure \(collectCallbacks(C)\)
8. \(callBacks \leftarrow empty\ Set\);
9. while fixed-point is not reached do
10. perform reachability analysis to mark methods that are reachable from \(C\)
11. \(callBacks \leftarrow callBacks \cup\) callBacks from the XML-resource files
12. \(callBacks \leftarrow callBacks \cup\) interface-based callbacks as registered in \(C\)'s source code
13. \(callBacks \leftarrow callBacks \cup\) other callbacks (system methods that are overridden) in \(C\)'s source
14. end while
15. return \(callBacks\);
16. end procedure

Argus-SAF generates the Environment Method \((Env\_C)\) of each component \(C\) in the app automatically. Algorithm 1 shows the pseudocode for generating \(Env\_C\) of a component \(C\). As the first step, an empty method with an Intent \(i\) as the parameter is generated. (Note that, Intent \(i\)

\(^2\)The alternative is to fully analyze the whole Android system’s code, which is both expensive and unnecessary as also observed by others [3, 25, 26, 30].
typically represents the Intent which starts the component – for instance, \( e.g. \), the parameter of Environment Method of \( BarActivity \) is the intent that starts \( BarActivity \). Then, Argus-SAF collect necessary information from the resource files in the apk and uses this information to collect layout callback methods. Argus-SAF then generate the body of \( Env_C \) with lifecycle methods based on the type of \( C \). Finally, Argus-SAF collect other callback methods (\( e.g. \), \( onLocationChanged \)) in \( C \) (through a reachability analysis) in an incremental fashion. All of these are done before performing the data-flow analysis.

### 2.2 System API Model

Android has a large number of library APIs (that an app can call). Argus-SAF does not analyze system library APIs; thus, it needs to provide models for those methods that summarize how the data-flow facts may be changed. In general, Argus-SAF adopts the following strategy in modeling Android system library APIs:

1) For library APIs that provide essential information for static analysis (\( e.g. \), intent manipulation functions), a precise heap manipulation summary model is built for them based on the function’s implementation and documentation.

2) For all other library APIs, a uniform conservative model is provided. The conservative model essentially assumes that for every object parameter, any of its fields may be modified and becomes \textit{unknown}; that is, the field can points-to a new object, or any existing object reachable from the method parameters (and static fields) that is type compatible. If the method also returns an object, the returned object is also considered unknown.

To construct a meaningful static analysis for Android application, Argus-SAF needs to precisely model thousands of system API methods. Therefore, an effective way is required, such as design a domain specific language (DSL).
2.2.1 Heap Manipulation Summary Language

The following language presents a summary $\Delta$ for a method $m$:

$$\langle \Delta \rangle ::= \langle Rule \rangle^* \langle Rule \rangle^*$$

$$\langle Rule \rangle ::= \langle AssignRule \rangle \mid \langle ActionRule \rangle$$

$$\langle AssignRule \rangle ::= \langle HeapLoc \rangle \langle '=' | '+=' | '-' \rangle \langle RHS \rangle$$

$$\langle ActionRule \rangle ::= \langle Action \rangle \langle '(' \langle RHS \rangle \langle ')' \rangle \langle '@' \rangle \langle Loc \rangle \rangle$$

$$\langle RHS \rangle ::= \langle HeapLoc \rangle \mid \langle Instance \rangle$$

$$\langle Action \rangle ::= \langle '=' \rangle \mid \langle '+=' \rangle \mid \langle '-' \rangle$$

$$\langle HeapLoc \rangle ::= \langle HeapBase \rangle \langle Index \rangle$$

$$\langle HeapBase \rangle ::= \langle 'arg' \rangle \langle Digits \rangle \mid \langle 'ret' \rangle \mid \langle ID \rangle$$

$$\langle Index \rangle ::= \langle '.' \rangle \langle ID \rangle \mid \langle '[' \rangle \langle ']' \rangle$$

$$\langle Instance \rangle ::= \langle ID \rangle \langle '@' \rangle \langle Loc \rangle$$

$$\langle Loc \rangle ::= \langle ID \rangle$$

$\Delta$ consists of a list of $Rule$. There are two types of $Rule$: $AssignRule$ and $ActionRule$. $AssignRule$ defines what kind of data propagation happened for the given $HeapLoc$ at which $Loc$, whereas $ActionRule$ defines what action should take for the $HeapLoc$. $AssignRule$ allows three operations: 1) ‘=’ strong update for a $HeapLoc$; 2) ‘+=’ weak update for a $HeapLoc$; 3) ‘-‘ kill facts from $RHS$. $ActionRule$ has three $Action$: 1) ‘∼’ clear all heap for $RHS$; 2) ‘source’ mark an $RHS$ as sensitive data; 3). ‘sink’ mark an $RHS$ as a leaky point. $RHS$ consists of $HeapLoc$ or $Instance$ which represents right-hand-side values. $HeapLoc$ is used to represent the heap location which consists of $HeapBase$ and $Index$. There are three types of $HeapBase$ a callee method could use to create heap manipulating side-effect: the heap of arguments, return value and global variables. Depending on the object type of $HeapBase$, field access or array access can be used to present the $Index$. $Instance$ represents the object instance created at particular $Loc$. For example, the $setClass()$ system API call in Figure 2.3 has a summary $\Delta(setClass) = \langle (this.mClass = arg2) \rangle$ where the $this.mClass$ is a $HeapLoc$ which means the $mClass$ field of “this” argument, and $(this.mClass = arg2)$ indicates the $mClass$ field of “this” argument will get whatever value from the second argument.
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent i = new Intent();
        i.setClass(getApplicationContext(), FooActivity.class);
        i.putExtra("key","value");
        startActivity(i);
    }
}

Figure 2.3: Library API call example

Take Figure 2.3 as an example. J7 invokes setClass() to set FooActivity as the target component for Intent i. The setClass() is an Android system API; thus the \( \Delta(\text{setClass}) \) is applied. \( \Delta(\text{setClass}) \) sets a class value obtained from the second argument to its containing Intent's mClass field. J8 inserts a key-value pair ("key," "value") into Intent i's mExtras field. The putExtra() is an Android system API, and the \( \Delta(\text{putExtra}) \) is applied. In this case, the summary tells the static analysis engine to assign the key-value pair to the mExtras field of Intent i.

---

The mExtras field is an aggregate object that may store multiple key-value pairs. Argus-SAF currently does not model such aggregates and instead "flatten" all the elements in an aggregate into singleton instances. This will create two possible interpretations of multiple facts regarding an aggregate object: either they are different possibilities from different program branches, or they are part of a single aggregate in the same branch. Argus-SAF's static analyzer conservatively assumes both are possible to ensure soundness, but this could lose some precision. Modeling aggregates is an engineering work that Argus-SAF will address in future work.
CHAPTER 3: DATA-FLOW ANALYSIS

Determining object points-to information is a core underlying problem in almost all static analyses for Android app security, such as finding information leaks, inferring Intent calls, identifying misuse of specific library functions, and others. Instead of addressing each of these problems using different specialized models and algorithms, it is advantageous to pre-calculate all object points-to information at once and use this as a general framework for different types of further analysis. This way the cost of computing points-to information is amortized across the large number of specialized analyses one will likely need to perform on a given app.

Existing off-the-shelf static analysis tools such as Soot [35] (used by FlowDroid [19, 30] and Epicc [18]) and Wala [39] (used by CHEX [3]) have not provided capability of calculating all objects’ points-to information in both flow and context-sensitive way [45, 46]. This is due to concerns about computation cost. However, with the advancements in hardware (e.g., many-core machines), it opens new possibilities to perform a more precise analysis.

Generally speaking, the core task of Argus-SAF’s analysis is aimed to build a precise inter-procedural data-flow graph (DFG). The flow and context-sensitive data-flow analysis to calculate object points-to information is done at the same time with building inter-procedural control flow graph (ICFG). This is because, for one to precisely know the implementation method of a virtual method invocation, one needs to know the receiver object’s dynamic type; conversely, flow-sensitive data-flow analysis requires one to know how the program control flows. Thus, there is a mutual dependency between the two analyses. Such integrated control and data-flow analyses approach has been demonstrated to be both practical and effective for even analyzing temporal properties of concurrent Java programs including the standard Java library codebase [50]. However, [50] does not keep track of method calling context (typically termed monovariant calling context analysis

---

1This chapter is partially based on the authors CCS 2014 [48] and TOPS 2018 [49] papers. See Appendix A for permission.
or 0-calling context [44]). Argus-SAF generalizes the approach to precisely track the last $k$ calling contexts (polyvariant [44], a.k.a. $k$-limiting where $k$ is user-configurable, and the additional calling context beyond $k$ is monovariant).

3.1 Reaching Facts Analysis (RFA)

Reaching Facts Analysis (RFA) computes points-to facts for each statement. Figure 3.1 illustrates a complete analysis from entry point (EP) method $foo$ using RFA, and below the notations and algorithm is introduced.

```java
public abstract class A0 {
    String f;
    abstract String bar(String ...)
}

public class A1 extends A0 {
    @Override
    String bar(String s) {
        this.f = s;
        return s;
    }
}

public class A2 extends A1 {
    @Override
    String bar(String s) {
        return s;
    }
}

public void foo(B b) {
    A0 a;
    if (tossCoin) {
        a = new A1();
    } else {
        a = new A2();
    }
    String str = "abc";
    a.bar(str);
}
```

Figure 3.1: Building the DFG for $foo$: The intra-procedural control flow graph (CFG) of $foo$ is extended to a callee, $bar$

3.1.1 Notations

There are two sets of facts associated with each statement: the set of facts entering into a statement $s$ is called the entry set of $s$ (or just entry($s$)); the set of facts exiting a statement $s$
is called the exit set of $s$ (or just $\text{exit}(s)$). Statement $s$ may change $\text{entry}(s)$ by killing stale facts ($\text{kill}(s)$) and/or generating new facts ($\text{gen}(s)$). The $\text{gen}$ and $\text{kill}$ sets can be calculated using flow functions that are based on $s$’ semantics. In general, the flow equations have the following forms.

$$\text{exit}(s) = (\text{entry}(s) \setminus \text{kill}(s)) \cup \text{gen}(s)$$ (3.1)

RFA keeps track of points-to facts, which provide information about what objects a variable (register in Dalvik), an object field, or an array element may point to at a particular program point. A points-to fact has the general form of $(\text{lhs}, \text{rhs})$.

The $\text{rhs}$ may refer to either an object or an aggregate (usually key-value pairs). Objects are dynamically allocated in the Dalvik VM heap space at object creation sites (through a “new” statement). In our IR, each statement in the program is assigned a unique location $N$. Argus-SAF uses this to represent the new object created at the location, and refer to it as instance $N$. For example, (in Figure 3.1) location $J4$ generates the points-to fact $\langle a, J4 \rangle$. Here $J4$ represents instance $J4$, the object created at location $J6$. From the object creation site, Argus-SAF can directly find the precise runtime type of the instance.

Let us use $\Box N$ to indicate any possible value that is type compatible with the received objects at location $N$. For instance, Argus-SAF does not know the possible values that will be received for an $\text{EP}$. As an example, location $J1$ generates a points-to fact $\langle b, \Box J1 \rangle$, indicating that the object variable $b$ points to an object that is passed to $\text{EP}$ at location $J1$.

There are two types of $\text{lhs}$ of a points-to fact, yielding two types of facts. A variable-fact is when the $\text{lhs}$ is a variable. A heap-fact is when the $\text{lhs}$ is an object field or an array element. For example, location $J18$ generates a heap-fact $\langle (J4, f), (J8) \rangle$, meaning that the field $f$ of instance $J4$ points to the string “abc” created at $J8$.

### 3.1.2 The Basic DFG Building Process

A static analyzer simulates the program and keeps track of the fact sets until a fixed point is reached. The convergence to a fixed point (analysis termination) is guaranteed as long as the flow equations are monotone, and the number of facts is finite, which hold for Argus-SAF’s analysis. For a given app, it contains a finite number of object creation sites and variables/fields (and as
typically done, elements of an array are summarized as one); moreover, Argus-SAF keeps track of calling contexts up to a finite number $k$.

Argus-SAF builds the $DFG$ by flowing the points-to facts from the program’s entry points. Here the program is the IR of the app’s dex code augmented with the environment methods as discussed in Section 2.1. Unlike Java applications, there is no “main” method in an Android app; every component could be the starting point of an app. Our component-based environment model captures the full life cycle of a component and all of its possible execution paths, including those due to interacting with other components. Thus, if we assume one particular execution path starts from component $C$, we can use $C$’s environment method $E_C$ as the program’s entry point. To include all possible execution paths from all possible components, Argus-SAF does this for every component in the app, yielding multiple $DFGs$. Formally, let $C$ be a component, the $DFG$ from $C$ is denoted $DFG(E_C)$ where $E_C$ is the environment method of $C$ and is a tuple defined as the following.

$$DFG(E_C) \equiv ((N, E), \{entry(n) \mid n \in N\}),$$  \hspace{1cm} (3.2)

where $N$ and $E$ are the nodes and edges of the inter-procedural control flow graph starting from $E_C$ (denoted $ICFG(E_C)$). $entry(n)$ is the entry set of the statement associated with node $n$. Each $DFG(E_C)$ captures the execution that starts from component $C$ and may involve other components due to ICC. Each statement node is annotated with the statement entry set (the exit set is not shown for presentation sake). In this example, Argus-SAF starts building the $DFG$ from the entry point method $foo$ with an empty fact set. Argus-SAF then simulates the program statically based on each statement’s semantics and transforms the fact sets along the way based on the flow equation (3.1).

As Figure 3.1 illustrated, at a control-flow join point, the exit fact sets from all incoming edges are unioned (e.g., at $J\overline{8}$); facts such as $\langle a, J4 \rangle$ and $\langle a, J6 \rangle$ coming from the different branches accumulate in $entry(J\overline{8})$. Similarly, one can compute $entry(J9)$. At this point, Argus-SAF needs to resolve the target for $J9$’s virtual method invocation with static type $A0$. The first argument of the call instruction, $a$, is the receiver object. Since we now have calculated the possible points-to values of $a$ — $instance J4$ or $instance J6$, Argus-SAF can resolve the possible call targets precisely:
**Algorithm 2 Building Data-flow Graph (DFG)**

**Require:** The entry point procedure, \( EP \).

**Ensure:** \( DFG(EP) \)

1: procedure BuildDfg(\( EP \))
2: \( icfg \equiv (N,E) \leftarrow \) empty graph;
3: \( addCFG(icfg,CFG(EP)) \);
4: \( \iota \leftarrow \) initial fact set;
5: \( entry \leftarrow \) emptyMap;
6: worklist \( \leftarrow \) emptyList;
7: \( entry(EntryNode\_EP) \leftarrow \iota \);
8: worklist \( \leftarrow \) worklist :: EntryNode\_EP;
9: while worklist \( \neq \) empty do
10: \( n \leftarrow \) get (and deque) head from worklist;
11: \( nodes \leftarrow processNode(icfg,n) \);
12: worklist \( \leftarrow \) worklist :: nodes;
13: end while
14: return (\( icfg, entry \));
15: end procedure

\( A1.bar \) for instance \( J4 \) and \( A2.bar \) for instance \( J6 \) (because both \( A1 \) and \( A2 \) override \( A0.bar \)). This shows the advantage of doing a precise points-to analysis concurrently with \( ICFG \) building — not only can Argus-SA F has more precise information on the call targets, but also it allows Argus-SAF to flow more accurate facts to the different call targets. All of these increase the precision and can potentially reduce the number of false alarms in the analysis results.

As shown in Figure 3.1, a call statement contributes a pair of \( CallNode \) and \( ReturnNode \) to the \( ICFG \). The \( CallNode \) connects to the callee’s \( EntryNode \) while the callee’s \( ExitNode \) connects to the \( ReturnNode \). In transferring facts between the caller and the callee, the variable-facts need to be remapped to the formal parameters of the callee (e.g., \( str \) in the caller maps to \( s \) in the callee). This should be restored when the control returns to the caller. Only heap-facts reachable from the call parameters are passed to the callee. The unreachable heap-facts, as well as unrelated variable-facts, are transferred to the \( ReturnNode \) directly to improve efficiency. In the example of \( J9 \)’s method invocation, there is one variable-fact \( (b, \Box J1) \) which is unrelated to both arguments \( a \) and \( str \). The flow of such fact (which is unrelated to any callee) is represented as a double-head arrow from the \( CallNode \) to the \( ReturnNode \). Similarly, there can be some facts at the callee side that are unrelated to the caller (e.g., callee’s local variables and temporary objects), and Argus-SAF filters them out at the callee’s \( ExitNode \) to improve efficiency.

Consider the data-flow analysis for \( A1.bar \) or \( A2.bar \), which is a callee for \( J9 \)’s method invocation. Argus-SAF tracks the \textit{entry} of each statement of \( A1.bar \) (or \( A2.bar \)). The author
Algorithm 3 \textit{processNode}: Pushing facts to successors

\begin{verbatim}
Require: ICFG, icfg \equiv (N, E) and a node, n \in N
Ensure: n’s successor nodes whose entry are updated.

1: procedure \textsc{processNode}(icfg, n)
2:    tempList \leftarrow empty;
3:    if n is an EntryNode or a ReturnNode then
4:       for all p \in successors(n) do
5:          entry(p) \leftarrow entry(p) \cup entry(n);
6:          tempList \leftarrow tempList :: p;
7:       end for
8:    else if n is an ExitNode then
9:       for all p \in successors(n) do
10:          passRequiredFactsToCaller(n, p);
11:          if p gets any new fact then
12:             tempList \leftarrow tempList :: p;
13:       end if
14:    end if
15:    else if n is a CallNode or a RegularNode then
16:       if visit(icfg, n) = true then
17:          tempList \leftarrow tempList :: successors(n);
18:       end if
19:    end if
20:    return tempList;

21: end procedure

22: procedure visit(icfg, n)
23:    if n is a CallNode then
24:       (fMapForCs, factsToR) \leftarrow reslovCall(icfg, n);
25:       update callees’ EntryNodes with fMapForCs;
26:       update ReturnNode(n) with factsToR;
27:    else if n is a RegularNode then
28:       for all p \in successors(n) do
29:          entry(p) \leftarrow entry(p) \cup exit(n);
30:       end for
31:    end if
32:    if any p \in successors(n) gets any new fact then
33:       return true;
34:    end if
35:    return false;
36: end procedure

37: procedure reslovCall(icfg, n)
38:    calleeSet \leftarrow getCallees(entry(n), callSig(n));
39:    for all M \in calleeSet do
40:       if (EntryNode\_M \notin N) then
41:          addCFG(icfg, CFG(M));
42:          E \leftarrow E \cup (n, EntryNode\_M);
43:          E \leftarrow E \cup (ExitNode\_M, ReturnNode(n));
44:       end if
45:    end for
46:    fToCallees \leftarrow empty;
47:    factsMapForCallees \leftarrow emptyMap;
48:    for all p \in successors(n) do
49:       factsToCallers \leftarrow filterFunc(n, p, entry(n));
50:       factsMapForCallees(p) \leftarrow factsToCallers;
51:       fToCallees \leftarrow fToCallees \cup factsToCallers;
52:    end for
53:    factsToReturn \leftarrow exit(n) \setminus fToCallees;
54:    return (factsMapForCallees, factsToReturn);
55: end procedure
\end{verbatim}
observes that entry(Return J9) contains heap-facts which show that field f of Instance J4 points to the String “abc” at J8. This is the effect of J18. It is interesting to see that this is not true for the same field (i.e., f) of Instance J6 because no assignment like J18 happens inside A2.bar.

Now, we can get entry(J10), and continue to process until reaching the ExitNode.

The algorithm for the DFG building process is formally presented as Algorithm 2. This is a fixed-point algorithm (ref. the while loop from L9 to L13), which tracks what points-to facts reach each statement from the given entry point (EP). The core of Algorithm 2 is L11, which processes different type of nodes in the control flow graph, and this is formally elaborated in Algorithm 3. Algorithm 3 presents how to process each type of node (e.g., CallNode, ReturnNode, etc.). As an example, if it’s a CallNode, the ICFG will be expanded by including the callee graph based on the points-to facts flowing there. Algorithm 2 computes all the possible object point-to information for each program points; therefore it is sound with respect to conservatively model the side-effect could be introduced by library API calls.

3.1.3 Handle Library APIs

An app can call large number of library APIs (system or third-party), and it is not feasible to track the flow facts inside all those APIs. Argus-SAF modeled thousands of library APIs to cover critical data flows using the summary language discussed in Section 2.2.

3.2 Summary-based Bottom-up Data-flow Analysis (SBDA)

Android allows a developer to design a part or the complete app using native language (C/C++) and allows Java code to communicate with native code bi-directionally using Java Native Interface (JNI). A comprehensive static analysis framework needs to be able to analyze both languages and address there inter-language communication channels. However, there exists significant challenges for RFA to work in such an inter-language analysis setup:

1) Difference in intermediate data representation: Java data-flow analysis typically tracks points-to facts, whereas binary data-flow analysis typically uses symbolic execution. Thus the two analysis engines use different data representations in the analysis process, making it hard to integrate. How to design a unified data-flow representation for both analyses is a challenge.
2) Efficiency: Both Java data-flow analysis and binary symbolic execution are computationally expensive. The traditional data-flow analysis requires propagating data-flow facts continuously over the complete program’s control flow graph until a fixed point is reached. For inter-language analysis, this means the analysis process need to switch between the java and binary analysis context continually. This further exacerbates analysis time.

To address above challenges, Argus-SAF adopt the Summary-based Bottom-up data-flow Analysis (SBDA) algorithm introduced in [51]. The benefit of this method is that Argus-SAF only needs to visit each method exactly once to generate a unified heap manipulation summary for both Java and native procedures, while still preserving a flow and context-sensitive data-flow analysis result.

Figure 3.2 illustrates the workflow of SBDA. It takes the environment method as EP and generates a call graph G from it. From G Argus-SAF applies a topological sort algorithm with the reverse order to get a list of method MList, which guarantees the callee method always comes before the caller method. If there is a cycle in the call graph, the algorithm will break the cycle.
arbitrarily to make sure the topological sort will always hold. For each method $M_i$ in $MList$, Argus-SAF applies a heap manipulation summary generation algorithm to get summary $\Delta_i$. The $\Delta_i$ is represented using the same DSL language as illustrated in Section 2.2.1. The callee method’s summary will propagate to its caller methods until the $EP$ is reached.

As discussed in [51], the soundness of a summary-based analysis is guaranteed if the summary $\Delta$ over-approximates the heap manipulation side-effects. Therefore, $SBDA$ is sound as long as the summary generation process is conservative, which means conservative consideration of input data and conservative model of library API calls.

Let’s take Figure 3.3 as an example to walkthrough the heap manipulation summary generation process and how to leverage the summary $\Delta$ to resolve the data-flow problem for the motivating example. Start from method $ep()$ Argus-SAF builds a Call Graph, and topological
sort it in reverse order. For building the Call Graph, Argus-SAF needs to address the native method call from Java code and Java method call (reflection style) from Native code, respectively. The details are discussed in Chapter 5. Argus-SAF starts generating the summary $\Delta$ from the leaf function $n_2()$. Native function $n_2()$ leaks the first argument\(^2\) thus Argus-SAF generate a summary $\Delta(n_2) = \langle (\text{sink}\(\text{arg1}\)\text{C15}) \rangle$ and propagate it to java method $\text{bar()}$. $\text{bar()}$ pass the first argument to $n_2()$, and the $\Delta(n_2)$ is applied. Therefore, Argus-SAF gets summary $\Delta(\text{bar}) = \langle (\text{sink}\(\text{arg1}\)\text{C15}) \rangle$ and propagate it to native function $n_1()$. $n_1()$ read the str field from first argument $d$ and invokes method $\text{bar()}$. Therefore, $\Delta(\text{bar})$ is applied and Argus-SAF gets summary $\Delta(n_1) = \langle (\text{sink}\(\text{arg1}.\text{str}\)\text{C15}) \rangle$. $\text{foo()}$ puts second argument $\text{imei}$ into the str field of first argument $d$ and invokes native function $n_1()$. Argus-SAF applies $\Delta(n_1)$ and then get $\Delta(\text{foo}) = \langle (\text{arg1}.\text{str} = \text{arg2})(\text{sink}\(\text{arg1}.\text{str}\)\text{C15}) \rangle$. Java method $\text{ep()}$ assigns a sensitive data to variable $\text{imei}$ at J17 and creates a Data instance to $d$ at J18. J19 of java method $\text{ep()}$ invokes method $\text{foo()}$. $\Delta(\text{foo})$ tells us the str field of variable $d$ gets data in variable $\text{imei}$ which is sensitive, and this str field of variable $d$ will flow to a leak point at C15. Therefore, Argus-SAF captures the data leakage problem.

### 3.3 Discussion

Currently, both data-flow analysis algorithms only do constant propagation for string values and have a conservative model for string operations. This limitation could introduce imprecision for the intent resolution and reflection call handling. Precise and general string analysis in static analysis is non-trivial, and the author leaves this for future research. For example, prior research [52, 53] could be applied. Moreover, recent work [38] shows that ICC resolution can benefit from domain knowledge and probabilistic models, which Argus-SAF could adapt to prioritize inferred ICC destination choices.

$\text{RFA}$ does not currently handle Java reflection, dynamic class loading. Adding preliminary support for reflections and dynamic class loading is similar to handling ICC in Argus-SAF. More-

\(^2\)First two arguments of native functions are not counted in the summary as env is not presented in java method and obj is “this”.

25
over, Li et al. [54] have proposed ways to handle Java Reflection and dynamic class loading in a reliable way, which might be able to leverage in the future.

Both data-flow analysis algorithms’ data and control flow analysis depend on the faithfulness of the models, including the models of the Android environment and its APIs. I designed a DSL as discussed in Section 2.2 to let researchers more easily model library APIs for their analysis purpose. Currently, there exist more than 1,000 API models using the DSL, which covers a significant portion of the real-world Android API usage. However, due to the size of the Android library and complexity of third-party libraries, it remains a challenge to reliably detect all library API and provide a precise and sound model for them. Recent work [55,56,57] offer approaches to detect third-party libraries. Prior work [58] shows that static analysis can compute accurate data-flow summary for the Android framework. They might be able to leverage in the future.
CHAPTER 4: COMPONENT-BASED ANALYSIS

An Android app might have multiple components while the components can communicate with each other via various channels: Intent, RPC, static field. Thus, security sensitive data items can also flow through these channels. Moreover, in an inter-app communication, one component of app X interacts with one component of app Y; hence, communication across different apps can be considered as inter-component communication. Thus, our approach considers the component-based analysis as the basic building block for app vetting. Argus-SAF does both intra- and inter-component analysis (covering both intra-app and inter-app analysis, if necessary). Our analysis approach consists of the following phases:

1) Build data-flow graph (DFG) for each component (discussed in Section 4.2).

2) Build data dependency graph (DDG) for each component (discussed in Section 4.3).

3) Perform inter-component analysis (discussed in Section 4.4, 4.5, and 4.6).

4.1 Motivating Example

A malicious app can conduct bad behaviors by leveraging the design (e.g., event-driven and inter-component nature) of Android system and try to obfuscate its real objectives. Figure 4.1 shows an example app (named “inter-component-leak”), which consists of a few components while each one is a separate Java class. Android apps are component-based where each component is an independent entity and is typically responsible for a specific task. For instance, an Activity component implements the UI of the app, a Service component typically performs a long-running task on the background, and a Broadcast Receiver component receives a broadcast message from one component (or the system) and takes specific actions and more.

---

1This chapter is based on the authors TOPS 2018 [49] paper. See Appendix A for permission.
An Android app does not have a “main” method; instead, components are invoked through the various callback methods (including lifecycle methods). Depending on the events, the system invokes the lifecycle methods of the components. It also remembers the recently sent intents and passes them around, which can be abstracted in a component-level environment. Furthermore, there can be control flow and data-flow among the app components through the Android system. For a comprehensive analysis, the app analyzer tool needs to track such control and data flows.

As an example, the following sequence of events (as labeled in Figure 4.1) can happen in reality:

1) FooActivity starts BarActivity (via “startActivityForResult” API) and waits for BarActivity to send back some result.

2) When the user clicks on a button of BarActivity screen, the onClick method is triggered.

3) BarActivity makes an RPC (Remote Procedure Call) call getImei() to a Service component named MyService, and MyService returns an inner field (which has already possibly stored the IMEI Id) to BarActivity.

4) BarActivity sends back an intent (via setResult API), which contains the IMEI Id.

5) Android system invokes the onActivityResult method of FooActivity with the above intent as a parameter, and the IMEI Id is extracted and leaked (to the attacker) through an SMS message.

To track the control and data-flow inside a component, a static analyzer needs a model of the Android system to track invocation of the callback methods including the component lifecycle methods as illustrated in the above example. Our model of the Android environment is inspired by FlowDroid [19,30], which uses a “dummyMain” method to capture all possible sequences of lifecycle method invocations as followed in Android. However, unlike an app-level environment model used in FlowDroid, Argus-SAF designs a component-level environment model. The motivation behind the component-level model choice is that Android apps work in this way.

Furthermore, Argus-SAF needs to track data and control flow through each type of inter-component communication channel (such as Intent, RPC). As an example, when BarActivity sends
Figure 4.1: The inter-component-leak App: The arrowed lines among the app components highlight some of the inter-component communication.
out an intent $i3$ via $\text{setResult()}$ API, the Android system invokes the $\text{onActivityResult}$ method of $\text{FooActivity}$ with $i3$ (i.e., $\text{data} = i3$) as a parameter. The reason for the above action is that $\text{FooActivity}$ has started $\text{BarActivity}$ before with the $\text{startActivityForResults()}$ API. To track the control and data flow involved in such a “stateful” ICC (inter-component communication) mechanism, the analyzer tool needs to remember which Activity has started a given Activity $A$. Another challenge for the analyzer tool is how to track the RPC channel if any. As an example, when $\text{BarActivity}$ invokes the $\text{getImei()}$ method, the analyzer tool has to map the call to the corresponding method of $\text{MyService}$ component. $\text{BarActivity}$ receives some data flow as the return from the call. Furthermore, $\text{MyService}$ might have been running already before this RPC takes place and has stored the IMEI Id in field $\text{imei1}$ (e.g., because another RPC method $\text{setImei()}$ got invoked by others), and the $\text{getImei()}$ call returns the sensitive information from $\text{imei1}$ to $\text{BarActivity}$. This shows that the analyzer tool needs to address the re-entry nature of the component code. In addition to the above channels of communication among app components, two components can also exchange data via static variables and more. So, the app analyzer tool needs to track these channels too.

4.2 Building the Component-level Data-flow Graph

Argus-SAF computes points-to facts for each statement. In the component-based analysis, Argus-SAF builds the $\text{DFG}$ of each component of an app follow algorithm introduced in Section 3.1.

Figure 4.2 illustrates part of the resulting $\text{DFGs}$ of the components in the example app. During the intra-component analysis phase, one cannot tell what data will be received by this component from others through inter-component channels, such as Intent, RPC, static field. Thus, at any information retrieval point for those channels, a conservative model like that used in Section 3.1.3 is applied. More detailed discussion on how to handle data flows across components will be discussed in Section 4.4.

4.3 Building the Component-level Data Dependence Graph

A component-level data dependence graph ($\text{DDG}$) is derived from the component’s $\text{DFG}$. With the help of $\text{DDG}$, Argus-SAF can determine which part(s) of the program a particular program
Figure 4.2: *DFGs and STs of the components in App “inter-component-leak”: An excerpt*
point depends on. \textit{DDG} is a directional graph; its node set is the same as the nodes in \textit{DFG} and has two types of edges:

1) Object dependence edge: Linking the use site of an instance to the creation site of the instance.

2) Variable def-use edge: Linking a use site of a variable to the def-site of the variable.

Since object flows in a component are captured in \textit{DFG}, the constructed \textit{DDG} automatically captures data dependencies within the component boundary. As an example, in Figure 4.2, the \textit{J14} in \textit{FooActivity} uses \texttt{imei3} while the entry of statement \textit{J14} has a fact \langle \texttt{imei3}, □J12 \rangle. This tells us that the object □J12 (generated at J12) is used in statement J14. Thus, there is a data dependency path from J14 of the \textit{FooActivity} to the def-site J12 in the same component.

4.4 \textbf{Linking Inter-component Data Flows}

When components interact through Inter-component communication (ICC) channels, the data-flow facts will propagate from one component to another. There are a couple of challenges in analyzing inter-component data flows for Android apps.

1) Android app components run concurrently, and their execution sequence can be arbitrarily interleaving or parallel depending on the events that trigger the various call-back methods.

2) Android app components are stateful. After component \textit{A} invokes ICC on component \textit{C} and changes its state, another component \textit{B} may invoke ICC on \textit{C} later and be impacted by the effect of the previous ICC from \textit{A}.

Figure 4.3a shows a case where a Service \textit{C} has a field \texttt{f} and two RPC methods \texttt{set()} and \texttt{get()} which set and get data from field \texttt{f}, respectively. These two RPC methods can be invoked in any order with any data from all other components. For example, component \textit{A} may set a sensitive data into Service \textit{C}'s field \texttt{f}, and component \textit{B} could retrieve such data from \textit{C} via the \texttt{get()} RPC call later, forming an information flow path. Figure 4.3b shows another case where component \textit{A}, \textit{B} share data via static field \texttt{X.f}, which can form an information flow path from \textit{A} to \textit{B}.

Traditional context-sensitive call graph generation cannot capture this type of information flow from “stateful ICC.” In the above example, neither the call sequence \textit{A} → \textit{C} nor \textit{B} → \textit{C} can
Figure 4.3: Data flow between app components via RPC and static field

capture the information flow $A \rightarrow C \rightarrow B$. The information flow only happens through interleaving
the three components’ execution in the order \{A, C, B, C\}, where the first two captures the RPC
call $A \rightarrow C$ and the latter captures the RPC call $B \rightarrow C$. Such concurrency execution semantics
can be modeled by treating ICC in a context-insensitive manner and merging all the data-flow facts
at a component’s ICC entry point – simulating the effect of all possible orders of interleaving.

Based on this idea, one approach is to compute a global fixed-point among all the components
while flowing the points-to facts context-insensitively between components (intra-component data-flow
is still context-sensitive).\(^2\) The downside is that for any new set of components need to analyze,
Argus-SAF would have to re-compute the global fixed-point, making it impossible to re-use the
per-component analysis result. Thus, a different approach is adapted. When computing the $DFG$
for each component in the intra-component analysis phase, Argus-SAF assumes that any type-
compatible data is possible to enter the ICC channels. Besides, Argus-SAF book-keeps all the data
that may enter and leaves the component through the channels. In the inter-component analysis
phase, Argus-SAF then “stitch” the inter-component communication channels’ receive points with

\(^2\)It is quite non-trivial to compute this global fixed-point while at the same time simulating the non-determinism
caused by the interleaving concurrent threads [50].
the corresponding send points (between two different components), forming the inter-component data dependence graph. This conservative approximation serves the purpose of our goal well with following reasons:

1) Android is a component-based system, and any component may receive data from any other component – not necessarily the ones in the same app; thus assuming any type-compatible data may come into the ICC channel is consistent with Android’s execution semantics.

2) This reasoning model obviates the need for computing ICC call graphs, thus eliminates the call graph explosion problem that may happen in other Android analysis tools, including the original version of Amandroid [48].

3) By analyzing each component separately, it allows us to re-use the intra-component analysis result for any further inter-component analysis, possibly involving different subsets of the components. This will scale better with large volumes of apps and naturally extends to inter-app analysis.

In the inter-component analysis phase, the DFG of all the involved components is loaded. Based on the ICC channel book-keeping information Argus-SAF then finds the data dependence between the sender and recipient points. The book-keeping information is stored in a data structure called the summary table (ST). Argus-SAF generates an ST for each component $C$ via processing $C$’s DFG, where ST lists the communication channels through which $C$ communicates with other components. ST records specification of different types of channels including, e.g., Intent, RPC, and static fields\(^3\). In particular, for each such channel, the ST of $C$ records the following items:

1) **send-points** where $C$ is the sender of the channel. The information recorded includes what kind of data is sent (e.g., outgoing Intent value for an Intent channel) and the receiver’s name.

2) **receive-points** where component $C$ is the receiver of the channel. The recorded information includes receiver’s name which allows matching with other components’ send-points. For

---

\(^3\)Files can serve as an inter-component communication channel like static fields, and can be handled in a similar way. This would require a precise string value solver, which will be handled in future work.
example, for Intent channel, the intent filter value; for RPC channel, the RPC method’s signature, and so on. Table 4.1 lists the main items in an ST.

Table 4.1: Communication points of an app component

<table>
<thead>
<tr>
<th>Channel</th>
<th>Send-points</th>
<th>Receive-points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent</td>
<td>Outgoing Intent</td>
<td>Intent Filter</td>
</tr>
<tr>
<td>RPC</td>
<td>Method signature, params, return</td>
<td>Method signature, params, return</td>
</tr>
<tr>
<td>Static Field</td>
<td>Field signature to write, data</td>
<td>Field signature to read</td>
</tr>
</tbody>
</table>

Figure 4.3 helps to discuss how the STs are constructed and used. There are three components in Figure 4.3a, whose DFG has already been built. In component A, we saw an RPC call C.set(d) that sends data d to Service C via the RPC channel C.set(). Argus-SAF adds this to the RPC channel’s send-point description in A’s ST. Component B has an RPC call C.get() which sends a request to Service C and expects a return value from it. Argus-SAF adds it to both the send-point and receive-point description of B’s RPC channel. Service C has two RPC methods C.set(x) and C.get(); Argus-SAF adds them to the receive-point of C’s RPC channel. C.get() is returning an object to its caller; Argus-SAF adds it to the send-point of C’s RPC channel. Figure 4.3b shows the inter-component communication caused by the static field. Here the send-point description indicates a write to the static field, and a receive-point description indicates a read from the field. With the STs for each component constructed, Argus-SAF can “stitch” the send and receive points of the channels between two components to identify all possible inter-component data dependency. The “stitching” process is matching each channel’s send-point with receive-point between two components based on channel specific criteria. For example, in Figure 4.3a Argus-SAF can stitch component A’s send-point 1 to component C’s receive-point 1 because their method signatures match. After “stitch” all the send-points and receive-points (the arrows shown in Figure 4.3a), we can easily see the information flow path from d in component A to leak in component B.

In the next three subsections, the ST construction and this “stitching” process for each type of ICC channels is further discussed.
4.4.1 Intent

4.4.1.1 ST Construction

Section 4.1 illustrates that malicious apps can easily manipulate Android’s inter-component communication (ICC) to leak sensitive data stealthily. To track data-flow through the Intent channel, Argus-SAF needs to solve statically specific values for the intent involved. At a send-point, Argus-SAF needs to solve for the Intent call parameters to infer the value of the outgoing Intent so that Argus-SAF can match it with the correct receive-points. At the receive-point, Argus-SAF needs to discover the Intent filter value so Argus-SAF can match it with the possible send-points. Argus-SAF infers the Intent API call parameters and Intent filters using the points-to facts computed and the app manifest file. This information will enable us to discover the source-destination component pair of the Intent call in the inter-component analysis phase.

The destination of an Intent can be either explicitly or implicitly specified in the outgoing intent. The standard way of creating an explicit intent is by adding the destination component’s name using Android APIs such as `setClass` (J7 in Figure 4.2). For instance, at J8 in Figure 4.2 Argus-SAF can derive that the intent parameter `i1`’s field `mComponentName` is “BarActivity.” This fact comes from the modeling of the API function `setClass` called at J7, which generates a field-fact `⟨(J6, mComponentName), “BarActivity”⟩`, where J6 represents Intent `i1` which was created at J6. Argus-SAF records the destination component name as a send-point in ST. Also, Argus-SAF documents in ST whether the Intent caller expects a result is returning later from the callee component (in case of stateful Intent call like “startActivityForResult” as opposed to stateless Intent call like “startActivity,” “bindService”).

An implicit intent does not include the name of a specific destination component but instead requests a general action to perform, and the System finds a capable component (from the same app or another) which can fulfill the request. Some fields of an Intent object are used in this matching: `mAction` (String), `mCategories` (set of String), `mData` (Uri), and `mType` (String). These intent fields can be manipulated by invoking certain Android APIs. For instance, `i.setData(Uri.parse(http://abc.com/xyz))`, which sets the Uri corresponding to a http url to the `mData` field of an Intent `i`. Through proper modeling of these API functions (Section 3.1.3), Argus-SAF can derive possible
(String) values of the relevant fields of an Intent object, which the Android system bases its decision on Intent destinations. Argus-SAF documents these fields of the Intent as send-points in ST.

### 4.4.1.2 Stitching Intent Channels – Intent Destination Resolution

For explicit intents, it is straightforward to find the correspondence between the source component and the destination component. The matching information is directly available as the send-point (in the ST) of the source component and as the receive-point (in the ST) of the destination component. For example, FooActivity has a send-point at J8 (startActivityForResult()) where Intent i1 has the target component name set to “BarActivity,” which matches the receive-point in the ST of BarActivity. Hence Argus-SAF discovers the correspondence.

However, tracking the “return” intent j sent by the callee component X in a stateful Intent is more complicated, e.g., the name of the destination component of the intent i3 sent through the “setResult” API as in J40 of BarActivity is not available in the app code (neither in the ST of BarActivity). To know the possible destinations of intent j, Argus-SAF first checks through all components’ ST to find each component Y which have initiated a stateful Intent call (i.e., startActivityForResult) to component X (e.g., BarActivity). Then, Argus-SAF infers that onActivityResult API of each component Y is receiving intent j as a parameter.

Furthermore, there are some challenges in resolving the target of an implicit intent. The Android system finds the destination based on the intent fields as well as the manifests of all the apps which specify intentfilters for a component. An intentfilter is an XML expression involving the action tag, category tag, and data tag (which includes both Uri and type). The Android system determines the destination of an implicit intent by applying a set of rules [59] matching the relevant intent fields and the intent filter specification for every component of the system. Argus-SAF implements all those matching rules, using the static analysis results that show the possible string values of the relevant intent object fields. It runs a precise actiontest, categorytest, and datatest (having both Uri and type) to find the destination component(s). Our static analysis can readily handle Intent fields. For complicated String operations (e.g., concatenation in a while loop), if Argus-SAF cannot infer the exact string value, it reports it as any string, ensuring the soundness of our analysis. Argus-SAF can run the Uri test matching different parts of the Uri (e.g., scheme,
path, host, port) between the intent and an intent filter. Furthermore, Argus-SAF is also able to find the specifications of dynamically registered Broadcast Receivers, if any.

4.4.2 RPC

4.4.2.1 ST Construction

Service provides the programming interface that a client component can use to interact with. This allows a client component to send/receive data to/from the service via an RPC call. In the example app of Figure 4.1, MyService defines an inner class MyBinder which extends the Binder class and returns such a Binder instance in onBind() lifecycle method. MyBinder returns handle of MyService which exposes two RPC methods, MyService.setImei() and MyService.getImei(). BarActivity binds to MyService at J25 which uses a ServiceConnection defined at J45. After the bind succeeds, it will set the above handle to the s field of BarActivity. At J37 when a user clicks on a button at BarActivity, it will invoke the RPC call of MyService.getImei() to retrieve data from MyService.

Fortunately, in static analysis, discovering the above RPC connection between two components (intra-app, or Local Service) is straightforward. After resolve bindService() call at J25, Argus-SAF knows the target service is MyService. Then at J37, Argus-SAF knows the target method’s signature is MyService.getImei(). In addition to the Local Service (intra-app) case above, there are two more cases, Messenger Service and AIDL (a.k.a. Remote Service), which allows both intra- and inter- app RPC calls. For Messenger Service case, Argus-SAF first infers the Handler type registered to the Messenger instance that used at the service side and marks the Handler’s handleMessage() as the RPC callee. At the client side, Argus-SAF marks the invocation of Messenger.send() as the RPC caller. AIDL case is like the local service case, Argus-SAF can resolve the bindService() call to find target service and then find the RPC callee. For both the caller component and the callee component, Argus-SAF documents the RPC method signature, parameters, return variable (some as send-points and some as receive-points) in ST.
4.4.2.2 Stitching RPC Channels

Argus-SAF first evaluates Intent channel of ST to find the binding relation between client component and service component. Then, based on the binding relation to match the RPC caller and callee. For Local Service and AIDL case, Argus-SAF matches the call signatures to link the RPC caller and RPC callees. For Messenger Service, Argus-SAF matches the Messenger.send() to Handler.handleMessage().

4.4.3 Static Field

4.4.3.1 ST Construction

Documenting static field is straightforward as each static field has its unique name. In our ST, Argus-SAF needs to record from which program point which static field is read (receive-point) or written to (send-point).

4.4.3.2 Stitching Static Field Channels

Argus-SAF needs to match the static field’s name at send-point and receive-point to make the connection.

4.5 Building App-level Data Dependence Graph

After figuring out all the channel matchings, Argus-SAF connects the data dependency links among components’ DDGs to build an app-level DDG. The time complexity of this stitching process is in the worst case quadratic to the number of components being analyzed. Then Argus-SAF can perform data dependency analysis of the app. For instance, to query the data leakage on the example app in Figure 4.2, Argus-SAF can find a taint source at MyService.setImei() method – any other component can use this RPC call to set the phone IMEI to the MyService.imei field. Then at the MyService.getImei() RPC method the return point can get IMEI and return to J39 at BarActivity; then it puts this information into Intent i3’s mExtra field, and at J40 sends as a result Intent to the caller component FooActivity. At FooActivity.onActivityResult(), J6 extracts IMEI and sends it out via sendTextMessage(), which is a sink point.
Since $DDG$ is a directed acyclic graph, the complexity of shortest path finding is linear to the number of nodes and edges, which in the worst case is quadratic to the size of all the components combined. Thus, even if over-approximation in the Intent destination resolution resulted in spurious data dependence paths, it will not have a substantial impact on the running time.

### 4.6 Inter-app Analysis

Inter-app communication is inter-component communication which passes control and data across app boundaries. Thus, our component-based analysis can be directly used to perform the inter-app analysis. However, it has some challenges.

1) Only a subset of ICC channels can be used for inter-app communication. For example, local service does not support another app to bind to it; static field only allows the same app to read and write as they run in the same JVM.

2) Multiple apps may share the same package and class name which can cause trouble for static analysis tool if it is not aware of the different app contexts.

To address challenge (1)), Argus-SAF manages different scopes for different ICC channels. When linking the inter-component data dependence, it knows which channel can across app boundary. To address challenge (2)), Argus-SAF uses different class loaders for different apps, and in the stitching phase, it adds origin information for each program point to avoid any naming conflict.

### 4.7 Using Component-based Approach for Security Analyses

Argus-SAF provides an abstraction of the app’s behavior in the forms of $DFGs$ and $DDGs$. They can be easily used for a number of useful security analyses as discussed below.

#### 4.7.1 Data Leak Detection

A critical problem in app vetting is to find whether an app may leak any sensitive data. Examples of sensitive data include user-login credentials (e.g., password), location information, and so on. This can be performed through standard data dependence analysis using the $DDG$. Given a source and a sink, one can find whether there is a path from source to sink in the $DDG$. For instance, prior research [30,60] has documented a list of security-critical source and sink APIs,
which can be used here. One could also customize the definition of the source and sink for the specific problem at hand. DDG can only capture explicit information leaks. For information leaks through controls (e.g., leaking conditionals through the branches) one would need to build a control dependence graph, which can be obtained from the DFGs through the standard process [61].

Argus-SAF can perform a comprehensive analysis since it captures control and data flows across the component boundaries through the Intent channel, RPC channel, and others so that security problems like the one shown in Figure 4.1 can be captured.

### 4.7.2 Data Injection Detection

An app can have a vulnerability which allows an attacker to inject data into some internal data structures, leading to security problems. Researchers [3] identified a subclass of this vulnerability called intent injection. The attacker can send an ill-crafted intent to a public component of a vulnerable app, which retrieves data from the incoming intent and uses it for security-sensitive operations. For instance, the app’s logic can be such that the incoming intent determines the destination of critical data flow — the URL of a backup server, the name of a file, the destination component of an ICC call, phone number of an outgoing SMS, or others. As a result, the attacker will be able to control the destination, which can lead to serious security problems.

Argus-SAF can detect this vulnerability using the DDG, by defining the source as the possible entry point of attacker-controlled data (e.g., a public-facing interface), and the sink is the critical parameters of the security-sensitive operations. If a data-dependency path exists between the source and the sink, the attacker can potentially manipulate the parameters of the security-sensitive operations.

### 4.7.3 Detecting Misuse of APIs

Another critical part of security vetting is to find if the developer (intentionally or unintentionally) has inappropriately used a library API, which may lead to security problems. Past research has applied static analysis to identify misuse of Crypto APIs [28] and SSL APIs [29]. The main idea is to detect if the app satisfies a set of rules on the proper use of the APIs. For example, if the parameters for calling the AES encryption method have certain values, the cipher will run in
the insecure ECB mode. Argus-SAF can verify these rules by checking the possible values of the parameter objects in a relevant API call by querying the DFGs.

4.8 Discussion

Argus-SAF can capture the parallel execution semantics at the inter-component level as discussed in section 4.4. However, Android also allows general thread-based concurrent execution within a component. This is not currently handled by Argus-SAF. The author leaves it as future work to adequately account for all possible concurrent executions, by leveraging existing work such as Indus [50].
CHAPTER 5: NATIVE CODE RESOLUTION

As discussed in previous Chapters, many works [3, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 41, 62] have design or utilize static analysis tools to detect security issues in Android applications. Only a couple of them [41, 62] address security issues related to native code. However, none of them can track precise inter-language data-flow. The existing state-of-the-art Android static analysis frameworks, such as FlowDroid [19], DroidSafe [25], IcTA [26] and CHEX [3], do not currently provide the capability to perform inter-language data-flow analysis or handle native components. When encountering a native method invocation, all of the existing data-flow analysis frameworks either apply a conservative model which assumes any data-flow could happen or ignore the side-effects produced by the native call, which will cause major imprecision in the analysis result.

5.1 Background and Example

Below provides necessary background information to understand how Android native world works, and how the inter-language communication is handled. A motivating example is also provided to discuss the challenges to track static data-flow for Android application with the native world.

5.1.1 Native Code Usage Modes in Android

Android developers can introduce native code in two ways. In the first mode, the developer can write certain functions in native language (C/C++) and include the compiled binary as a shared object as part of the application. Those functions are then called by an Android component that is still written in Java. In the other mode, a complete component can be written in native code, and the Android runtime directly calls the life-cycle methods of the component in the native code. Currently Android only allows the second mode for the Activity component (called native
Activity). Whereas all four Android component types could involve native code through the first mode.

5.1.2 Native Development Kit (NDK)

The Native Development Kit (NDK) [43] is a set of tools that allow designing part of the Android application using native languages. NDK provides platform libraries to help manage native Activity components and access physical device components. It uses Java Native Interface (JNI) [63] as the interface via which the java and C++ components talk to one another. It is mainly used in cases such as improving performance, reusing existing third-party C or C++ libraries, and so on.

NDK together with JNI defines how java code sends data to native functions and receives return values, and how native code creates/modifies/inspects java objects and invokes java methods.

Since Android 2.3, NDK provides a helper library which allows the developer to design a whole Android Activity using native code. To precisely handle inter-language dataflow in Android, Argus-SAF must have a comprehensive model for JNI and native Activity as explained in later Sections.

5.1.3 Binary Code Analysis

BitBlaze [64] is a hybrid binary analysis platform, which contains three components: 1) Vine: a static analysis component that translates assembly to IR, which supports x86 and ARMv4 architectures; 2) TEMU: It enables whole-system monitoring and dynamic binary instrumentation; 3) Rudder: It utilizes Vine and TEMU to conduct symbolic execution.

BAP [65] is binary analysis platform which supports x86 and ARM architectures. BAP re-designs Vine to assist its front-end features. After the IR translations process finished, BAP conducts its back-end analysis in the IR granularity.

Angr [47] is a binary analysis framework that combines many existing program analysis technique into a single, coherent framework, such as Dynamic Symbolic Execution, Veritesting, Value-Set Analysis (VSA). Angr leverages the IR lifter of Valgrind [66] to translate assembly to VEX IR. With the aid of VEX IR, Angr provides analysis support for many architectures including 32-bit and 64-bit versions of ARM, MIPS, PPC, x86. NativeDroid of Argus-SAF is built on top of Angr and uses its SimProcedure and Annotation features to model NDK libraries and JNI functions.
5.1.4 A Motivating Example

A malicious app developer can make use of NDK and develop part of the app’s functionality in the Native world. Figure 5.1 illustrates an example app (named “inter-language-leak”). It consists of two worlds, 1) Java world: An Activity component which loads a native library “multiple_interactions” and imports two native methods `propagateData()` and `leakImei()`; 2) Native world: Export two native functions which leverage NDK libraries to read Java objects and invoke Java methods.

Resolving native method call is different from resolving regular java calls. To find the native method callee, one has to know which native library is loaded by the instance. From the native library Argus-SAF needs to know what native functions are exported, then Argus-SAF can find the corresponding function as the native method callee.

To track the control and data-flow across language boundaries, a static analyzer must understand the semantics of both languages, as well as understand the inter-language communication interface and APIs.

As an example, the following sequence of events (as labeled in Figure 6.1) can happen in reality:

1) `MainActivity` invokes native method `propagateData()` and passes an object `d` which carries a sensitive data.

2) `Java_test_multiple_interactions_MainActivity_propagateData()` receives `data`, gets `str` field (sensitive data) and then invokes java method `toNativeAgain()`.

3) `toNativeAgain()` at `MainActivity` receives `data` and passes it to native method `leakImei()`.

4) `Java_test_multiple_interactions_MainActivity_leakImei()` will receive the `imei` and leaks to the log.

To track the data and control flow across language boundary, a static analyzer needs to understand the bridge interface – JNI. For example, when `MainActivity` invokes `propagateData()` at `J23`, the static analyzer needs to know: 1) the `libmultiple_interactions.so` has been loaded at `J7`;

45
Figure 5.1: The inter-language-leak App: The arrowed lines among the app components highlight some of the inter-language-communication
2) the corresponding native function name is `Java_test_multiple_interactions_MainActivity_propagateData` via applying naming convention. Furthermore, when native function `Java_test_multiple_interactions_MainActivity_propagateData()` invokes `MainActivity.toNativeAgain()` at C9, the static analyzer needs to model and analyze the reflection style JNI functions: 1) C4-C6 read `str` field from `data` and assign to `imei`; 2) C7 and C8 construct a method identifier to java method `MainActivity.toNativeAgain();` 3) C9 invokes `MainActivity.toNativeAgain()` with parameter `imei`.

After resolving the native method call at J23 and J26 and the native reflection call at C9, Argus-SAF can track data-flow between the two worlds. Then at C15, Argus-SAF will be able to say that the variable `imei` to be written to the log is sensitive.

5.2 Resolving Native Method Calls

JNI allows two ways to resolve a native method call to a native function:

1) Default: Follow the naming convention in JNI specification [67] to generate corresponding native function name. For example, as Figure 5.1 illustrated, the corresponding native function name for native method `MainActivity.propagateData()` is `Java_test_multiple_interactions_MainActivity_propagateData`.

2) Dynamic register: JNI allows the developer to register native method signature to native function mapping dynamically.

To assist data-flow analysis engine to find native method callee, the author proposes a `Native Method Mapping` data structure. `Native Method Mapping` is a map where the key is the native method signature, and the value is the corresponding native function name and the containing `so` file.

Algorithm 5 shows the pseudocode for generating `Native Method Mapping n_map` of a given APK A. First, each `class` in A is visited. If `class` defined native methods, then follow Algorithm 4 to find the possible native function containing `so` files. For each native `method` in the `class`, native function name `funcName` is generated following the naming convention. Then load each `so` file `nLib`, and see if the `funcName` exists in `nLib`. If yes, add it to the `n_map`. If not, continue checking the dynamically registered function list for `nLib` and check if the `method` is dynamically registered.
Algorithm 4 Resolve loaded library for class C

Input: all classes’ IR of A.
Output: Loaded library for class C, libNameSet

1: procedure RESOLVELibNameSet(A, C)
2:     libNameSet ← empty set
3:     loadSigs ← Set("System.load()", "System.loadLibrary()", "Runtime.load()", "Runtime.loadLibrary()")
4:     for all class ∈ A.getAllReachableClasses(C) do
5:         clinit ← class.getStaticInitializer();
6:         for all invoke ∈ clinit.getInvokeStatements() do
7:             if invoke.signature ∈ loadSigs then
8:                 libNameSet ← libNameSet :: invoke.getValueForParameter(1)
9:         end if
10:     end for
11: end for
12: return libNameSet;
13: end procedure

If yes, add it to the n_map. However, to obtain the dynamically registered functions for nLib is a non-trivial work. The following approach is used.

5.2.1 Dynamic Function Register Resolution

As illustrated in Figure 5.2, JNI allows register dynamic function mapping by implementing the JNI_OnLoad() method. The JNINativeMethod structure contains the mapping information between the native method name, signature and the corresponding native function pointer. C5-C8 defines a JNINativeMethod array gMethods to indicate the mapping for native methods foo() and bar(), then C16 invokes RegisterNatives() with gMethods to register. The procedures of resolving dynamic function register is as follows:

1) Dynamic register begins at JNI_OnLoad() method, whose first argument is JavaVM *vm. Therefore, Argus-SAF first constructs a fake pointer to the JNIIInvokeInterface structure, which has been modeled and attaches the initialized pointer to the first argument (register R0) of JNI_OnLoad().

2) Argus-SAF does the symbolic execution from the JNI_OnLoad(). In this situation, Argus-SAF needs to get the JNINativeInterface to make JNI calls. As Figure 5.2 illustrated, JNI_OnLoad() method will first declare an uninitialized JNIEnv *env variable. Then it will call GetEnv() function from vm to initialize the env variable. Argus-SAF creates a SimProcedure(GetEnv) to simulate this behavior. Argus-SAF constructs a fake JNINativeIn-
Algorithm 5 Generate Native Method Mapping of APK A

Input: All classes’ IR of A.
Output: A’s native method to so file map, n_map

1: procedure GENNATIVEMETHODMAP(A)
2:   \( n\text{\_map} \leftarrow \text{empty map} \)
3:   for all class \( \in \text{A.getClasses()} \) do
4:     nativeMethods \( \leftarrow \text{class.getNativeMethods()} \);
5:     if nativeMethods \( \neq \text{empty} \) then
6:       libnames \( \leftarrow \text{resolveLibNameSet(A, class)} \)
7:         for all name \( \in \text{libnames} \) do
8:           nLib \( \leftarrow \text{A.loadNativeLibrary(name)} \);
9:             for all method \( \in \text{nativeMethods} \) do
10:               funcName \( \leftarrow \text{method.toJNIName()} \);
11:                 if funcName \( \in \text{nLib.getFunctionNames()} \) then
12:                   n_map(method) \( \leftarrow (\text{funcName, name}) \);
13:                 else
14:                   dynamicMap \( \leftarrow \text{nLib.getDynamicRegisterFunctions()} \);
15:                     if method \( \in \text{dynamicMap} \) then
16:                       n_map(method) \( \leftarrow (\text{dynamicMap(method), name}) \);
17:                    end if
18:                 end if
19:           end for
20:       end for
21:     end if
22:     end for
23: return n_map;
24: end procedure

3) Argus-SAF hooks SimProcedure(RegisterNatives) to JNINativeInterface’s function pointer table. When the symbolic execution engine executes SimProcedure(RegisterNatives), Argus-SAF can get the memory address of the gMethods array, because each element is accessible at a fixed offset through the JNINativeMethod structure. Argus-SAF can resolve each element value of the gMethods based on the address and the structure of JNINativeMethod.

4) Each JNINativeMethod contains three elements, native method name, native method signature, native function address. Argus-SAF matches the native method information from SBDA and find its corresponding native function address. Then Argus-SAF can begin Native Function Summary Builder from that address.

5.3 Leveraging Existing Binary Analyzer for Data-flow Analysis

There are a number of existing binary analysis tools [47, 64, 65]. Angr [47] is used in this work. Angr is a general binary analysis platform which uses symbolic execution technique to recover
static JNINativeMethod gMethods[] = {
    {"foo", "(Ljava/lang/String;)V", (void*) native_foo},
    {"bar", "(Ljava/lang/String;)V", (void*) native_bar},
};

JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
    JNIEnv *env = NULL;
    if (vm > GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    ...}
    if (env > RegisterNatives(clazz, gMethods, numMethods) < 0) {
        return -1;
    }
    ...}
}

Figure 5.2: JNINativeMethod structure

precise CFG (called CFGAccurate) in binary and allows the user to perform an annotation-based analysis. However, Angr is not aware of NDK library, JNI function, and Java object/method. Therefore, it cannot be directly used to track data-flow in Android binaries.

To do NDK/JNI-aware data-flow analysis for Android binary, Argus-SAF leverages Angr’s symbolic execution engine and implements an Annotation-based Dataflow Analyzer.

Annotation-based Dataflow Analysis (ADA) leverages Angr’s Annotation and SimProcedure features and is NDK/JNI-aware. Annotation is a customizable interface which Angr uses to allow users to define what kind of data needs to be carried in the state of symbolic execution process and what’s the propagation rule. SimProcedure allows users to replace library function calls with a fake function that models the original library function’s effect on the symbolic execution state.
5.3.1 Custom Annotations

Two custom Annotations is designed to assist NDK/JNI-aware data-flow analysis:

1) SummaryAnnotation: Native code uses JNI functions to create/inspect/update Java objects, invoke Java methods, catch and throw exceptions, etc. What’s more, native code can conduct inter-component communication (ICC) with the aid of JNI functions. Therefore, NativeDroid implements SummaryAnnotation to capture data related to java operations in native code.

2) TaintAnnotation: It annotates tainted data with information, such as, taint type (source or sink), taint label, taint locations. There are two kinds of source and sink APIs in the native world: 1) Linux system calls; 2) JNI functions which invokes java world methods. All of them is annotated to capture all the possible taint information.

![Figure 5.3: JNINativeInterface and JNIInvokeInterface structures](image)

5.3.2 JNI Function Model

There are two key data structures in JNI, JNINativeInterface [68] and JNIInvokeInterface [69]. As Figure 5.3 illustrated, both of them contains a list of function pointers. JNIEnv * and JavaVM * are the pointers which point to the head of each table.

1) JNINativeInterface provides JNI functions to create/inspect/update java objects, invoke java methods, catch and throw exceptions, query java class information, etc. For example, the CallObjectMethod function is used to call a java instance method from a native method;
SetObjectField sets the value of an instance field of an object. As the native code of Figure 5.1 shows, each native function receives a JNIEnv * as its first argument and can invoke JNI functions based on it.

2) JNIInvokeInterface provides JNI functions to create/destroy java VM, and allocate/discover JNIEnv. EP of native Activity does not have JNIEnv * parameter. Therefore, the developer needs to use GetEnv() function to discover the thread’s JNIEnv *. If the thread has not been created, developer needs to use AttachCurrentThread() or AttachCurrentAsDaemon() function to attach a thread and allocate JNINativeInterface.

Understanding the semantics of the aforementioned JNI functions are essential for ADA to do NDK/JNI-aware analysis. Therefore, Argus-SAF needs to model each of the JNI functions in JNINativeInterface and JNIInvokeInterface using the SimProcedure technique provided by Angr. However, the invocation instructions for JNI functions are stripped in the released version of Android applications, and the JNI function calls happen through the indirect jump in the function pointer table of those two data structures. Therefore, Argus-SAF has to create a fake data structures to imitate JNINativeInterface and JNIInvokeInterface and set the corresponding function pointers at each offset to address of our modeled SimProcedures.

```
JNIEnv *
Fake JNINativeInterface
SimProcedure(GetStringUTFChars) {
TaintAnnotation: arg1 à ret;
}
```

Figure 5.4: GetStringUTFChars function model

Figure 5.4 illustrates our model of JNINativeInterface and its SimProcedure table. The model of GetStringUTFChars indicates that the TaintAnnotation of the first argument is passed to return value. For example, Figure 5.5 shows a native function getCharFromString that receives a JNIEnv *env as its first argument at C1. It invokes GetStringUTFChars() function from env at C5. As Figure 5.4 illustrated, GetStringUTFChars is the 170th element of JNINativeInterface. Therefore, its offset to JNIEnv * is 169 * 4 = 676 = 0x2A4. As the calling convention prescribed, the first
C/C++ Source Code

const char *getCharFromString(JNIEnv *env, jstring string) {
    if (string == NULL)
        return NULL;
    return env->GetStringUTFChars(string, 0);
}

Assembly

.text:00000610 ; getCharFromString(_JNIEnv *, _jstring *)
.text:00000610   PUSH   {R7,LR}
.text:00000612   ADD     R7, SP, #0
.text:00000614   MOVS    R2, #0
.text:00000616   CMP      R1, #0
.text:00000618   BEQ      loc_628
.text:0000061A   MOVS    R2, #0x2A4
.text:0000061E   LDR      R3, [R0]
.text:00000620   LDR      R3, [R3,R2]  
.text:00000622   MOVS    R2, #0
.text:00000624   BLX      R3
.text:00000626   MOVS    R2, R0
.text:00000628 loc_628
.text:00000628   MOV S    R0, R2
.text:0000062A   POP      {R7,PC}
.text:0000062A ; End of function getCharFromString(_JNIEnv *, _jstring *)

Concise Process

R0 = env
R2 = 0x2A4
R3 = R0 = env
R3 = R3 + R2 = env + 0x2A4 = address of GetStringUTFChars

Figure 5.5: getCharFromString function source code and assembly
argument of each function is stored in $R0$ register. The register value update process is illustrated in the Concise Process of Figure 5.5 which simplifies the procedures showed in Assembly code. First, $R0$ register is assigned to the value of $env$ (a pointer) parameter at $L1$. Second, $R2$ is assigned to $0x2A4$ at $L2$, which is the offset of $GetStringUTFChars$ from $JNIEnv \ast$. Then, $R3$ is updated with the value of $R0$ at $L3$, which equals the $env$ parameter. Finally, add $R2$ to $R3$ to get the address of $GetStringUTFChars$. $BLX R3$ instruction at $A11$ will call the $GetStringUTFChars$. When $ADA$ executes $A11$, it will call $SimProcedure(GetStringUTFChars)$, which will propagate any $TaintAnnotations$ from the first argument to the return value.

5.3.3 Java Method Summary

As shown in Figure 5.1, $C9$ invokes $CallVoidMethod()$ function which will make a java method call, and the callee is $MainActivity.toNativeAgain()$. $SBDA$ already generated a method summary for $MainActivity.toNativeAgain()$, which is $\Delta(toNativeAgain) = \langle(sink(arg1)@C15)\rangle$. The function model $SimProcedure(CallVoidMethod)$ takes $\Delta(toNativeAgain)$ and operates on its arguments to properly mark $TaintAnnotations$. For this case, the $data.str$ will be marked as leak.

5.3.4 Inter-component Communication (ICC) Resolution

Native code can make inter-component communication (ICC) by invoking java ICC APIs. As described in Section 4.4, the java code analysis has a comprehensive model for ICC, thus Argus-SAF applies the same model in function model $SimProcedure(CallVoidMethod)$ to capture the possible ICC in native code.

5.4 Handling Native Activity

Android NDK allows the developer to develop Activity in pure native language since Android 2.3 [43]. There are two ways to implement a native Activity [70].

1) $native\_activity.h$: In this way, the app needs to include $native\_activity.h$ header to implement a native activity. It contains the callback interface and data structures that are required to create a native activity. The default entry point is $ANativeActivity\_onCreate$ function. NDK allows developers to use a customized function name by specifying in Manifest.
2) android_native_app_glue.h: With include `android_native_app_glue.h`, an app can utilize `android_main` as entry point function to implement a native Activity.

**Algorithm 6 Collect Native Activity Info of APK A**

**Input:** Manifest file and all classes’ IR of A.

**Output:** A’s native Activity information, `native_activities`

1. `procedure collectNativeActivityInfo(A)`
2. `native_activities ← empty set`
3. `manifest ← A.getManifest()`
4. `for all compTag ∈ manifest.getComponentTags() do`
5. `compName ← compTag.getAttribute("android:name")`
6. `compClass ← A.getClass(compName)`
7. `if compClass.isChildOfIncluding("android.app.NativeActivity") then`
8. `map ← compTag.getMetaDataMap()`
9. `libs ← empty set`
10. `libName ← map("android.app.lib_name")`
11. `if libName = null then`
12. `libs ← resolveLibNameSet(A, compClass)` ▷ Invoke Algorithm 4
13. `else`
14. `libs ← libs :: libName`
15. `end if`
16. `funcName ← map("android.app.func_name")`
17. `if funcName = null then`
18. `if libs = empty then`
19. `libs ← A.getAllNativeLibs()`
20. `end if`
21. `for all lib ∈ libs do`
22. `if lib.hasSymbol("android_main") then`
23. `libName ← lib`
24. `funcName ← "android_main"`
25. `else if lib.hasSymbol("ANativeActivity.onCreate") then`
26. `libName ← lib`
27. `funcName ← "ANativeActivity.onCreate"`
28. `end if`
29. `end for`
30. `end if`
31. `native_activities ← (compName, libName, funcName)`
32. `end if`
33. `end for`
34. `return native_activities;`
35. `end procedure`

There are three important pieces of information needed for resolving a native Activity: name, containing so file and entry function name. Algorithm 6 shows the pseudocode for collecting these for all native Activities from an app A. Argus-SAF first iterates each component `compClass` in the `AndroidManifest.xml` and find the native Activities by check whether `compClass` is or is the child of “android.app.NativeActivity”. If `compClass` is a native Activity, Argus-SAF then read its metaData to obtain the `libName`. If did not get `libName`, Argus-SAF then evaluates `compClass`’s static initializer `jclinit` to find out the argument value for load library method calls, `System.load()`, `System.loadLibrary()`, `Runtime.load()`, and `Runtime.loadLibrary()`. Then assign it to `libName`. Argus-
SAF read the “android.app.func_name” from compClass’s metadata to obtain the funcName. If “android.app.func_name” does not exist, then the default entry function name is used. Argus-SAF then checks if the default name is “android_main” (the android_native_app_glue.h case) or “ANativeActivity_onCreate” (the native_activity.h case).

5.4.1 native_activity.h

As Figure 5.6 illustrated, the default EP of the native Activity is ANativeActivity_onCreate (NDK also allows developers to use a custom EP). ANativeActivity * is the first parameter whose first member is ANativeActivityCallbacks *callbacks. ANativeActivityCallbacks structure contains the callback functions which will be executed in the native activity lifecycle. However, when Argus-SAF conducts the ADA from EP, the symbolic execution engine cannot execute those callbacks, as there are no explicit calls. To comprehensively model this type of native Activity Argus-SAF takes a two-fold approach:

1) Resolve callback function address: As illustrated in Figure 5.6, the ANativeActivity_onCreate function assigns the callbacks to the corresponding index of ANativeActivityCallbacks structure. Argus-SAF applies symbolic execution on this EP to get addresses of those callbacks and its index in ANativeActivityCallbacks structure.
Argus-SAF first constructs a fake `ANativeActivityCallbacks` structure. Argus-SAF then constructs a fake `ANativeActivity` structure and maps the fake `ANativeActivityCallbacks` structure’s pointer to the `ANativeActivity` structure. Finally, Argus-SAF assigns the pointer to the fake `ANativeActivity` structure to the first argument (R0 register) of `ANativeActivity_onCreate`. Argus-SAF does the under-constrained symbolic execution from `ANativeActivity_onCreate` function. After the symbolic execution has finished, the elements of `ANativeActivityCallbacks` will be assigned real addresses of those callbacks.

2) Explicitly invoke callback functions: Argus-SAF hooks each callback function to `ANativeActivity_onCreate` and applies ADA from `ANativeActivity_onCreate` as the EP. One challenge here is when native Activity invokes JNI functions. As illustrated in Figure 5.6, there are no `JNIEnv *` in the EP, and the `ANativeActivity` structure’s `JNIEnv *` is uninitialized. The developers need to invoke `AttachCurrentThread` on `JavaVM *` to assign `env` like in C2 and C3. In ADA, Argus-SAF applies `SimProcedure(AttachCurrentThread)` to assign `env` element. After the `env` element is assigned, the ADA will be able to resolve JNI functions correctly.

```
int32_t handle_input(struct android_app* app, AInputEvent* event) {...}
void handle_cmd(struct android_app* app, int32_t cmd) {...}
void android_main(struct android_app* state) {
    ...
    state->onAppCmd = handle_cmd;
    state->onInputEvent = handle_input;
    // Read all pending events.
    while (1) {...}
}
```

Figure 5.7: android_native_app_glue.h example

### 5.4.2 android_native_app_glue.h

As illustrated in Figure 5.7, `android_main` is the EP, and the only argument is the `android_app * state`. There are two important callback function pointers in `android_app` structure, `onAppCmd` and `onInputEvent`. `onAppCmd` is used for activity lifecycle events, and `onInputEvent` is
used for input events. Developers need to provide their processing functions to the two callbacks. These callbacks will be triggered when activity and an input event occur, respectively.

To comprehensively model this native Activity type, Argus-SAF applies a similar approach as used to resolve $ANativeActivity\_onCreate$. Firstly, Argus-SAF run symbolic execution from $android\_main$ to resolve the two callbacks value. Then, Argus-SAF hook the two callbacks to $android\_main$ function and run ADA.

5.5 Discussion

The inter-language related operations such as JNI reflection call construction, dynamic function registration, and Intent value resolution, all require precise resolution of string values. Argus-SAF does constant string propagation in both $Amandroid$ and $NativeDroid$. If the string is manipulated, Argus-SAF will not be able to construct the precise value. Precise string analysis is expensive and non-trivial in both java analysis and binary analysis as mentioned in prior research [52,53,71]. The author leaves this for future research.

Argus-SAF inherits path explosion issues from Angr [47]. Control-/Data-flow analysis of $NativeDroid$ is mainly based on the symbolic execution engine of Angr. Path&State explosion is the natural defect of any symbolic execution techniques when encountering large programs as the analysis need to separate all the states for different execution paths. To alleviate explosion problem, $NativeDroid$ needs to better constrain the possible execution paths and states which are non-trivial [72]. These limitations will be handled in future work.
CHAPTER 6: DESIGN AND IMPLEMENTATION

Argus-SAF is a precise, general and efficient Android static analysis framework that leverages the Android system model, component-based analysis and inter-language analysis discussed in previous three chapters. Argus-SAF consists of Amandroid, NativeDroid and JNI Bridge. Amandroid is responsible for Dalvik-bytecode (java world) analysis, which I built from scratch. NativeDroid is responsible for binary code (native world) analysis, which is built on top of Angr [47]. NativeDroid implements the ADA algorithm described in Section 5.3. JNI Bridge is the middle layer that assists the control and data communication between Amandroid (implemented in Scala ¹) and NativeDroid (implemented in Python). JNI Bridge leverages jpy [73], a bi-directional Java-Python bridge to enable Amandroid and NativeDroid transfer control and data.

¹Scala is a JVM-based language.

Figure 6.1: The Argus-SAF analysis pipeline
Figure 6.1 illustrates the pipeline of Argus-SAF which consists of four major steps:

1) **APK Preprocess** collects useful information from an app.

2) **Environment Model** generates environment model for both Java and native components.

3) **Intra-component Analysis** computes information flow for each Android component in a native-aware fashion.

4) **Inter-component Analysis** link inter-component data flows.

The output of Argus-SAF is the data-flow graph (DFG), data dependence graph (DDG), summary table (ST) and app-level data dependence graph (ADDG), which can be applied in various types of security analysis. For example, one can use DDG or ADDG to find whether there is any information leakage from a sensitive source to a critical sink by querying whether there is a data dependency chain from source to sink.

Figure 6.2: APK preprocess and environment model
6.1 APK Preprocess

The first half of Figure 6.2 illustrates the APK preprocess steps. Argus-SAF takes an APK as the analysis input. It decompiles the APK into three parts, \textit{dex} files, \textit{Manifest\&Resource} files and \textit{so} files. Amandroid leverages the \textit{DEX2IR} and \textit{Resources Parser} to decompile Dalvik bytecode into \textit{Intermediate Representation} (IR) language Jawa\(^2\) and collect component information. NativeDroid uses \textit{pyvex} from \textit{Angr} to translate binary into \textit{VEX IR} [76].

The \textit{Native Info Analyzer} receives information from \textit{DEX2IR} and \textit{Resources Parser} to compute native world related information:

1) Generate \textit{Native Method Mapping} following Algorithm 5 described in Section 5.2.

2) Collect \textit{Native Activity Info} following Algorithm 6 described in Section 5.4.

6.2 Environment Model

Android is an event-based system, and as such no single method can be used as EP for the data-flow analysis. To capture all lifecycle and event control-/data-flow of an Android java component and to generate EP for data-flow analysis, \textit{APK Preprocessor} implements the algorithm 1 described in Section 2.1 to build environment model for each Android java component, and generates an \textit{Environment Method} as the EP for each java component.

\textit{Native Component Environment Builder} is implemented by following the solution described in Section 5.4 to generate an \textit{Environment Function} as the EP for each native Activity component.

The \textit{Environment Method/Function} explicitly invokes the event/lifecycle callbacks as the Android runtime would.

\(^2\) \textit{Jawa} language is designed based on Pilar, a language used in static analysis and model checking frameworks such as Bakar Kiasan [74] and Bakar Alir [75]. \textit{Jawa} is a structured and annotation based language, where user can add different types of annotations to support different languages. This gives the flexibility to extend Argus-SAF to support analysis for other frameworks and languages in the future. \textit{Jawa} language specification is available at http://pag.arguslab.org/jawa-language.
6.3 Intra-component Analysis

As described in Chapter 4, Argus-SAF first needs to conduct intra-component data-flow analysis to capture all intra-component-level behaviors. Argus-SAF has following two workflows to works with pure java Android application and Android application with native code, respectively.

6.3.1 RFA-based Analysis

Figure 6.3: RFA-based analysis

Figure 6.3 illustrates the main steps for RFA-based analysis. It follows the intra-component analysis approach discussed in Chapter 4. It computes the intra-component level Data-flow Graph (DFG) using the data-flow analysis algorithm discussed in Section 3.1. The DFG contains a ICFG to represent the whole control-flow within a component, and for each program point it maintains the points-to information. Leveraging the DFG, RFA-based analysis then computes the intra-component Data Dependent Graph (DDG).

6.3.2 SBDA-based Analysis

Argus-SAF implements the Summary-based Bottom-up Data-flow Analysis (SBDA) algorithm by following the techniques described in Section 3.2. As presented in Figure 6.4, it consists of the following components.

6.3.2.1 Call Graph Builder

It receives the environment method/function from Environment Model and uses it as the EP to compute a native-aware call graph. Unlike traditional java call graph building algorithm, our
call graph will not stop at native method calls. Instead, it will evaluate the corresponding native function to address possible reflection call from native to java and add those call target as callee of this native method. The native reflection style call is resolved by following the JNI function model described in Section 5.3.

6.3.2.2 Bottom-up Summary Propagator

It receives the call graph $CG$ from Call Graph Builder and applies a topological sort with the reverse order to get a list of method/function $MList$. It iterates the $MList$ to send the work order to corresponding Method/Function Summary Builder to compute summary $\Delta$ and propagate to their callers.

6.3.2.3 Java Method Summary Builder

Argus-SAF leverages the $RFA$ engine to compute the summary for a given method. The difference is that an intra-procedural analysis is applied. When the engine reaches a method call, it will not flow the points-to facts into the callee. Instead, it will obtain the summary $\Delta(\text{callee})$ and apply such summary on current points-to facts to imitate the heap manipulation behaviors.
When data-flow analysis finishes, Argus-SAF collect the heap manipulation behavior of the current method and generate a summary $\Delta(method)$.

### 6.3.2.4 Native Function Summary Builder

Upon receiving a work order with native method signature and the containing so file, the **Native Function Summary Builder** first identifies the binary address for the corresponding native function of the native method. Then it applies ADA (as described in Section 5.3) to generate $\Delta$ starting from such $EP$ as follows.

1) Add $SummaryAnnotation$ to each argument including argument index and type information, because from $EP$’s perspective all mutable arguments are considered as $HeapBase$.

2) Add $SimProcedure$ to all JNI functions which might create/delete/manipulate the heap of java objects. When ADA evaluates, those $SimProcedures$ will adequately update and propagate $SummaryAnnotation$. As an example, native code can construct java $String$ with the aid of JNI function $NewString()$ or $NewStringUTF()$, JNI function $SetObjectField()$ will set data to a java object.

3) When ADA encounters any method/function invocation, it will check whether it is a source or sink API. If so ADA will add $TaintAnnotation$ to proper $HeapLocs$. For method invocation, Argus-SAF will also check with SBDA to obtain its $\Delta$ and apply it on the arguments $SummaryAnnotations$.

4) When ADA is over, Argus-SAF extract the $SummaryAnnotation$ together with $TaintAnnotation$ related to each argument and return node (if the JNI function returns a java object) to build the summary.

Argus-SAF takes $Java\_test\_multiple\_1interactions\_MainActivity\_propagateData()$ function at Figure 5.1 as an example to walkthrough the native function $\Delta$ building process. $Java\_test\_multiple\_1interactions\_MainActivity\_propagateData()$ function receives one argument $data$. Argus-SAF assigns $SummaryAnnotation(arg1, test.multiple\_interactions.Data)$ to $data$ and $SummaryAnnotation(arg1.str, \text{’java.lang.String’})$ to $data.str$. C$\&$E invokes $GetObjectField()$ to read $str$ field of $data$.
to variable imei. SimProcedure(GetObjectField) get SummaryAnnotations from data.str and propagate it to variable imei. C9 invokes java method toNativeAgain() and pass imei as the first argument. SimProcedure(CallVoidMethod) obtain Δ(toNativeAgain) from SBDA, and apply on SummaryAnnotations of imei, Argus-SAF then gets TaintAnnotation(sink(arg1.str), ‘C15’). After finish running ADA, Argus-SAF collects the SummaryAnnotations and TaintAnnotations related to each argument (there are no return value in this case). Finally, Argus-SAF checks the heap changes of each HeapBase and taint informations to construct the summary Δ(propagateImei) = ⟨(sink(arg1.str)@C15)⟩.

6.4 Inter-component Analysis

![Inter-component Analysis Diagram]

Figure 6.5: Inter-component analysis

Resolving Inter-component communication (ICC) is essential for any Android static analysis tool. Argus-SAF implements the Summary Table (ST) building algorithm discussed in Section 4.4. The Inter-component Analyzer collects ICC information from all Java components and native Activity components. Then, it computes ST for each component and uses Component-based Analysis to address ICC data-flow.

6.5 Distributed Computation

Argus-SAF’s modules are implemented using Scala, leveraging Akka’s actor-model [77] to achieve distributed computation. Actor-model is a mathematical model of concurrent computation that treats “actors” as the universal primitives of concurrent computation [78]. Each actor is a
computation unit which maintains its private state and can only affect each other through messages to avoid usage of any locks.

As Figure 6.6 indicates, Argus-SAF’s phases are encapsulated as actors whereas each of them maintains its own state and behavior. Argus-SAF Supervisor Actor is responsible for handling the user’s app analysis request and dispatching orders to individual worker actors and based on the response (of worker actors) moving the analysis to the next phase. Each phase of the analysis has multiple worker actors that perform the computation concurrently, leveraging parallel computing power. The actors communicate with each other with only a small amount of data; thus Argus-SAF could run in a highly distributed fashion.

The component level $DFG$, $DDG$, and app metadata make the core information to be used in the security analysis phase. New security analyses may be needed to be performed from time to time while the required core information is the same for the same app. Thus, storing the core information can save a tremendous amount of computing time. However, the data dependency graphs can be quite big (GBs for a typical app). Thus, Argus-SAF does not attempt to store the graphs, but rather only store the data-flow facts computed during the static analysis phase. The
graph structure can be reconstructed efficiently when needed. This *staging* strategy is illustrated in Figure 6.6. Apk Info Collect Actor and Points-to Analysis Actor store the collected apk information and computed dataflow facts into the stage database, which can be used to rebuild the component-level DFGs, DDGs for the Security Analysis Actor. The data-flow facts stored in the database does not take much space — few MBs for an app.
CHAPTER 7: EVALUATION

This dissertation evaluated Argus-SAF extensively on benchmark and real-world apps. Several sets of apps have been used in the experiments:

1) 2,300 popular apps from Google Play (GPlay).
2) 2,300 randomly selected malware apps from the AMD dataset [9] (MAL).
3) 100,000 randomly selected popular apps from AndroZoo [79] (ZOO).
4) 24,553 malware apps from the AMD dataset [9] (AMD).

And three benchmarks (hand-crafted apps by other researchers and me). This chapter describes the results of those tests. The experiments were run on a machine with 2.20 GHz, 48-core Xeon, and 256 GB RAM.

7.1 Performance and Scalability

7.1.1 Component-based Analysis

Argus-SAF offers the user options of choosing multiple precision levels. For instance, the context depth $k$ (of the control flow graph) serves as a parameter to set the trade-off between precision and performance. Our reported experiment results correspond to $k = 1$ (unless otherwise mentioned), meaning that the static analyzer tracks up to one calling context. Argus-SAF also allows the user to define the scope of the analysis by applying the conservative model for certain third-party libraries (section 3.1.3). In our experiment, Argus-SAF applied the conservative model for about 100 well-known third-party libraries.

The most computation-intensive step in component-based analysis module of Argus-SAF is building the $DFG$ for each component. Once the $DFG$ is built, the running times of the subsequent

---

This chapter is partially based on the authors CCS 2014 [48] and TOPS 2018 [49] papers. See Appendix A for permission.

68
analyses are negligible – these include building ST, DDG and running the specialized analyses on top of them. Figure 7.1 presents the time is taken by Argus-SAF to construct DFG for 4,600 real-world apps (GPlay and MAL).

These apps have 141319.50 lines of bytecode instructions on average. The median running time for computing the DFG for all the components in an app is 3 minutes; the minimum is 0.15 seconds whereas the maximum is 169 minutes. The scatter plot shows both the running time and the size of the app (in a number of bytecode instructions).

### 7.1.2 Inter-language Analysis

This dissertation evaluated inter-language analysis module of Argus-SAF on ZOO and AMD.

SBDA is the core engine and the most computation-intensive step in inter-language analysis module of Argus-SAF. Figure 7.2 presents the time taken to construct SBDA for 10,000 randomly picked real-world app components. These components reach 144 methods on average. The average running time for computing the SBDA for each component is 42.288 seconds; the minimum is 0.001 seconds whereas the maximum is 86 minutes.
This dissertation constructed a separate experiment focused on the running time for native code analysis. Figure 7.3a illustrates the time taken to build function summary for 2,000 randomly picked real-world app native functions (from ZOO and AMD). These native functions reach 4,417 instructions on average. The average running time is 88.982 seconds; the minimum is 0.107 seconds whereas the maximum is 136 minutes. Figure 7.3b illustrates the time taken to construct native Activity analysis for all 579 native activities in ZOO and AMD (failed to analyze 33 due to path explosion problem). These native activities reach 41,285 instructions on average. The average running time is 570.513 seconds; the minimum is 0.247 seconds whereas the maximum is 438 minutes.

7.2 Intent Resolution Effectiveness

This dissertation evaluates Argus-SAF on all 4,600 real-world apps in our dataset, to calculate the precision of the Intent resolution. As Table 7.1 indicates, in GPlay 21,062 ICC calls require
Figure 7.3: Native code analysis performance

(a) Function summary builder

(b) Native Activity analysis
Intent resolution, and Argus-SAF can infer precise Intent string values for 17,354 (89%) of them. In MAL apps, 18,749 ICC calls require Intent resolution, and Argus-SAF can infer precise Intent string values for 13,883 (80%) of them. Overall 85% of the ICC cases can be precisely resolved, showing that Argus-SAF's Intent resolution is capable of handling real-world apps and will not introduce too much over-approximation for ICC data flows.

Table 7.1: Intent resolution precision

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GPlay</th>
<th>MAL</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Intent invocation point</td>
<td>21,062</td>
<td>17,354</td>
<td>38,416</td>
</tr>
<tr>
<td>Precise Intent resolution</td>
<td>18,749</td>
<td>13,883</td>
<td>32,632</td>
</tr>
<tr>
<td>Precision</td>
<td>89%</td>
<td>80%</td>
<td>85%</td>
</tr>
</tbody>
</table>

7.3 Statistics of Native Library Usage

Native library usage on both ZOO and AMD is collected. As Table 7.2a indicates, the overall native library usage is reasonably high no matter in benign dataset or malware dataset. ZOO has much higher native library usage than AMD which means there are many benign use cases for native libraries, so native library existence is not a good indicator for detecting Android malware. Dig into the detail of native library and understand its behavior is needed. The author also found cases where an app has native methods but no .so files. This means the .so file is probably downloaded at runtime (in which case no static analyzer will be able to identify). This dissertation found native Activity usage in both ZOO and AMD, which shows the necessity of handle such case.

Table 7.2b lists the usage of different architectures. Overall, 32-bit architecture has much higher percentage over 64-bit architecture. ARM is the most popular architecture for Android. Not surprisingly most of the binaries are in ARM architecture.

The native library can invoke java method through reflection style function calls. The author conducted an experiment to study the capability of NativeDroid to resolve such calls, and the results are shown in Table 7.2c. The author also studied the distribution of those reflection call targets and found that the majority of the reflection calls (especially from AMD) are targeted to library APIs as oppose to App methods. Argus-SAF experiences poor performance on ZOO reflection call resolving due to the larger code base and complex logic in market apps as opposed to
Table 7.2: Native library statistics for datasets

(a) Native library usage.

<table>
<thead>
<tr>
<th></th>
<th>ZOO</th>
<th>AMD</th>
<th></th>
<th>ZOO</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total App (^a)</td>
<td>99,910</td>
<td>24,384</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has Native (^b)</td>
<td>39,661</td>
<td>5,365</td>
<td>/ Total App</td>
<td>39.7%</td>
<td>22.0%</td>
</tr>
<tr>
<td>Has .so File</td>
<td>35,705</td>
<td>5,164</td>
<td>/ Has Native</td>
<td>90.0%</td>
<td>96.2%</td>
</tr>
<tr>
<td>Has Native Method</td>
<td>32,576</td>
<td>3,867</td>
<td>/ Has Native</td>
<td>82.1%</td>
<td>72.1%</td>
</tr>
<tr>
<td>Has Native Activity</td>
<td>583</td>
<td>29</td>
<td>/ Has Native</td>
<td>1.5%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Native Method</td>
<td>4,232,699</td>
<td>112,000</td>
<td>/ Has Native Method</td>
<td>106.7</td>
<td>29.0</td>
</tr>
<tr>
<td>Pass Data</td>
<td>3,661,881</td>
<td>90,212</td>
<td>/ Native Method</td>
<td>86.5%</td>
<td>80.5%</td>
</tr>
<tr>
<td>Pass Object</td>
<td>1,496,911</td>
<td>45,981</td>
<td>/ Pass Data</td>
<td>35.4%</td>
<td>51.0%</td>
</tr>
</tbody>
</table>

\(^a\)Argus-SAF failed to analyze a few apps that use advanced obfuscation.

\(^b\)Has Native = Has .so File \(\cup\) Has Native Method \(\cup\) Has Native Activity.

(b) Architecture.

<table>
<thead>
<tr>
<th></th>
<th>ZOO</th>
<th>AMD</th>
<th></th>
<th>ZOO</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total .so File</td>
<td>235,616</td>
<td>16,116</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td>162,356</td>
<td>13,792</td>
<td>/ Total .so File</td>
<td>69.0%</td>
<td>85.6%</td>
</tr>
<tr>
<td>ARM 64</td>
<td>10,111</td>
<td>2</td>
<td>/ Total .so File</td>
<td>4.3%</td>
<td>0.01%</td>
</tr>
<tr>
<td>X86</td>
<td>37,745</td>
<td>1,149</td>
<td>/ Total .so File</td>
<td>16.0%</td>
<td>7.1%</td>
</tr>
<tr>
<td>X86 64</td>
<td>8,511</td>
<td>2</td>
<td>/ Total .so File</td>
<td>3.6%</td>
<td>0.01%</td>
</tr>
<tr>
<td>MIPS</td>
<td>9,658</td>
<td>770</td>
<td>/ Total .so File</td>
<td>4.1%</td>
<td>4.8%</td>
</tr>
<tr>
<td>MIPS 64</td>
<td>2,477</td>
<td>2</td>
<td>/ Total .so File</td>
<td>1.1%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Other</td>
<td>4,758</td>
<td>399</td>
<td>/ Total .so File</td>
<td>2.0%</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

(c) Reflection call.

<table>
<thead>
<tr>
<th></th>
<th>ZOO</th>
<th>AMD</th>
<th></th>
<th>ZOO</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Reflection Call</td>
<td>7,664(^a)</td>
<td>33,497</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolved Call</td>
<td>4,744</td>
<td>29,336</td>
<td>/ Total Reflection Call</td>
<td>61.9%</td>
<td>87.6%</td>
</tr>
<tr>
<td>Library API Call</td>
<td>2,555</td>
<td>24,249</td>
<td>/ Resolved Call</td>
<td>53.9%</td>
<td>82.7%</td>
</tr>
<tr>
<td>App Method Call</td>
<td>2,189</td>
<td>5,087</td>
<td>/ Resolved Call</td>
<td>46.1%</td>
<td>17.3%</td>
</tr>
</tbody>
</table>

\(^a\)Due to time constraint Argus-SAF only finished analyzing 37,781 native functions from ZOO.
malware apps. From the obtained reflection call list, the author sees many interesting library APIs being called, such as `SmsManager.sendDataMessage()`, `ClassLoader.loadClass()`, which might raise red flags.

7.4 Application to Security Analysis

Argus-SAF is a highly extensible framework that allows analysts to write customized security checkers as plugins on top of it. This section reports experimentation results for most of Argus-SAF’s security analysis plugins. All experiments are done using the GPlay and MAL datasets (real-world apps).

7.4.1 Password Leakage Checker

This dissertation used the following policy to vet apps for adequately handling user passwords: “password should not be saved in the device (not even when encrypted) and should be transferred to a remote server only via HTTPS.” (similar guidelines can be found in, e.g., [80]). Argus-SAF can be readily used to verify whether the input app obeys such a policy. The only “to-do” task is to identify which variables in the app’s code corresponds to a password object (source), and to define the potential leaking sinks.

Argus-SAF finds the `TextView` item corresponding to a password (when the `inputType` attribute’s value is `textPassword`) in an app’s layout file and identify its unique ID. Argus-SAF then looks for the usage of this particular ID in method call `Context.getViewbyId(x)`, which is done through a standard reaching-definition analysis on the intra-procedural control-flow graph; this method returns an `EditText` object `y`, and `y.getText()` gives the password object. Argus-SAF can then define this object as the source. Argus-SAF prepares the list of sink APIs by considering the relevant I/O operations (e.g., `Log.i(key, value)`, and `URL.openConnection()`). The rest of the analysis is the straightforward application of DDG as explained in Section 4.7.

Argus-SAF found several examples of password leakage. Table 7.3 show part of the results. A few interesting patterns is observed:

---

2If the app’s developer obfuscates the ID through, e.g., mathematical manipulation, our reaching-definition analysis will not be able to return concrete values for the view ID. In this case Argus-SAF will conservatively report a possible malicious app, since it is extremely unlikely a benign app will perform such manipulations on a view ID.
1) the password is logged in clear text (Case 1 in Table 7.3);

2) the password is reaching a Network API over HTTP channel (Entry 2 in Table 7.3);

3) the password is saved in `SharedPreferences` (Entry 3 in Table 7.3).

Case 2 stems from a third-party library for Twitter. The `DDG` and `DFG` shows that the app sends the user’s password to http://api.twitter.com/1 (an HTTP connection). Interestingly, one can see that the URL is not currently working and only responds with a message “SSL is required.”

![Table 7.3: Password leakage checker report](image)

<table>
<thead>
<tr>
<th>App Name</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.datpiff.mobile.apk</td>
<td>GPlay</td>
<td>Get user password, encode it, then write it into log.</td>
</tr>
<tr>
<td>com.toystorymusic.musicapp.*</td>
<td>MAL</td>
<td>Send password to server via http.</td>
</tr>
<tr>
<td>com.snappii.angel_investing__news_v10.apk</td>
<td>GPlay</td>
<td>Write user password into SharedPreferences.</td>
</tr>
</tbody>
</table>

### 7.4.2 OAuth Token Leakage Checker

OAuth 2.0 [81] is a popular authentication protocol which is frequently used for single-sign-on (SSO), social sharing, etc. Typically, Google, Facebook, and other popular services are the Identity Provider (`IdP`). Thus, if the OAuth token is stolen, the user’s corresponding `IdP` account can be compromised. Similar to password tracking, Argus-SAF can be used to check whether the input app obeys the OAuth token protection policy. The source of the potential leak is determined using a simple strategy of tracking the string literal “access-token” and marking the related object creation statements as the source. The sinks are the same as in the password leak detection. Argus-SAF found several potential OAuth token leakage cases, some of which are shown in Table 7.4. A couple of interesting patterns is observed:

1) The implicit intent carrying the token can possibly reach a malicious app. (*e.g.*, Case 1 in Table 7.4), and

2) A malicious app having Log-read permission can grab the OAuth token (*e.g.*, Case 1 and Case 2 in Table 7.4).
Note that the above type of token leakage is very different from the explicit token discoveries reported in a recent work [82].

Table 7.4: OAuth token leakage checker report

<table>
<thead>
<tr>
<th>App Name</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.skout.android.apk</td>
<td>GPlay</td>
<td>Send OAuth token via implicit ICC; also write it to Log.i().</td>
</tr>
<tr>
<td>com.keek.apk</td>
<td>GPlay</td>
<td>Write OAuth token into log using Log.d().</td>
</tr>
</tbody>
</table>

7.4.3 Hiding-icon Checker

Hiding-icon is one common malware scheme to hide the application’s physical existence on the phone. In particular, it hides the malware app’s launcher icon while making the malware’s background service run. To do this, the app needs to disable its main component while telling the android system not to kill its background service by calling an API `Context.setComponentEnabledSetting()` with specific parameters as shown in Figure 7.4.

The idea of detecting such suspicious behavior is to extract from DFG the values passed to the `Context.setComponentEnabledSetting()` API and match them with the malformed parameters (as shown in Figure 7.4). Applying this checker to the app dataset, Argus-SAF found 4 GPlay apps and 75 MAL apps having this suspicious behavior.

7.4.4 Crypto Library Misconfiguration Checker

The author implemented a plugin to check whether an app conforms to the following crypto API configuration rules [28]:

1) Do not use ECB mode for encryption.

2) Do not use a non-random IV for CBC encryption.
Rule 1 is to evaluate the string value used to create the `javax.crypto.Cipher` instance. If the string value indicates that the cipher will run in ECB mode, the checker will report an alarm. To check Rule 2, the checker first detects the cipher is using the CBC mode, and then checks the IV creation process to see whether a constant IV is used. Table 7.5 summarizes the results Argus-SAF obtained through running the above checker on the app dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GPlay</th>
<th>MAL</th>
</tr>
</thead>
<tbody>
<tr>
<td># apps using ECB mode</td>
<td>438</td>
<td>303</td>
</tr>
<tr>
<td># apps using non-random IV</td>
<td>210</td>
<td>87</td>
</tr>
</tbody>
</table>

### 7.4.5 SSL/TLS Misconfiguration Checker

SSL/TLS protocols are widely adopted in Android applications to provide secure data transmission between the client app and their backend server. App developers may not be adequately trained for correctly using SSL/TLS library, and there is a lack of visual security indicators for SSL/TLS usage in the development environment (IDE). As a result, SSL/TLS library APIs can be easily misconfigured [22,29].

One common misuse case is allowing all hostnames for the SSL/TLS’s `HostnameVerifier`, by invoking `SSLSocketFactory.setHostnameVerifier()` with parameter `ALLOW_ALL_HOSTNAME_VERIFIER`. To capture this, the checker will evaluate whether the parameter passed to `SSLSocketFactory.setHostnameVerifier()` is equal to `ALLOW_ALL_HOSTNAME_VERIFIER`.

Another misuse case is accepting all certificates or accepting all hostnames for a certificate as long as a trusted CA signed the certificate, by providing their own or third-party-implemented `TrustManager` and `SocketFactory`. [29] provides a list of problematic `TrustManager` and `SocketFactory` implementations with its class names, which our checker plugin searches for in a given app. Table 7.6 summarizes the results Argus-SAF obtained through running the above checker on the app dataset.

### 7.4.6 Communication Data Leakage Checker

Phone call logs, contacts, and SMS messages are a few examples of user’s sensitive information which should be kept private. Argus-SAF can be used to check whether an app obeys
the above data usage policy. Argus-SAF applies simple strategies to identify the various communication data sources. Basically, Argus-SAF tracks the corresponding (i.e., tied with the data source) string literals or BroadcastReceivers: (1) Call logs: “content://call_log/calls”; (2) Sim card contacts: “content://icc/adn”; (3) Phone contacts: “com.android.contacts”; (4) SMS: “content://sms/inbox/” and input for BroadcastReceivers handling the “SMS_RECEIVED” event.

On the other hand, the sinks are any outgoing communication channel, such as http/https write, SMS send, implicit Intent send, etc. Argus-SAF found several potential sensitive data leakage cases, some of which are shown in Table 7.7.

Table 7.7: Communication data leakage checker report

<table>
<thead>
<tr>
<th>App Name</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.skymoons.hqg.anzhi.apk</td>
<td>GPlay</td>
<td>Read user’s SMS inbox, write into log, then send text message to the senders.</td>
</tr>
<tr>
<td>12050f267d5e8ce6f77d2111cd-3043f0.apk</td>
<td>MAL</td>
<td>Read user’s SMS inbox, store in a JSON object, write into SharedPreferences, then upload to its C&amp;C server.</td>
</tr>
<tr>
<td>5339a0e7e86ac1f5472f832874-426c25.apk</td>
<td>MAL</td>
<td>Upload user’s SMS content and information to its C&amp;C server.</td>
</tr>
<tr>
<td>51bf3112982473e99b88965f6e-271799.apk</td>
<td>MAL</td>
<td>Read user’s SMS inbox, upload to its C&amp;C server, send text message to senders.</td>
</tr>
</tbody>
</table>

7.4.7 Intent Injection Checker

Intent is one of the most common ways for an Android component to receive and process data from outside. If an app makes wrong assumptions for the incoming intent and performs sensitive operations based on it, that may result in serious security holes [3,4].

To detect the above issue, Argus-SAF mark the intent receiving point as the source and sensitive operations (such as open URL connection, crafting another intent) as the sink. Argus-SAF
then queries the DDG to find whether there is a data dependence path between them. Argus-SAF found several potential intent injection cases, some of which are shown in Table 7.8.

Table 7.8: Intent injection checker report

<table>
<thead>
<tr>
<th>App Name</th>
<th>Dataset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.fcbh.dbp.BibleSocietyOffPhilippines.apk</td>
<td>GPlay</td>
<td>TwitterLoginActivity retrieves the “url” from incoming intent and sends it to another Activity.</td>
</tr>
<tr>
<td>com.kamagames.notepad.apk</td>
<td>GPlay</td>
<td>Start an activity by using the mData of the incoming intent.</td>
</tr>
<tr>
<td>com.qryptal.verifydetailsauthtenticate.android.apk</td>
<td>GPlay</td>
<td>Allows any app inject URL to its ShareActivity, which will then encode it to a Barcode and display to the user. If user scan the Barcode, they might be redirected to malicious websites.</td>
</tr>
<tr>
<td>com.freegame.basketball.apk</td>
<td>GPlay</td>
<td>Allows any app inject data into its SharedReference, which will disable this app’s functionality.</td>
</tr>
<tr>
<td>com.mmmono.mono.apk</td>
<td>GPlay</td>
<td>Allows any app send commands to start/stop its service’s heartbeat and connectivity status.</td>
</tr>
<tr>
<td>com.bigfishgames.dmddgoog-free.apk</td>
<td>GPlay</td>
<td>Allows any app send commands to launch arbitrary URL and components.</td>
</tr>
</tbody>
</table>

### 7.4.8 Inter-language Security Checker

The author evaluated Argus-SAF for real-world app security vetting on our datasets and successfully detected four kinds of security issues in real-world apps that hide malicious behavior into the native world to evade the detection of static analysis tools that only cover the java world.

#### 7.4.8.1 Case Study 1: Inter-language Data Leakage

Sensitive information leakage has been a widespread security issue in Android platform. To make detection harder, malware moves the leaky behavior into the native world. For example, Backdoor.Triada.1 (Version 1 of backdoor malware Triada) [9] obtains the IMSI of the device in java layer. Then it passes the IMSI to native method nativeSayTest(). The corresponding native function will then leak IMSI by invoking SmsManager.sendTextMessage().

Argus-SAF can detect this issue because Native Function Summary Builder will generate a summary $\Delta(nativeSayTest) = \langle (sink(\text{arg2}@Cx) \rangle$ and feed back to SBDA. SBDA will then mark the IMSI as the source, and when nativeSayTest() is invoked with such source, Argus-SAF will report the leak issue.
7.4.8.2 Case Study 2: Stealthy Command Execution

Malware writers love to use shell command to execute malicious behaviors. For example, Backdoor.DroidKungFu.1 is a backdoor malware that tries to root the device and execute malicious code. It roots the device with the aid of the sebino program. If the device has not been rooted, it will copy sebino to /data/data/pkg/secbino and chmod 4755 to get the execution permission. Then it executes sebino to get the root privilege and start a service to download other malware apks to install.

Argus-SAF can detect these behaviors by modeling those Linux programs that can execute the shell command, such as, popen, system, execv. Argus-SAF can get the parameters of those system API and know what shell commands are executed.

7.4.8.3 Case Study 3: Stealthy Command and Control Communication

Command and Control (C&C) server is frequently used in malware to conceal the malware command and control information generation process into network communication. This process can also move to native world. For example, Trojan-Dropper.Boqx.1 launches a thread to exec native code in StatService class. In the native world, it enables the WIFI to ensure the success of communicating with a server. Then it communicates with the server to get the malicious payload and then dynamically loads these payloads. All these behaviors are completed by native reflection calls.

Argus-SAF can detect such malicious behavior because of Argus-SAF models all the JNI functions from the JNINativeInterface structure. After running ADA, Argus-SAF can know what kind of reflection calls are made in the native world.

7.4.8.4 Case Study 4: Malicious Identity Hiding

Malicious identity such as server URL and premium number is essential for many malware analysis techniques. There exist malware samples that hide those identities in the native world. For example, Trojan-SMS.Ogel.1 encapsulates its C&C server URL in native code, and when it starts running, it will read the URL data by invoking a native function Java_com_google_cn_ni_u(). Java_com_google_cn_ni_u() uses NewStringUTF() to create a java String of its URL.
Argus-SAF can obtain the value of the C&C server URL. When malware returns the server URL from native world to java world through native method, NativeDroid can generate the summary that illustrates this process $\Delta(u) = \langle (\text{ret} = URL@Cx)(source(URL)@Cx) \rangle$. Then Amandroid will continue SBDA with the summary information.

### 7.5 What it Takes to Build a New Analysis

The advantage of Argus-SAF’s approach is that the general framework provides a means for building a variety of further security analyses in a straightforward and easy way. Each special analysis built on top of Argus-SAF involves developing a “Checker plugin” that leverages the DFGs and DDGs from Argus-SAF’s analysis. Moreover, once the core analysis produces DFGs and DDGs for an app, they can be stored and reused in multiple security analyses. Table 7.9 presents the summary of the plugins used in the above applications, which shows their size in Scala LOC, as well as the average running time. This can be compared with the size of the core engine and its average running time, shown in the last row of the table.

#### Table 7.9: Code size and running time (checkers and core)

<table>
<thead>
<tr>
<th>Name</th>
<th>Approx. Size (Scala LOC)</th>
<th>Avg. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password Leakage Checker</td>
<td>120</td>
<td>50ms</td>
</tr>
<tr>
<td>OAuth Token Leakage Checker</td>
<td>120</td>
<td>50ms</td>
</tr>
<tr>
<td>Hiding-Icon Checker</td>
<td>40</td>
<td>50ms</td>
</tr>
<tr>
<td>Crypto Library Misconfiguration Checker</td>
<td>109</td>
<td>50ms</td>
</tr>
<tr>
<td>SSL/TLS Misconfiguration Checker</td>
<td>62</td>
<td>20ms</td>
</tr>
<tr>
<td>Data Leakage Checker</td>
<td>73</td>
<td>50ms</td>
</tr>
<tr>
<td>Intent Injection Checker</td>
<td>23</td>
<td>100ms</td>
</tr>
<tr>
<td>Core Framework</td>
<td>46,345</td>
<td>440s</td>
</tr>
</tbody>
</table>

### 7.6 Comparison with Existing Tools

This dissertation uses three benchmarks, DroidBench, ICC-Bench and NativeFlowBench to compare Argus-SAF with most well-known static analysis tools for Android: FlowDroid [19], IcctA [26], and DroidSafe [25]. The benchmark test suites consist of hand-crafted apps designed to test specific analysis features. Since those apps are hand-crafted, the ground truth is known, which allows us to compute metrics such as precision and recall. However, one needs to keep in mind that
these metrics are not representative of the performance of the tools on real-world apps. They can only be used for comparison purposes.

DroidBench [83] is a benchmark test suite published by the FlowDroid team, which consists of Android apps for evaluating information-flow analysis. The version this dissertation used contains 21 apps, including inter-component communication challenges as well as inter-app communication challenges. ICC-Bench [84] contains 24 apps for testing various Intent communication, RPC communication, static fields tracking capabilities as well as multi-app analysis capabilities. The test apps in ICC-Bench are categorized into four parts each of which focuses on one type of ICC: Part A involves various types of intent handling: explicit intent target finding, implicit intent target finding (via matching action, categories, data, and type), and dynamically registered component handling, etc.; Part B focuses on the accuracy of the analysis by including a variety of scenarios where certain Intent-related information flow paths do or do not exist, and the capability to handle IntentService\(^3\) and Stateful ICC; Part C tests the ability of handling different types of RPC communications; Part D contains one comprehensive test case to test whether the tool can handle complex scenarios where data may flow via various communication channels. ICC-Bench is designed by us and publicly available [84]. The apps in these test suites are not crafted to favor a particular tool. They represent common scenarios one will find when reasoning about the relevant security issues.

The author run each tool on each test app to check if the tool can report the correct data leak paths. The detailed comparison of the performance of FlowDroid/IccTA, DroidSafe and ArgusSAF on DroidBench and ICC-Bench is available in Table 7.10 and Table 7.11, respectively. The results are shown regarding True Positive (O), False Positive (*), and False Negative (X), if any. If a test app contains multiple data leak paths, the result is shown for each of them. As an example, in Table 7.10 for the ActivityCommunication2 app of DroidBench, both IccTA and ArgusSAF have an entry “OO*,” which indicate that these tools detect two paths (i.e., OO) but also report one false path (i.e., *). The author observes that ArgusSAF outperforms IccTA and DroidSafe on both benchmarks. The sole false negative of ArgusSAF for DroidBench is due to ArgusSAF

\(^3\)IntentService is a special Service, which receives an Intent and executes the corresponding operation in background.
Table 7.10: Results on DroidBench

<table>
<thead>
<tr>
<th>App Name</th>
<th>Argus-SAF</th>
<th>FlowDroid</th>
<th>ICC TA</th>
<th>DroidSafe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Inter-component Communication (ICC)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication2</td>
<td>OO*</td>
<td>OO*</td>
<td>OO</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication3</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication4</td>
<td>OO</td>
<td>OO*</td>
<td>OO</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication5</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication6</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication7</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ActivityCommunication8</td>
<td>OO</td>
<td>OO*</td>
<td>OO</td>
<td></td>
</tr>
<tr>
<td>BroadcastTaintAndLeak1</td>
<td>OO</td>
<td>OO</td>
<td>OX</td>
<td></td>
</tr>
<tr>
<td>ComponentNotInManifest1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EventOrdering1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>IntentSink1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>IntentSink2</td>
<td>O</td>
<td>O</td>
<td>O**</td>
<td></td>
</tr>
<tr>
<td>IntentSource1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>ServiceCommunication1</td>
<td>O</td>
<td>X</td>
<td>O**</td>
<td></td>
</tr>
<tr>
<td>SharedPreferences1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Singleton1</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>UnresolvableIntent1</td>
<td>OOO</td>
<td>OOO</td>
<td>OOO</td>
<td></td>
</tr>
</tbody>
</table>

### Sum, Precision and Recall — ICC

<table>
<thead>
<tr>
<th></th>
<th>Argus-SAF</th>
<th>FlowDroid</th>
<th>ICC TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>O, higher is better</td>
<td>22</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>*, lower is better</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>X, lower is better</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Precision p = O/(O + *)</td>
<td>96%</td>
<td>86%</td>
<td>85%</td>
</tr>
<tr>
<td>Recall r = O/(O + X)</td>
<td>96%</td>
<td>83%</td>
<td>96%</td>
</tr>
<tr>
<td>F-measure 2pr/(p + r)</td>
<td>96%</td>
<td>85%</td>
<td>90%</td>
</tr>
</tbody>
</table>

### Inter-app Communication (IAC)

<table>
<thead>
<tr>
<th></th>
<th>Argus-SAF</th>
<th>FlowDroid</th>
<th>ICC TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Echoer</td>
<td>0</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>StartActivityForResult1</td>
<td>O</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Precision and Recall — IAC

<table>
<thead>
<tr>
<th></th>
<th>Argus-SAF</th>
<th>FlowDroid</th>
<th>ICC TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision p = O/(O + *)</td>
<td>74%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Recall r = O/(O + X)</td>
<td>100%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F-measure 2pr/(p + r)</td>
<td>85%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

O = True Positive, * = False Positive, X = False Negative.
Table 7.11: Results on ICC-Bench

<table>
<thead>
<tr>
<th>App Name</th>
<th>Argus-SAF</th>
<th>FlowDroid <em>ICC_TA</em></th>
<th>DroidSafe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Part A — Intent Addressing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Explicit1</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Intent_Implicit_Action</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_Implicit_Category</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_Implicit_Data1</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_Implicit_Data2</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_Implicit_Mix1</td>
<td>OOO</td>
<td>OOO</td>
<td>XXX</td>
</tr>
<tr>
<td>Intent_Implicit_Mix2</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_DynRegisteredReceiver1</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_DynRegisteredReceiver2</td>
<td>OO*</td>
<td>OO*</td>
<td>XX</td>
</tr>
<tr>
<td><strong>Part B — Intent Data Flow Tracking</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Explicit_NoSrc_NoSink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Explicit_NoSrc_Sink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Explicit_Src_NoSink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Implicit_NoSrc_NoSink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Implicit_NoSrc_Sink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Implicit_Src_Sink</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Intent_Implicit_NoSrc_NoSink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Implicit_NoSrc_Sink</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_Implicit_Src_Sink</td>
<td>OO</td>
<td>OO</td>
<td>XX</td>
</tr>
<tr>
<td>Intent_Stateful</td>
<td>OOO</td>
<td>OOO</td>
<td>OXX</td>
</tr>
<tr>
<td><strong>Part C — RPC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC_LocalService</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>RPC_MessengerService</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RPC_AIDL</td>
<td>O</td>
<td>X</td>
<td>X***</td>
</tr>
<tr>
<td>RPC_ReturnSensitive</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td><strong>Part D — Mixed</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intent_RPC_Comprehensive</td>
<td>O</td>
<td>X</td>
<td>X******</td>
</tr>
</tbody>
</table>

**Sum, Precision and Recall — ICC-Bench**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>O, higher is better</td>
<td>97%</td>
<td>97%</td>
<td>10%</td>
</tr>
<tr>
<td>*, lower is better</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>X, lower is better</td>
<td>0</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>Precision p = O/(O + *)</td>
<td>97%</td>
<td>97%</td>
<td>10%</td>
</tr>
<tr>
<td>Recall r = O/(O + X)</td>
<td>100%</td>
<td>90%</td>
<td>3%</td>
</tr>
<tr>
<td>F-measure 2pr/(p + r)</td>
<td>98%</td>
<td>93%</td>
<td>5%</td>
</tr>
</tbody>
</table>

O = True Positive, * = False Positive, X = False Negative.
not modeling Java Singleton. The false positives of Argus-SAF on both benchmarks are due to context-insensitive inter-component data flow handling and the rudimentary string analysis.

Although IccTA’s website claims that the tool is capable of performing inter-app analysis by combining multiple apks into a single apk, in our experience their ApkCombiner failed to combine the inter-app communication apps in DroidBench. Thus the author could not obtain any result from IccTA on the inter-app communication experiment for DroidBench. Moreover, the ICC-Bench apps have all been updated to the newest Android version (Android 7.1.1), representing the current Android application design with the new permission acquiring mechanism introduced by Android M and later versions. Neither IccTA nor Argus-SAF had the problem of detecting data leaks in the new version of apps after the author manually updated some of their dependency libraries and Android sdk. However, DroidSafe could not handle the new design even after update the dependency libraries and Android SDK, and that is the reason DroidSafe is shown to be missing so many paths over ICC-Bench test suite.

For evaluation purpose, the author designed NativeFlowBench [85] since there is no existing benchmark for evaluating inter-language data-flow analysis capability of Android static analysis tools. NativeFlowBench contains a set of hand-crafted apps designed to test specific analysis features. Since those apps are hand-crafted, the ground truth is known, and metrics like precision and recall can be computed. NativeFlowBench contains 22 apps categorized in three parts: Part A focuses on inter-language dataflow analysis challenges: native source and sink finding, native method to native function resolving, JNI library function modeling, native dataflow analysis with Java objects, etc. Part B focuses on the native Activity resolving. Part C focuses on inter-component communication between Java and native components. The apps in these test suites are not crafted to favor a particular tool. They present common scenarios one will find when reasoning about the relevant security issues.

The effectiveness comparison of Argus-SAF with all other major Android static analysis tools: FlowDroid [19], IccTA [26], DroidSafe [25]. The author run each tool against each of the benchmark apps to check if the tool can report the correct data leak paths, and the detailed comparison is reported in Table 7.12. The results are shown regarding True Positive (O), False Positive (*) and
Table 7.12: Results on NativeFlowBench

<table>
<thead>
<tr>
<th>App Name</th>
<th>Argus-SAF</th>
<th>FlowDroid</th>
<th>IccTA</th>
<th>DroidSafe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part A: Inter-language Dataflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native_source</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_nosource</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native_source_clean</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>native_leak</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_leak_dynamic_register</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_dynamic_register_multiple</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_noleak</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native_noleak_array</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>native_method_overloading</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native_multiple_interactions</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_multiple_libraries</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_complexdata</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_complexdata_stringop</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>native_heap_modify</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_set_field_from_native</td>
<td>OO</td>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>native_set_field_from_arg</td>
<td>OO</td>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>native_set_field_from_arg_field</td>
<td>OO</td>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>Part B: Native Activity Resolve</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native_pure</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_pure_direct</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>native_pure_direct_customized</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Part C: Inter-component Communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>icc_javatov native</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>icc_nativetov java</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Sum, Precision and Recall</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O, higher is better</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>*, lower is better</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>X, lower is better</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Precision $p = \frac{O}{O + \ast}$</td>
<td>90.5%</td>
<td>0.0%</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Recall $r = \frac{O}{O + X}$</td>
<td>100%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>F-measure $2pr/(p + r)$</td>
<td>95.0%</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

O = True Positive, * = False Positive, X = False Negative.

IccTA is applied for handle part C: Inter-component Communication.
False Negative (X), if any. If an app has more than one leakage path, then the result is shown for each of them. Not surprisingly, Argus-SAF outperforms all other tools as none of the existing Android static analysis tools have inter-language analysis capability. DroidSafe is outdated and failed to analyze any of the benchmark apps. Amandroid and FlowDroid both identified one false path at native_source_clean. This is caused by their conservative model for native method calls – if one of the argument is tainted all other arguments will also be considered as tainted. IccTIA failed to handle the inter-component communication cases due to the lack of native code resolution. Argus-SAF has false alarm on native_noleak_array because Argus-SAF cannot distinguish different index of a java array. Argus-SAF has false alarm on native_complexdata_stringop because Argus-SAF does not do precise string analysis.
CHAPTER 8: CONCLUSIONS

This dissertation proposes a static analysis approach to assist security vetting of Android applications. Compare to all other related approaches, the approach proposed in this thesis has the advantage of comprehensive coverage of Android-specific behaviors, modularity, and efficiency. Comprehensive Android behavior coverage brings precision, crucial for a useable static vetting system. Modularity makes the static analysis framework more flexible and easy to be extended and customized to address emerging security issues. Efficiency is essential for practical use.

This dissertation made available a comprehensive model of handle Android component-based nature as discussed in Chapter 4 and proposed a new algorithm to address Android application NDK/JNI-awareness in Chapter 5. Those approaches form a comprehensive

This dissertation describes a static analysis framework, Argus-SAF, that precisely track the control and data-flow in an inter-component, inter-language and NDK/JNI-aware fashion, and can compute an abstraction of the app’s behavior in the forms of data-flow graph ($DFG$) and data dependence graph ($DDG$). As a general framework, Argus-SAF can be easily extended to achieve a large variety of specialized security analyses. The experiment results showed that Argus-SAF scales well.

This dissertation presented various security analysis checkers implemented on top of Argus-SAF. Compare with the Argus-SAF core framework they have an ignorable footprint. Their detection result indicates the Argus-SAF can be effectively applied to the real-world security vetting scenario. However, the currently available checkers are not rich enough to cover a broad range of Android security issues. Therefore, it remains future work to extensively implement new security analysis plugins for Argus-SAF to provide better security issue coverage.

There are several limitations for Argus-SAF that might cause imprecisions in its analysis, such as, preliminary string analysis, conservative treatment for unmodeled APIs and unhandled
language features. More work is needed in the future to further improve the effectiveness and precision of Argus-SAF.

Static analysis is one of the fundamental steps towards complete automation of Android application security vetting. Once the static features are generated, the next question to ask is whether the given application is truly malicious or vulnerable. The precise, general and efficient static analysis engine in Argus-SAF provides a promising outlook to generate rich application behavior of security vetting system.
REFERENCES


APPENDIX A: COPYRIGHT PERMISSIONS

Copyright permission for material reuse can be seen below. The publishers state that authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.
ACM Information for Authors

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

**CHOOSE**

Authors have the option to choose the level of rights management they prefer. ACM offers three different options for authors to manage the publication rights to their work.

- Authors who want ACM to manage the rights and permissions associated with their work, which includes defending against improper use by third parties, can use ACM’s traditional copyright transfer agreement.
- Authors who prefer to retain copyright of their work can sign an exclusive licensing agreement, which gives ACM the right but not the obligation to defend the work against improper use by third parties.
- Authors who wish to retain all rights to their work can choose ACM’s author-pays option, which allows for perpetual open access through the ACM Digital Library. Authors choosing the author-pays option can give ACM non-exclusive permission to publish, sign ACM’s exclusive licensing agreement or sign ACM’s traditional copyright transfer agreement. Those choosing to grant ACM a non-exclusive permission to publish may also choose to display a Creative Commons License on their works.

**POST**

Authors can post the accepted, peer-reviewed version prepared by the author-known as the "pre-print"-to the following sites, with a DOI pointer to the Definitive Version of Record in the ACM Digital Library.

- On Author’s own Home Page and
- On Author’s Institutional Repository and
In any repository legally mandated by the agency funding the research on which
the work is based and

On any non-commercial repository or aggregation that does not duplicate ACM
tables of contents, i.e., whose patterns of links do not substantially duplicate an
ACM-copyrighted volume or issue. Non-commercial repositories are here
understood as repositories owned by non-profit organizations that do not charge a
fee for accessing deposited articles and that do not sell advertising or otherwise
profit from serving articles.

**DISTRIBUTE**

Authors can post an *Author-Izer* link enabling free downloads of the Definitive Version of the
work permanently maintained in the ACM Digital Library

- On the Author's own Home Page or
- In the Author's Institutional Repository.

**REUSE**

Authors can reuse any portion of their own work in a new work of *their own* (and no fee is
expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital
Library are included.

- Contributing complete papers to any edited collection of reprints for which the
author is *not* the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a
dissertation as long as citations and DOI pointers to the Versions of Record in the ACM
Digital Library are included. Authors can use any portion of their own work in
presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are sold to students require permission
and possibly a fee.

**CREATE**

ACM's copyright and publishing license include the right to make Derivative Works or new
versions. For example, translations are "Derivative Works." By copyright or license, ACM
may have its publications translated. However, ACM Authors continue to hold perpetual
rights to revise their own works without seeking permission from ACM.

- If the revision is minor, i.e., less than 25% of new substantive material, then the
work should still have ACM's publishing notice, DOI pointer to the Definitive
Version, and be labeled a "Minor Revision of"

- If the revision is major, i.e., 25% or more of new substantive material, then ACM
considers this a new work in which the author retains full copyright ownership
(despite ACM's copyright or license in the original published article) and the author
need only cite the work from which this new one is derived.

Minor Revisions and Updates to works already published in the ACM Digital Library are
welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

**RETAIN**

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy,
including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images
intended for exploitation in other contexts
- All patent and moral rights
Ownership and control of third-party permissions to use of software published by ACM

Have more questions? Check out the FAQ.