AFRL-RI-RS-TR-2016-288

DROIDSAFE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DECEMBER 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

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

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-288 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2016	FINAL TECHNICAL REPORT	FEB 2012 – JUN 2016

4. TITLE AND SUBTITLE

DROIDSAFE

5a. CONTRACT NUMBER
FA8750-12-2-0110

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Jeff Perkins, Michael Gordon

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
97

5f. WORK UNIT NUMBER
74

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
77 Massachusetts Avenue, Build E19-750
Cambridge, MA 02139-4307

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-288

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The DroidSafe project developed effective program analysis techniques and tools to uncover malicious code in Android mobile applications. The core of the system is a static information flow analysis that reports the context under which sensitive information is used. The DroidSafe project invested significant time developing a comprehensive semantic model of Android run-time behaviors alongside the analysis to achieve acceptable precision, accuracy, and scalability for real-world Android applications.

15. SUBJECT TERMS

Comprehensive semantic modeling, Static Analysis for Android, Provably safe applications.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK WILLIAMS**
U	U	U	UU	115	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-4560**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

The DroidSafe project developed effective program analysis techniques and tools to uncover malicious code in Android mobile applications. The core of the system is a static information flow analysis that reports the context under which sensitive information is used. For example, "Application A has the potential to send location information to network address 128.6.21.6 on 'Button B' press". The DroidSafe project invested significant time developing a comprehensive semantic model of Android run-time behaviors alongside the analysis to achieve acceptable precision, accuracy, and scalability for real-world Android applications. The combined system has been demonstrated to be the most precise and accurate information flow analysis for Android applications. The analysis results can be used to automatically check applications for security policy violations, and the results can help a human analyst inspect sensitive behaviors of an app, increasing accuracy and throughput of application vetting. For each of the last six APAC engagements to date, the DroidSafe team has been unsurpassed in malware diagnosis accuracy and human-analysis diagnosis throughput.

To address subtle functional correctness bugs and vulnerabilities, we have produced a formal model of (part of) the Android system capable of supporting proofs of functional correctness of simple but non-trivial Android apps.

We also explored how to move precise static analysis and verification techniques from specialized research tools to an approach that can feasibly be adopted by programmers in the real world. The critical issue in doing so is the lack of precision that such analyses inevitably encounter when analyzing programs that occur in practice. Our hypothesis is that, in existing programs, only a small percentage of the code (the code's dark corners) is responsible for this lack of precision. We found that in many cases relatively straightforward code modifications can yield code that is analyzable and still practical.

2.0 METHODS, ASSUMPTIONS, AND PROCEDURES

For all the research we performed in this program, we adopted an experimental approach driven by the evaluation scenarios. Our overarching goal was to produce systems that could aid an analyst (or make automated decisions) in settings much like the evaluation scenario, i.e., a trusted analyst inspecting an unknown Android application with no code modifications or implementation artifacts relating to the analyst tools. The metrics employed to evaluate our progress towards this goal are analyst throughput and accuracy, with second-order metrics of reduction in false positives and false negatives in the automated analysis results.

For our research papers and program presentations, we chose to evaluate our developed systems on both previously released micro-application suites and the applications developed by the APAC Challenge Teams. We also reported our analyst throughput (average manual analysis time) and diagnosis accuracy for each of the APAC engagements.

During the course of the program, we devoted a major effort to the packaging, testing, and usability of the system. Our releases included both unit and system tests (comprising our regression test). We also devoted major effort to documentation included with system deliverables, including walk-throughs and screencasts. Finally, a major effort was devoted to open-sourcing and releasing to the public all of our analyses, models, and tools.

All of the systems that we developed run on open source infrastructure (e.g., Linux) and do not require proprietary software to build and run.

3.0 INTRODUCTION

Application marketplaces provide a centralized location for application developers to place applications for potential users. This new model of software acquisition has been shown, in existing application marketplaces such as the iPhone App Store and the Android Market, to promote the effective development of sophisticated software that satisfies the needs of a broad range of users at unprecedentedly low cost. This model therefore holds out the promise of revolutionizing software acquisition for a broad range of organizations and not just individual consumers (which current application marketplaces target).

A critical weakness with current application marketplaces, however, is that users have no way to be sure that the applications in the marketplace are free of malware. This problem can lead to the widespread exploitation of users with consequences that include, for example, theft or the compromise of information that should remain confidential. The potential presence of malicious malware and the resulting possibility of widespread security vulnerabilities can even eliminate the ability of service organizations (such as the United States Department of Defense) with stringent security needs to use application marketplaces.

During this program, we developed analyses and tools to improve the throughput and accuracy of application inspection and vetting. Our scenario is one in which an untrusted application is delivered to a trusted human analyst for review. Our tools enable an analyst to gain a rapid understanding of how the application interacts with sensitive API calls and verify functional correctness with formal proofs. Also, we have taken significant steps towards automatically identifying portions of the program that introduce imprecision in static analysis, so that the program can be modified (through annotations or functionality swapping) or more expensive analysis can be applied to minimize imprecision. The following sections provide more specifics on the three branches of our work.

3.1 Understanding Sensitive API Call and API Information Usage

Android applications are written in a type-safe language (Java) whose design eliminates the possibility of the low-level vulnerabilities to which most existing analyses are directed. Many Android malwares instead focus on insecure Android API call sequences which, for example, leak confidential information to an untrusted external agent or corrupt locally stored information. During this program, we designed analyses and analyst tools that aid in understanding how a third-party Android application interacts with the API and how it uses API-derived data. Through our exceptional performance on the APAC evaluations, we have demonstrated that our analyses and tools

enable a major increase in human-analyst-assisted diagnosis accuracy and throughput.

We developed a system, DroidSafe, a static information flow analysis tool that reports potential leaks of sensitive information in Android applications. DroidSafe combines a comprehensive, accurate, and precise model of the Android run-time with static analysis design decisions that enable the DroidSafe analyses to scale to analyze this model. This combination is enabled by accurate analysis stubs, a technique that enables the effective analysis of code whose complete semantics lies outside the scope of Java, and by a combination of analyses that together can statically resolve communication targets identified by dynamically constructed values such as strings and class designators.

Furthermore, we developed a plugin for the Eclipse IDE that represents to the analyst the results of our analyses in a straightforward and usable manner. Employing our full system, the DroidSafe team achieved both the highest diagnosis accuracy and the highest manual analysis throughput for the last 6 APAC engagements. We also demonstrated that our analysis is unmatched in information flow accuracy and precision by comparing it to previous state-of-the-art tools.

3.2 Verifying Functional Correctness

Currently, the functional correctness of an application (whether it does the right thing) is usually validated by testing. However, testing can only cover a small fraction of the possible behaviors and can miss malware that is triggered only in certain places or at certain times. For example, a malicious route-planning application could lead a platoon down dangerous routes when used in a certain geographic area; this behavior would be difficult to discover with testing.

During this program, we developed foundational models and tools to aid a trusted analyst with the task of verifying the correctness (or adherence to a functional specification) of a third-party Android application. Our work employs the ACL2 theorem prover to formally model the Android platform and to formally verify Android apps. Our approach allows the verification of the full functional correctness of apps (e.g., that a calculator app computes the correct numeric result) as well as security properties (e.g., that an app only sends data to certain URLs). Verifying an app with our system provides high assurance that it satisfies its specification. A major motivation for this work is to detect or prove the absence of "functional malware", malicious app functionality that is triggered by complex conditions on state and whose malicious action is to cause the app to calculate the wrong results or otherwise behave incorrectly, unbeknownst to the user.

Android is an event-driven system. Our formal model is an executable simulator of a growing subset of the Android platform, and app proofs are done by automated symbolic execution of the app's event handlers using the formal model. By induction, we prove that an app satisfies an invariant, including the correctness properties of interest, for all possible sequences of events. To our knowledge, our formal Android model is the most detailed and our Android app verification is the most thorough, compared to other approaches.

3.3 Identifying Sources of Imprecision in Implementations

Despite our major advances in analysis precision during this program, there are still sources of imprecision that cause either 1) unacceptable numbers of false positive alarms or 2) the use of unsound techniques that may leave errors uncovered. Our hypothesis is that, in existing programs, only a small percentage of the code is responsible for this lack of precision.

We investigated this hypothesis on a set of widely used open source C programs that operate on untrusted inputs (such as direct network connections or files downloaded from the network). Our goal is to create versions of the programs for which we can verify memory safety. Examples of memory safety problems include out of bounds accesses, null pointer dereferences, and reading uninitialized values. The analysis would verify the absence of these properties in the modified versions of the programs. We are focusing on memory safety because it is a well-defined goal and because memory safety errors are the cause of many important security vulnerabilities.

Our investigation had very encouraging results. In many cases, we were able to make relatively simple changes to the programs that allowed them to verify. We also learned a substantial amount about how a verification tool needs to operate and the features it needs if it to be of practical use to programmers.

One critical issue in applying verification is identifying the parts of the code base that introduce analysis imprecisions. We added a *logical abduction* feature to our tool to identify the problematic parts of the code base. Logical abduction is a complementary form of logical reasoning to deduction. It is a form of backward logical reasoning, which allows inferring likely premises from a given conclusion. Given we know the desired outcome (e.g., this variable is not null), logical abduction allows us to identify the additional property we need to reach the desired conclusion.

While our tool still requires some significant enhancements, we feel that we have identified an approach which could be successfully applied to moderate programs by their developers in a reasonable time frame.

3.4 Accomplishments

Here we highlight the major accomplishments of the DroidSafe project over the course of the APAC program:

- **Most precise and accurate global static information flow analysis for Android**: In [1], we demonstrated that the DroidSafe static analysis system (discussed in Chapter 4) is the most accurate and precise global static information flow analysis system to date. This system reports all malicious flows in 24 APAC applications with explicit leaks of sensitive data, while prior state-of-the-art analysis reported only 10% of the leaks.
- **Most comprehensive and accurate Android semantic model for static analysis**: In [1] we demonstrated that our model of the Android run-time and API is the most comprehensive and accurate model appropriate for static analysis. The model represents the semantics of the Android run-time and API. See Chapter 4 for a full discussion.

- **Most detailed Android formal model, enabling most rigorous verification of app functionality**: In [2] we demonstrated a system that enables an analyst to verify application functionality via the ACL2 theorem prover. By induction, an analyst can prove that an app satisfies invariants regarding correctness and security properties. See Chapter 7 for full discussion.
- **Outstanding diagnosis accuracy and throughput on APAC engagements**: Our team achieved the highest diagnosis accuracy and highest human-analysis throughput for the last 6 APAC engagements. This achievement was enabled by our tools, which enable human analysts to achieve a rapid understanding of the security-sensitive behavior of applications.
- **Implemented logical abduction**: We added a logical abduction feature to the Concord memory safety verification tool. The tool guides the user to the missing properties required to complete the verification.
- **Open-sourced and released static analysis tools**: Our team has open-sourced and released the static analysis, semantic model, and the Eclipse plugin. Our tools are being used by dozens of researchers across the world.
- **Technology transition to BBN's Artemis tool**: The static analysis described in Chapters 4 - 6 has been transitioned by Raytheon BBN Technologies into an alpha system being tested by various branches of the US Department of Defense. We hope that work will continue on the Artemis project so that our work will be used by analysts in the field.
- **Publications**: During the course of the project, we published outstanding papers in top research conferences [1, 2, 3, 4, 5, 6]. Our papers have received over 100 citations as of June 2016.

4.0 DROIDSAFE STATIC ANALYSIS SYSTEM RESULTS AND DISCUSSION

4.1 Introduction

Sensitive information leaks, as implemented by malicious or misused code (such as advertising libraries) in Android applications, constitute one of the most prominent security threats to the Android ecosystem [7, 8]. Android currently supports a coarse-grain information security model in which users grant applications the right to access sensitive information [9]. This model has been less than successful at eliminating information leaks [7], in part because many applications need to legitimately access sensitive information, but only for a specific limited purpose — for example, an application may legitimately need to access location information, but only with the right to send the information to authorized mapping servers.

Motivated by this problem, researchers have developed a variety of systems that are designed to analyze or explore the information flows in Android applications. Dynamic analysis frameworks execute instrumented versions of Android applications and observe behaviors [10, 11, 12]. Potential downsides of this approach include missed information flows that are not exercised during testing and, in some cases, the ability of the malicious application to detect the testing and modify its behavior to avoid exercising the malicious flow [10]. They also suffer from denial-of-service attacks if malware is activated during application execution and the application is killed or functionality is disabled.

Static analysis frameworks attempt to analyze the application before it executes to discover all potential sensitive flows [13, 14, 15, 16, 17, 18, 19]. Standard issues that complicate the construction of such systems are the challenges of 1) scaling to large applications and 2) maintaining precision in the analysis such that it does not report too many flows that do not actually exist in the application. One particularly prominent issue with developing static analyses for Android applications is the size, richness, and complexity of the Android API and runtime, which typically comprises multiple millions of lines of code implemented in multiple programming languages. Because sensitive flows are often generated by complex interactions between the Android application, API, and runtime, any static analysis must work with an accurate model of this runtime to produce acceptably accurate results.

Accuracy is critical for a static analysis seeking to calculate security properties of an application; any inaccuracies in the execution model provide a motivated attacker with the opportunity to insert malicious flows that will not be captured by an analysis. Also, imprecision in a model could lead to results that are unusable due to too many false positives; another target for a motivated attacker. To the best of our knowledge, the difficulty of obtaining an acceptably accurate and precise Android

model has significantly limited the ability of previous systems to successfully detect the full range of malicious information flows in Android applications.

4.1.1 DroidSafe

For the APAC program we developed and present in this chapter a new system, DroidSafe, for accurately and precisely analyzing sensitive explicit information flows in large, real-world Android applications. DroidSafe tracks information flows from sources (Android API calls that inject sensitive information) to sinks (Android API calls that may leak information). We evaluate DroidSafe on 24 complete real-world Android applications that, as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program, have been augmented with malicious information flow leaks by three hostile Red Team organizations. The goal of these organizations was to develop information leaks that would either evade detection by static analysis tools or overwhelm static analysis tools into producing unacceptable results (by, for example, manipulating the tool into reporting an overwhelming number of false positive flows). DroidSafe accurately detects all of the 69 malicious flows in these applications (while reporting a manageable total number of flows). A current state-of-the-art Android information-flow analysis system, FlowDroid [14] + IccTA [20], in contrast, detects only 6 of the 69 malicious flows, and has a larger ratio of total flows reported to true malicious flows reported.

Additionally, we evaluate DroidSafe on DROIDBENCH, a suite of 94 Android information-flow benchmarks from the developers of FlowDroid and IccTA, and report the highest accuracy (most actual flows reported) and highest precision (fewest false positive flows reported) for this benchmark suite to date, 94.3% and 87.6% respectively. DroidSafe fails to report only the implicit flows in DROIDBENCH. Finally, we evaluate DroidSafe on a suite of 40 Android explicit information-flow benchmarks developed by us to add coverage to DROIDBENCH; DroidSafe achieves 100% accuracy and precision for the suite, compared to FlowDroid + IccTA's 34.9% accuracy and 79.0% precision.

One advantage of working with applications that contain known inserted malicious flows is the ability to characterize the accuracy of our analysis (i.e., measure how many malicious flows DroidSafe was able to detect). As these results illustrate, DroidSafe implements an analysis of unprecedented accuracy and precision. To the best of our knowledge, DroidSafe provides the first usable information-flow analysis for Android applications [13, 14, 15, 16, 17, 18, 19, 20].

4.1.2 The Android Model and Analysis Co-design

Given the extensive and complex interactions between the Android execution environment and Android applications, an accurate and precise information-flow analysis for Android applications requires a comprehensive and accurate model of the Android environment. To obtain such a model, we started with the Android Open Source Project (AOSP) [21] implementation, which contains a Java implementation of much of the Android environment. The goal was to maximize accuracy and precision by directly analyzing as comprehensive a model of Android as feasible.

As we worked with AOSP, it quickly became apparent that the size and complexity of the Android environment made it necessary to develop the model and the analysis together as an integrated whole, with the design decisions in the model and the analysis working together synergistically to enable an effective solution to the Android static information-flow analysis problem. The result is the first accurate and precise model of the Android environment and the first analysis capable of analyzing such a model.

Accurate Analysis Stubs: While the AOSP provides an accurate and precise model for much of Android, it is missing critical parts of the Android runtime. And for good reason — it is currently not practical to implement much of the Android runtime in Java. We therefore developed a novel technique, *accurate analysis stubs*, to enable the effective analysis of code whose full semantics lies outside the scope of AOSP. Each stub is written in Java and only incompletely models the runtime behavior of the modeled code. But the semantics of the stub is complete for the abstractions that the analysis deploys (in this case points-to and information-flow analyses). Examples of semantics missing in the AOSP and added via accurate analysis stubs include native methods; event callback initiation with accurate context; component life-cycle events; and hidden state maintained by the Android runtime and accessible to the application only via the Android API.

Accurate analysis stubs simplify the development of the analysis — they eliminate any need to develop a library of method summaries written in a different specification language [22, 3], any need to conservatively hard code policies within the analysis that attempt to compensate for the missing semantics [14], or any need to analyze code written in multiple languages. They also simplify the development of the model — they enable the developers of the model to work flexibly and efficiently within the familiar implementation language. And they support the use of sophisticated language features such as inheritance, polymorphic code reuse, exceptions, and threads, all of which promote effective engineering of stubs that accurately and precisely model key aspects of the Android environment.

In addition to code, accurate analysis stubs also support the use of Java objects to model otherwise hidden state maintained by the Android runtime. Examples of such state include Android Activity saved state, the global Application object, Intent, Parcel, shared preferences, and the file system. Accurate analysis stubs enable DroidSafe to be the first analysis to accurately model these key Android features.

The AOSP implementation overlaid with our accurate analysis stubs represents our model of the Android API and runtime. We call this model the Android device implementation (ADI). Each application is analyzed in the context of the ADI, approximately 1.3 MLOC. For the information-flow analysis, we manually identified and classified 4,051 sensitive source methods and 2,116 sensitive sink methods in the Android API.

Scalable, Precise Points-To Analysis: Both our Android model and Android applications heavily use sophisticated language features (such as inheritance and polymorphic code reuse) that are known to significantly complicate static program analyses. To preserve acceptable precision, DroidSafe therefore deploys a modern global *object-sensitive* points-to analysis specifically designed to analyze code that uses such features [23]. DroidSafe further enhances scalability by identifying classes that are not relevant to the information flow and eliminating object sensitivity

for instances of these classes. This Android-specific optimization enables our global points-to analysis to achieve a context depth greater than what was achieved in prior work [24, 23], delivering a precise analysis appropriately tailored for solving Android information-flow problems.

Flow-Insensitive Analyses: DroidSafe employs a flow-insensitive information-flow analysis. Many interactions between Android applications and the Android environment are mediated by asynchronous callbacks. Because our DroidSafe implementation uses flow-insensitive points-to and information-flow analyses, it accurately considers all possible runtime event orderings that asynchronous callbacks can trigger. Developers of flow sensitive analyses, in contrast, have had difficulty obtaining a model that correctly exposes all of these event orderings to a flow sensitive analysis (see Section 4.9). Because the analysis does not consider all event orderings, it may miss sensitive flows.

Critically, flow insensitivity also enables the analysis to scale to analyze an accurate and precise Android model and therefore to accurately and precisely track information flows through the Android environment. Scalability issues restrict flow-sensitive analyses to significantly less accurate and precise Android models characterized by imprecise conservative flow summaries and/or blanket policies for Android API methods [15, 14, 17]. Our results show that the ability to analyze an accurate and precise model more than makes up for any loss of precision caused by flow insensitivity (see Section 4.9).

Static Communication Target Resolution: Information flows in Android apps may involve inter-component (between application components) and inter-application (between separate installed apps) communication; communication targets are identified by dynamically constructed values (such as String, Uri, and class designators) packaged in an Intent object.

To precisely analyze such flows, DroidSafe combines 1) accurate analysis stubs, 2) an internal representation of all defined IntentFilter registrations, 3) an analysis of operations that construct strings; this analysis delivers regular expressions that accurately summarize the strings that the application will construct when it runs, 4) a novel points-to analysis that precisely tracks Strings, and 5) algorithms that rewrite the DroidSafe intermediate representation to directly invoke resolved targets. Because DroidSafe works with a comprehensive model of the Android environment, it supports precise resolution of communication targets whose identification (typically via an Intent) involves significant interactions with the Android API.

These techniques enable DroidSafe to precisely analyze calls that start Activity components; start, stop, and bind Service components; invoke RPC calls on Service components; send and receive Service messages; broadcast messages to BroadcastReceiver components; and perform operations on ContentProvider components (shared databases). For Intent-based resolution, DroidSafe incorporates IntentFilter registrations defined both in the Android manifest and those defined programmatically in app code. We are aware of no other analysis that can provide comparable or even usable levels of accuracy or precision for all of these critical Android communication mechanisms.

4.1.3 Contributions

DroidSafe represents, for the first time, an effective point in the overall Android information-flow design space. Our overarching contribution is the identification of this design point and the resulting DroidSafe implementation. We attribute the ability of DroidSafe to operate at this design point to: 1) the identification of a set of techniques that work well together, 2) new implementations of known program analysis techniques that enable DroidSafe to deliver an analysis of unprecedented scalability, accuracy, and robustness, 3) a set of new mechanisms that enable these techniques to work together to provide a comprehensive, accurate, and precise information-flow analysis for Android applications, and 4) significant engineering effort that delivers a comprehensive model of the Android runtime. Specific contributions include:

- **Accurate Analysis Stubs**: A novel technique that enables the rapid and accurate development of semantics missing from a source code base. Each stub is written in the language of the implementation of the API model, simplifying analysis. Stubs augment the implementation with semantics possibly incomplete for the full runtime behavior, but complete for the analysis abstractions.
- **Android Device Implementation**: A comprehensive and precise model of the Android API and runtime system implemented in Java that accurately captures the semantics of life-cycle events, callback context, external resources, and inter-component communication. The core of the ADI includes 550 manually-verified Android API classes which cover over 98% of API calls in deployed Android applications. The ADI currently models Android 4.4.3, because updating the model for Android updates is not overly burdensome. Independent analysis tools can readily employ this model.
- **Static Analysis Design Decisions**: Our analysis occupies a new design point for information-flow analysis of Android: deep object sensitivity and flow insensitivity. Flow insensitivity enables DroidSafe to accurately consider all possible event orderings. It also enables DroidSafe to scale to analyze an accurate and precise model of the Android environment, which is critical for the overall success of DroidSafe. Any loss of precision due to flow-insensitivity is more than compensated for by the analysis's ability to scale to analyze our accurate and precise Android model.
- **Static Communication Target Resolution**: A comprehensive and precise model of inter-component communication resolution in Android that links data flows between sender and target. DroidSafe includes a global `Intent` and `Uri` value resolution analysis, `IntentFilter` reasoning, and coverage of all common forms of communication. To our knowledge it is the most complete such model to date.
- **Experimental Evaluation**: An evaluation demonstrating that 1) DroidSafe achieves unprecedented precision and accuracy for the information-flow analysis of Android and 2) DroidSafe can detect malicious sensitive information leaks inserted by sophisticated, independent hostile organizations, where a current state-of-the-art information-flow analysis largely fails.
- **Full Implementation**: A full open-source implementation of DroidSafe and our ADI is available.

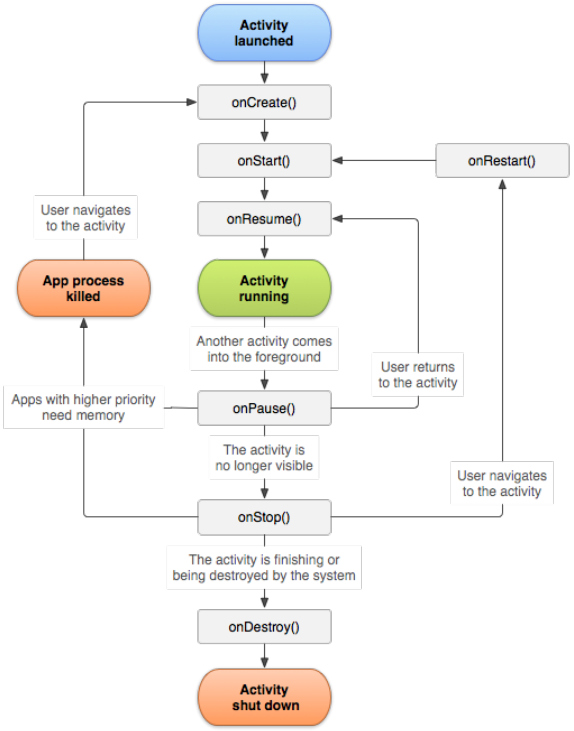

Figure 1: Android Activity Lifecycle.

4.2 Background and Problem

Android applications are implemented in Java on top of the Android API. The implementation of an application specifies handlers for the dynamic events that may occur during the execution of the application. Thus, Android applications are dynamic and event-driven by nature. Applications have multiple entry points, and interact heavily with the Android API via utility and resource access classes. The package for an Android application represents an incomplete program; the source package alone is not appropriate for analysis without an accompanying model of the Android API and runtime semantics to exercise all possible semantics in the application.

The Android API version 4.4.3 includes over 3,500 classes visible to an application developer. Analyzing the complete source code for the API is exceedingly difficult because it is implemented over multiple languages and some of the implementation is device-specific. Thus, static analysis frameworks rely on modeling the Android API semantics. Manually producing summaries for all of the application-visible methods of the Android API is daunting task that is potentially error prone. For a high-precision analysis, it is also exceedingly difficult to model all semantics of the implementation regarding memory allocation, data flows, and aliasing. A blanket policy for generating flows for all API methods would risk being too imprecise and inaccurate.

```
1 public class EventOrder extends Activity {
2   String urlPath = "";
3
4   protected void onCreate(Bundle savedInstanceState) {
5     //...
6     Intent intent = new Intent(Intent.ACTION_VIEW);
7     intent.setData(Uri.parse("http://untrusted.com" +
8                              urlPath));
9
10    startActivity(intent); //sink, if onCreate called
11                           //after onStop()
12  }
13
14  //.... Other events
15
16  protected void onStop() {
17    Location loc = <get location> //source
18    urlPath = loc.getLatitude() + "";
19  }
20 }
```

(a) Event ordering example: Green arrow shows end-to-end flow.

```
1 public class CallbackContext extends Activity {
2   String urlPath = "";
3
4   protected void onCreate(Bundle savedState) {
5     //...
6     if (savedState != null) {
7       urlPath = savedState.getDouble("LAT") + "";
8     }
9
10    Intent intent = new Intent(Intent.ACTION_VIEW);
11    intent.setData(Uri.parse("http://untrusted.com"
12                             + urlPath));
13    startActivity(intent); //sink, on 2nd activation
14  }
15
16  //.... Other events
17
18  public void onSaveInstanceState(Bundle state) {
19    Location loc = <get location> // source
20    savedState.putDouble("LAT", loc.getLatitude());
21    super.onSaveInstanceState(state);
22  }
23 }
```

(b) Callback context example: Green arrow shows end-to-end flow.

```
1  public class ICCSource extends Activity {
2    Messenger mService = null;
3
4    private ServiceConnection sc = new ServiceConnection() {
5      public void onServiceConnected(ComponentName cN,
6                                     IBinder iB) {
7        mService = new Messenger(iB);
8      }
9      public void onServiceDisconnected(ComponentName cN) {
10       mService = null;
11     }
12   };
13
14   protected void onCreate(Bundle savedState) {
15     Intent intent = new Intent("ICCServiceAction");
16     bindService(intent, sc,
17                 Context.BIND_AUTO_CREATE);
18   }
19
20   public void buttonClick(View v) {
21     double lat =<get location>.getLatitude();  //source
22     Message msg = Message.obtain(null, 0, lat, 0);
23     mService.send(msg);
24   }
25 }
```

```
1  // IntentFilter defined in manifest to accept
2  // action = "ICCServiceAction"
3  public class ICCService extends Service {
4
5    final Messenger mMessenger =
6        new Messenger(new IncomingHandler());
7
8    class IncomingHandler extends Handler {
9      public void handleMessage(Message msg) {
10       double data = msg.arg1;    //tainted
11       Intent intent = new Intent(ICCService.this,
12                                  ICCSink.class);
13       intent.putExtra("DATA", data);
14       startActivity(intent);
15     }
16   }
17
18   public IBinder onBind(Intent intent) {
19     return mMessenger.getBinder();
20   }
21 }
```

```
1  public class ICCSink extends Activity {
2    double data = 0.0;
3
4    protected void onCreate(Bundle savedState) {
5      Intent intent = getIntent();
6      data = intent.getDoubleExtra("DATA", 0.0);
7    }
8
9    public void buttonClick(View v) {
10     Log.v("ICCSink", data + "");  //sink, leak of location
11   }
12 }
```

(c) Android ICC Example: ICCSource sends message with tainted data to ICCService. ICCService forward data to ICCSink. Intermediate flows shown in grey arrows.

Figure 2: Examples of the challenges present in the static information flow analysis of Android applications.

4.2.1 Event Dispatch and Ordering

An accurate model of the event dispatch and ordering must represent all valid event orderings so that a static analysis can accurately capture possible runtime behavior. Otherwise, an attacker can hide flows in semantics not covered by the model. Android applications are composed of multiple components, each implementing one of four classes: Activity, Service, BroadcastReceiver, and ContentProvider. Each of these components has its own life-cycle defined with events for which a callback implementation can be provided. Figure 4.1 gives a partial picture of an Activity's lifecycle events and their orderings.

For example, Figure 4.2(a) provides an example of a single Activity that defines two life-cycle events. These events have the potential to run in many orders, and they are not called directly in application code. There exists a leak of sensitive information in one possible ordering, if onCreate is dispatched after onStop. This is possible if the activity is placed in the background by user interaction, and not reclaimed by the system before it is reactivated by the user.

In addition to life-cycle state orderings, components can have different launching modes that specify whether a single object should handle all activations or if a separate object is spawned for each activation. Thus memory could be shared across separate activations of a component.

4.2.2 Callback Context

An Android application defines callback handler methods that are called for dynamically-dispatched runtime events. Many event handler methods include arguments passed by the runtime to the application for processing. These arguments are generated by the runtime and could include data from the application (including tainted data), depending on the execution sequence prior to the event. We call the arguments to a callback handler its *callback context*. Figure 4.2(b) gives example of a flow through callback context. This example employs an Activity's ability to save state when it is paused, and restore that state when resumed. An accurate model must represent these possible flow connections (of which there are possibly thousands). Policies such as injecting taint for *all* callback handler arguments or connecting callback argument flows conservatively risk generating an overwhelming number of false positives (see Section 4.9).

4.2.3 Inter-component Communication (ICC)

The Android framework relies heavily on inter-component communication (ICC) to allow individual components to be independent and to better manage resources. Components initiate and connect to other components via android. content.Intent objects (which can themselves contain a payload). The resolution of Intent destination is complex [25]. An Intent can specify a class explicitly, or implicitly allow the Android system to select a destination based on a Uri and string fields. Components register for implicit Intent delivery programmatically or via their application's XML manifest. Service components additionally allow one to send and receive messages and perform remote procedure calls.

An accurate model of Android must represent the possible flows via ICC mechanisms. Figure 4.2(c) gives an example of three components that communicate via Intent objects and Service messages. In the example, there is a flow through ICC from ICCSource through ICCService to ICCSink. In addition to representing the communication, a model must consider all possible orderings of component activations.

A blanket conservative policy to deliver Intent objects and messages to all possible targets may not be acceptable because applications are typically constructed of many components. However, statically calculating the destination of each Intent requires resolution of Intent values such as Uri strings and action strings, and reasoning about components' implicit IntentFilter registration.

4.3 Threat Model and Limitations

In our scenario the application developer (or re-packager) is malicious. This attacker seeks to exfiltrate *sensitive data* from a mobile device to her servers or to an area on the device that is unprotected so that a colluding application can perform the exfiltration.

Our definition of sensitive data includes unique device ID, sensor data (location, acceleration, etc.), file data, image data and meta-data, email and SMS messages, passwords, network traffic, and screen-shots. All of these data items are retrieved or stored via the Android API; we define sources of sensitive data as flows initiated from calls to Android API methods that we have identified (see Section 4.4).

The attacking developer intentionally routes sensitive data to a destination (on or off the device) that is not authorized by the user. We define sinks as Android API calls that may exfiltrate data beyond the application boundaries. Sinks include network, NFC, file system, email or SMS message, or directly to a colluding application via ICC or RPC. All of these sinks are guarded by the Android API. Sinks are identified as described in Section 4.4.

DroidSafe protects against explicit sensitive information exfiltration by tracking sensitive source to sink flows present in the application. DroidSafe analyzes an application before it is placed on an app store or before device install. Not every flow reported by DroidSafe is malicious; maliciousness depends on the intent of the developer and the security policies of the user or organization. Thus, the user or a trusted entity reviews the information flows for malicious leaks.

4.3.1 Limitations

We assume the device has not been rooted, and dynamic code loading is not present in the application. We do not aspire to detect leaks of sensitive data via side channels or implicit flows [26]. Our trusted computing base on a device is the Linux kernel and libraries, the Android framework, and the Dalvik VM.

DroidSafe's reporting is defined by the source and sink calls identified in the Android API. An attacker could exfiltrate API-injected information that is not considered sensitive by DroidSafe, or

via a call that is not considered a sink; and it would not be reported.

Our analysis does not have a fully sound handling of Java native methods, dynamic class loading, and reflection. However, we compensate for these idioms with aggressive best-effort policies and analyses. Our analysis has a blanket flow policy for native methods of an Android application, but an application could inject a sensitive flow in a native method, and DroidSafe would not report it. We attempt to aggressively resolve reflection targets in a fashion similar to [27] and [28], but if a *reflected invoke* cannot be resolved, we inject a special REFLECTION taint on the method's arguments and return value (injected by DroidSafe). Thus we could miss a flow injected in an unresolved reflected call.

Finally, we intend the DroidSafe ADI to accurately reflect the runtime semantics of Android with respect to the information flow and points-to information. While we believe we largely cover the semantics, given the size of the Android runtime and API we acknowledge that there may be some methods whose semantics the current ADI does not fully reflect. Different versions exist of Android, and we analyze an application in the context of Android 4.4.3.

4.4 DroidSafe's Android Device Implementation

To our knowledge, our model of Android represents the most complete, accurate, and precise Android execution model suitable for static analysis. We accurately and precisely model complexities such as callback context, life-cycle events, data flows, heap object instantiations, native methods, and aliasing. Our Android model is expressed in a single language, standard Java, matching the source language of Android applications, and is appropriate for many existing analysis techniques. One could think of our Android model as a software implementation of an Android device; along with the application and harness. Thus we call our model the *Android Device Implementation* (ADI).

The ADI is a simplified (thus easier to analyze) model of the actual Android system that, with respect to our analysis, represents a best-effort over approximation of the possible behaviors of the real system. The combination of an application, our ADI core, our harness, the semantic transformations for ICC (see Section 4.6), and resources (unique for each application), creates a closed application for analysis of an Android application.

4.4.1 ADI Core

We seeded our ADI with the Java implementation of the Android API available from the Android Open Source Project (AOSP) [21], version 4.0.3, along with additional open-source libraries upon which the AOSP implementation depends. This created a code base with no missing dependencies that could be compiled. This code base was approximately 1.3 MLOC, however it was missing substantial portions of the semantics of the Android API and runtime such as native methods, event firings, callback initiation, and component life-cycle events. Furthermore, many commonly used classes included Java implementations that present difficulties for a static analysis.

We therefore developed a novel technique, *accurate analysis stubs*, to enable the effective analysis of code whose full semantics lies outside the scope of AOSP. Each stub is written in Java and only incompletely models the runtime behavior of the modeled code. But the semantics of the stub is complete for the abstractions that the analysis deploys (in this case points-to and information-flow analyses).

We added accurate analysis stubs for 3,176 native methods to model the data flow, object instantiation and aliasing of the missing native code. This was accomplished through a combination of automated and manual means, though all methods were reviewed manually. We developed concrete implementations of 45 classes for which concrete implementations are left to closed-source, commercial libraries.

We automatically created classes for 421 classes defined via the Android Interface Definition Language (AIDL) used for RPC contracts.

We simplified the implementation of 117 classes in the Java standard library and Android library to increase precision and decrease analysis time. Examples include container classes, component classes, I/O classes, primitive wrapper classes, strings, and threading classes. We attempted to faithfully maintain the semantics of the original code with respect to its contract with an Android application and a flow-insensitive analysis. The base AOSP plus our additions and modifications enable our ADI to accurately and precisely track flows through the API.

4.4.2 Event and Callback Dispatch

We created a runtime implementation hooked into the API implementation that models component creation, shared and saved state, life-cycle event firing and argument context, and callback event firing and argument context.

For callback handlers, we implement the callback registration method to invoke the application's callback handler method with the appropriate arguments. For example, Android defines the ability for a component to register to be notified if a database has changed, and handle this change in a given method in a new thread. The application will define a callback handler object, and register this to be notified of database changes. The ADI implements this registration method via a stub that creates the thread directly, and calls the callback method on the registered database. Since our analysis is flow insensitive and our harness is wrapped in a loop (see below), DroidSafe considers all event orderings even though the stub API method invokes the callback handler method directly from the callback registration method (with the appropriate context).

For arguments to callback handlers that are generated by the runtime system, our model creates a new object and passes it to the registered callback handler in the app. For example, to model a key press, our runtime system will create a new object to represent the key press, and call the callback handler with this object on each component.

We developed a separate package for implementing component creation and life-cycle event modeling. This package contains stubs for registration methods for each Android component type: Activity, Service, BroadcastReceiver, and ContentProvider. The harness (discussed below)

instantiates each application component and passes the component object to the appropriate registration method. The registration methods model shared preferences, saved state, global context classes, and device configuration. This context is passed accurately to the life-cycle events of components.

Since our runtime system makes explicit calls to all life-cycle events of each component, a flow-insensitive analysis can capture the flow between the two life-cycle events in the component in Figure 4.2(a). Also, since we accurately model saved state through the API and back into a callback handler, our model enables an analysis to report the flow in Figure 4.2(b).

4.4.3 Identifying and Classifying Sources and Sinks

We manually identified 4,051 sensitive source methods and 2,116 sensitive sink methods in the Android API. We also classified each source and sink with a high-level classification (e.g., location, device ID, file, network, and database) so that analysis results can be grouped for verification or consumption by a human. For example, a flow reported by the tool might be: "Location data can flow to the network."

Initially, we tested SuSi, a tool that automatically identifies sink and source methods in the Android API [29]. The automatically identified sources and sinks were incomplete. We compared our identifications of sources and sinks with the results of SuSi[1] and found that SuSi is missing hundreds of important sources. For example, SuSi did not identify 53% of source calls as "sensitive sources" and 32% of sink calls as "sinks" for the malicious flows in the APAC malicious applications (see Section 4.9). These missing sources and sinks indicate the challenge of automatic identification.

4.4.4 ADI Coverage and Keeping Current with Android Updates

The core of our model includes 550 commonly-used Android API implementation classes. We manually reviewed, added accurate analysis stubs, and verified these 550 classes. For verification, we manually confirmed that the class implementation is not missing semantics for data flow, object instantiation, and aliasing; and that event callbacks defined in the classes are called explicitly by our model (with the proper context). For classes not in our core set, we still maintain high accuracy because we analyze the actual Java implementation (with accurate analysis stubs), however we may experience a higher level of imprecision for these classes if their implementation is complex.

To measure the coverage of the ADI, we acquired a list of Android API method call frequencies accumulated over 95,910 Android applications downloaded from the Google Play Store. This list reports the number of invoke expressions to each Android API method over all the applications. Calls to the core 550 verified classes account for 98.1% of the total calls over these applications.

We initially seeded the ADI with the AOSP version 4.0.3, and verified the core based on this version. We have since upgraded our model to Android 4.4.3. This process included reviewing

[1] We used the source and sink lists for Android 4.2 in the SuSi public repository under the directory SourceSinkLists/Android 4.2/SourcesSinks.

changes to classes in our core 550 classes between these versions; and accounting for and verifying any changes in the ADI. This process required one person-week of work, for an experienced Java and Android developer. For the update, the rest of the ADI classes were copied over from AOSP 4.4.3, and accurate analysis stubs were created for native methods. This process required another person-week. We expect the update process for our ADI to continue to be relatively fast since there are few changes to the core of Android between version updates; historically new implementation has been contained in new packages.

4.4.5 Harness

Each analyzed application must be *hooked* into the ADI via a harness. In its first pass, DroidSafe generates this harness method automatically. DroidSafe scans the application source code for all classes that subclass one of Android's four component types. It instantiates objects in the harness for all such classes found. We cannot rely solely on the Android manifest for the complete list of components, since the manifest is required to list only components that are exported and available to other applications. We represent each component with a single heap object to account for the complexities of the Android component memory model (see Section 4.2.1). In our harness, each instantiated component object is passed to the appropriate runtime method to exercise all of its life-cycle events. The harness method is wrapped in a loop; the loop is present to capture all possible orderings for callbacks that are called in the harness.

XML Layout Incorporation: Android enables a developer to specify screen layout via XML declaration files. DroidSafe parses these files, and inserts statements in the harness to build the declared GUI elements programmatically via the associated method calls in the Android API. DroidSafe also sets field values for each declaring GUI item based on the XML proprieties declared for the item, e.g., button names, completion hints, and text field values; 16 properties are supported.

In the application, XML declared GUI elements are instantiated via a method that requires the ID of the XML element. DroidSafe maps these IDs to the fields it has created in the harness. XML layout instantiation calls (9 varieties of findViewById(int) and setContentView(ID)) in the application are translated into an access of the mapped field for the constant ID argument. If the argument cannot be resolved to a constant, then a special object is created with a special taint to denote it is un-modeled, and replaces the instantiation call. An *onClick* handler declaration defined for a button is transformed into calls to the *onClick* method in our harness, called on all components that inflate the button.

We also support incorporation of String values defined in XML resources by translating methods that access the Strings into the constants defined in the resources.

Incorporating GUI and String values from the XML resources enables our analysis to resolve more strings constant values (important for ICC resolution (Section 4.6). Also, the DroidSafe tool provides this rich information in its results such that the context of sensitive flows can be quickly understood. For example, instead of only reporting flow endpoints, we can report that a flow from location to network is executed during the "Send" button press to the server "http://maps.google.com". The name of the button is specified in the XML layout file, and server names are often given in the string resources.

Our ADI is precise enough for analysis that resolves string values throughout the API, and accurate enough to track data flows on large Android applications through API calls (Section 4.7). Our ADI implementation is available upon request to researchers who would like to utilize it for analysis of Android applications

4.5 Object-Sensitive Points-To Analysis

Points-to analysis (PTA) is a foundational static program analysis that computes a static abstraction of all the heap locations that a pointer (reference) variable may point to during program execution. In addition to the points-to relation, points-to analysis also constructs a call graph as modern languages require points-to results to calculate targets for dynamic dispatch and functional lambda calculations. Our goal was to employ much of the AOSP Android API implementation without modification, and achieve precise results for our client analyses. However, as with many static analyses there is a trade-off between scalability and precision; appropriate control-flow and data-flow abstractions must be chosen to avoid intractability and to calculate acceptably precise results.

The DroidSafe system places heavy demands on a points-to analysis:

1. Java source code is notoriously difficult to analyze given heavy object and method reuse in varying contexts.
2. An Android application plus our ADI represent a large code base (+100K lines of reachable code).
3. A precise call graph must be calculated to preclude unreachable code from analysis (remember our entire ADI is 1.3 MLOC).
4. Our information flow analysis tracks thousands of sources and sinks in the API and has the ability to track user-defined flows, thus many queries of the PTA are expected.
5. Our ICC resolution analysis tracks values for all String constants resolved in application code.

Let us consider the difficulties of analyzing complex Java code with precision. Figure 4.3 lists simplified ADI source code for two commonly used classes in the Android API: android.os.Bundle and java.util.HashMap. Bundle allocates a HashMap. The example also provides relevant code for two Android activities that each create a Bundle and store values to their Bundle; Activity1 puts non-sensitive data in its Bundle while Activity2 puts sensitive data in its Bundle. Consider the difficulty presented to an analysis given this code. To precisely separate the two Bundle objects created (Ⓝ and Ⓢ), a PTA must separate multiple levels of allocations started at each Bundle allocation (Bundle allocates a HashMap, Ⓗ, which allocates an array to store entries, Ⓣ). In other words, an analysis must be able separate analysis facts between the array of entries created in the two HashMaps objects of the two Bundle objects in this code. Otherwise, sensitive data put in one Bundle is conflated with data that can be retrieved from the other Bundle, decreasing precision.

Our PTA algorithm is based on a whole-program, flow-insensitive, subset-based foundation [30] for Java on which we have added *context sensitivity*. Context sensitivity is a general approach where a PTA is able to separate analysis facts for a method *m* that arise at multiple call sites of

Android Device Implementation (ADI)

```
1  package android.os;
2  public class Bundle ... {
3    private Map<String,Object> mMap =
4                  new HashMap<String,Object>(); (H)
5
6    public void put(String k, Object v) {
7      mMap.put(k,v);
8    }
9
10   public Object get(String k) {
11     return mMap.get(k);
12   }
13 }
```

```
1  package java.util;
2  public class HashMap<K,V>... {
3    private Entry[] table = new Entry[size]; (T)
4
5    public void put(K key, V value) {
6      ...
7      table[index] = new Entry<K,V>(key, value); (E)
8    }
9
10   public V get(Object key) {
11     ...
12     e = table[indexFor(hash, table.length)];
13     ...
14     return e;
15   }
16 }
```

Android Application Source Code

```
1  public class Activity1 extends Activity {
2    ...
3    Bundle bundle1 = new Bundle();  (N)
4    bundle1.put("data", <notSensitive>);
5    ...
6    sink(bundle1);  //not a sensitive flow
7  }
```

```
1  public class Activity2 extends Activity {
2    ...
3    double sensitive = location.getLatitude(); //source
4    Bundle bundle2 = new Bundle();  (S)
5    bundle2.put("data", sensitive);
6    ...
7    sink(bundle2);  //flow of sensitive -> sink
8  }
```

Figure 3: Example source code for our ADI and two Activity objects illustrating the challenges of points-to and information flow analysis.

m. There are multiple choices for context, and the DroidSafe PTA implements *object sensitivity*. Accumulating evidence demonstrates that object sensitivity is the best choice for object-oriented languages [31, 32, 33, 23].

Object sensitivity is notoriously difficult to understand and implement [23]. Here we give the reader an intuitive description of object sensitivity and its scalability challenges. For a rigorous description of object sensitivity see [23] (note that our PTA implements a 3Full+2Heap analysis modified as described below). An object-sensitive analysis uses object allocation sites (new expressions in Java) as context elements. In our analysis, a heap object, *o*, is represented by the allocation site of *o*, plus the allocation site of the object that allocated *o*, and so on, to a parameterized depth, *k*. For a given method, our analysis is able to separate facts depending on the heap object of the receiver on which the method is called.

Considering again the example in Figure 4.3, our analysis is able to separate analysis facts calculated for the array table of two HashMap objects allocated from the two Bundle objects. When the Bundle objects are created on line 3 and line 4 of Activity1 and Activity2, respectively, a series of allocations is performed via constructors. The object-sensitive heap abstraction will have two separate elements representing the two table arrays, with context being their allocation history:

(Ⓣ ← Ⓠ ← Ⓝ)

(Ⓣ ← Ⓗ ← Ⓢ)

Where $a \leftarrow b$ denotes "*a* allocated in *b*".

The new expression on line 3 of Activity1 creates a heap object (Ⓢ), and the cascading allocations

from the constructors of Bundle and HashMap create (Ⓗ ← Ⓢ) and (Ⓣ ← Ⓗ ← Ⓢ) respectively.

When the Bundle.put(...) method of Activity2 is called (line 5), the method context (receiver) for the call is (Ⓢ), this triggers a call to HashMap.put(...) (line 7 of HashMap) with context (Ⓗ← Ⓢ). In this context, the points-to set result for the reference to the field table on line 7 of HashMap is the array object (Ⓣ ← Ⓗ ← Ⓢ). Any elements placed in this array via the assignment of line 7 will only be reflected in this array heap object.

Thus, for the example, our PTA is able to separate analysis facts between the two Bundle objects in the two Activities via deep object-sensitivity. For this example, the analysis requires a heap context depth of 3 (to distinguish (Ⓣ ← Ⓗ ← Ⓝ) from (Ⓣ ← Ⓗ ← Ⓢ)).

Object sensitivity has powerful precision but scalability presents a challenge. Our example requires a depth of 3, and other commonly used classes require deeper depths, e.g., in the AOSP implementation, Intent allocates a Bundle (that allocates a HashMap...) requiring a depth of 4 to disambiguate the items placed in the Bundle of two Intent objects. We tested other whole-program object-sensitive frameworks, but found they could not scale to the required context depth [34] or did not maintain program information required for our client [23].

To solve this scalability challenge, our points-to analysis implementation operates on the pointer assignment graph (PAG) representation of the program [35], an explicit representation of the program. In the past, while explicit implementations provided the fastest running times, they would typically exhaust main memory for large programs [34, 36]. However, today, with the large and increasing size of available main memory, we found that an explicit implementation can now scale to large programs.

Furthermore, our implementation is flexible and parameterized. This flexibility enables us to implement a series of client analysis-specific and Android-specific optimizations of our object-sensitive points to analysis. Without our optimizations 3 of the 24 APAC applications (see Section 4.9) could not finish analysis in DroidSafe given a limit of 64GB of heap memory. With optimizations, all applications now run in under 34GB of heap memory. The optimizations pro- vide a savings of 5.1x in total analysis time. We highlight the important optimizations here.

4.5.1 Selectively Applying Context

Typically, a points-to analysis keeps the same base context depth for all allocated objects. Initially, we tried this policy, at a depth of 3, for the applications in our APAC suite. However, our analysis would fail on 3 of our APAC applications because it exhausted 64GB of allocated heap memory. Instead we implement a targeted approach: we add context for an abstract heap location at the minimum depth that is required to achieve precision for one of our client analyses. The depth of context on abstract heap objects varies from 0 to 4. The context depth is calculated based on the following.

By analyzing a suite of 211 Android applications for which we had source, we learned which API classes from our ADI could be analyzed content-insensitively without significant loss of precision for clients of the PTA. We performed our points-to analysis on our suite, and determined API classes that could never reach (via a series of local or field references) a String value our Intent resolution analysis was tracking, and could never reach a value that was tainted with sensitive information flow (across all Android applications in our suite)

This set, S, contains 1489 Android API classes. Most of these classes are Android GUI objects and libraries through which sensitive data should never flow. For all c ∈ S, our client analyses will calculate the same results regardless of whether an object of c in the heap abstraction of our PTA has context or not (for the 211 Android applications analyzed). This set represents 26% of the total classes of our ADI. We extrapolate that context-insensitivity analyzing the classes of S in our will give us an acceptable loss of precision across all Android applications.

We modified our PTA to never attach context to an allocation of or a method call on an object of a class in S. This means that for a method m called on c ∈ S, the information-flow analysis analyzes m without context, conflating the analysis results of all calls to m on objects of c. For information-flow analysis, this relaxation means that, in m, if a sensitive taint flows to a field f of an object of a class c, the taint will (imprecisely) flow to f for all heap objects of class c.

Conversely with a maximum context depth of 3, our points-to analysis is unable to disambiguate many important analysis facts. For example, we could not disambiguate the Bundle between separate Intent objects (as discussed above). To address this issue, we automatically increase the context depth to 4 for all heap objects of Array type. In the example of Figure 4.3, this means that the array object allocated at line 3 of HashMap has a depth of 4, giving it enough context to disambiguate the Intent object that allocated the Bundle that allocated the HashMap that allocated the array. This general strategy works across containers in the Java and Android API packages.

4.5.2 Strings

At uniform context depth of 3, 63% of all heap objects are Strings in our PTA across our APAC applications. Most object sensitive PTA implementations represent *all* dynamic string objects by the same abstract heap location for scalability [23]. We cannot make such a choice because String precision is required for our Intent resolution analysis and our information flow analysis.

To achieve scalable analysis time, first we limit the context of Strings to depth 2. This sacrifices some precision, but the differences are negligible for the clients of our analysis. Second, we altered the implementation of the base java.lang.String class in our ADI to limit String object creation without sacrificing accuracy for our information flow.

The most effective modification to the ADI's implementation of java.lang.String concerns String operations, e.g., concat, substring, and replace. In the original AOSP implementation, each of these operations return a new string; causing a blowup in the number of objects in our abstract heap. We altered these operations to return a reference to the receiver string such that a new string is not created. In the body of each altered method, we are careful to respect the information flow effects of the operation. For example, the replace method transfers the flow from its arguments to the receiver object, and returns the receiver object. Thus fewer String objects are created, helping scalability. This technique can also improve the precision of our information flow analysis (see Section 4.7) by preventing the collapse of distinct string objects at the (now removed) allocation within the string operation. Scalability and Usability of Points-To Analysis

For reference, over the APAC applications, the maximum number of edges in the context sensitive callgraph produced and analyzed by our PTA has 14 million edges. The largest context sensitive

callgraph size we have found reported in the literature is 9.5 million edges [23].

Finally, unlike most other available PTA tools (such as Doop [31] and Paddle [32]), our analysis implementation does not calculate its result on a compressed representation of the program. We sacrifice memory to decrease analysis time, taking advantage of growing RAM size in desktops.

4.6 Improving the Precision of ICC Modeling

Inter-component communication (ICC) is common in Android applications and must be modeled both accurately and precisely. However, the AOSP implementation of ICC-related classes is incomplete (relying on native methods for target resolution and payload delivery). To achieve precision, we implement our own model of ICC via accurate analysis stubs, aggressively resolve dynamic program values, and transform application code to increase precision.

Intent objects describe ICC destinations. Values involved in the resolution include both java.lang.String and java.lang.Class objects. We resolve these values statically to guide precision-enhancing transformations. Our ADI provides us with an accurate and precise model for resolution analysis of values involved in Intent resolution.

Strings are an essential base value type of many of the Intent fields. To resolve string values given the myriad operations performed on strings, we employ the JSA String Analyzer (JSA) [37]. JSA is a flow-sensitive and context-insensitive static analysis that includes a model of common operations on Java's String type. For a given String reference, the analysis computes a multi-level automaton representing all possible string values. As a first pass, we run JSA (on only the application source) to resolve values for string references that are arguments to Android API calls. We convert each resolved automaton to a regular expression that represents the possible values of the string value.

After JSA is run, we replace resolved string values in the application code with constants representing their computed regular expression, and perform a pass of our points-to analysis such that these values can be propagated globally. We run our points-to analysis and store the results of this analysis for all string references in the program, such that later we can query the resolved regular expressions representing values for all string references in application code.

4.6.1 Resolving Explicit Intent Destinations

Explicit Intent objects are initiated with the destination component's fully-qualified class name or class constant object. Before the PTA is run, each class constant passed to a method of Intent is converted into a component name string constant representing the class. To determine the destinations of an Intent object in our abstract heap, we query the points-to information for the

Source Method	Target Method Call Injected
Context: void send*Broadcast(Intent,. . .) [*6 variants*]	BroadcastReceiver: void onReceive(Intent)
Activity: void startActivity*(Intent,. . .) [*6 variants*]	Activity: void setIntent(Intent)
Context: void bindService(Intent, Connection)	Service: void droidSafeOnBind(Intent, Connection)
Context: void startService(Intent)	Service: void onStartCommant(Intent, ...)
ContentResolver: insert, query, delete, update	ContentProvider: insert, query, delete, update

Figure 4: DroidSafe's ICC source to target methods transformations.

fields of component name. If all of the strings objects in the points-to set are constants, we consider the Intent object resolved.

4.6.2 Resolving Implicit Intent Destinations

Implicit Intent objects are Intent objects for which a component name is not specified; in our analysis these are Intent objects for which the component name field is null. Implicit Intent objects do not directly reference a destination but instead leave it to the Android system to deliver them to the appropriate destination(s). A component registers as a destination of implicit Intent objects by declaring IntentFilter elements in the manifest or programmatically installing IntentFilter objects on components. IntentFilter registrations specify string constants that the component will accept for the action, category, data type, and uniform resource identifier (Uri) fields of a dispatched Intent. We parse the Android manifest, and keep a map of implicit Intent registrations, i.e., for each component the implicit Intent field values it accepts. We also support updating this map with programmatic registrations for BroadcastReceiver objects, by modeling IntentFilter.

An implicit Intent object in our abstract heap is resolved if our PTA concludes that the points-to set for one of the action, category, data type, or Uri fields reference only constants (or is empty).

For a resolved implicit Intent, *i*, we build *i*'s list of in-app targets by comparing to each component's intent filter. For component *c*, we test the action, category, data type, and Uri fields in sequence. For each field, if *i*'s field is unresolved, then the test passes. If the field of *i* is resolved, then if any of its string constants are in the set of strings accepts by *c*'s intent field for the field, then it is a match. All fields of *i* have to pass the test against the respective fields of *c*'s intent filter for *i* to be able to target *c*.

There is additional complexity for programmatic intent filters, as DroidSafe may not be able to resolve all fields of an intent filter to constants. An intent filter field that cannot be resolved matches the respective field for all Intents.

4.6.3 Transforming ICC calls to Improve Precision

ICC initiation calls are methods that pass an Intent to Android's runtime system to perform inter-component communication or binding. Our strategy for improving precision for ICC is to trans-

form ICC initiation calls into appropriate method calls at the destination(s), thus linking the data flows between source and destination.

Figure 4.4 presents a list of the most common ICC initiation calls, and the linkage calls that are inserted by DroidSafe to improve precision and accuracy. For example, for an invoke of startActivity(Intent), we transform this call into calls of the destination activities' setIntent(Intent), linking the source and targets. Thus when a target Activity calls getIntent(), all Intent objects that could possibly be sent to the Activity are calculated by our PTA (we update the PTA result after all ICC transformations are completed).

For ICC initiation calls on resolved Intent objects, we link the ICC initiation call to only the destination components that are specified by the Intent. This is achieved by calling the appropriate linkage method on the heap object allocated in our harness for the destination component. For unresolved Intent objects, we insert linkage calls to all components of the appropriate type. For example, a startActivity(Intent) call with an unresolved Intent is delivered to all Activity components.

4.6.4 Android Services

Android Services require additional sophistication because in addition to Intent-mediating communication, messaging and RPCs can be performed. We illustrate our Service transformations via the examples in Figure 4.2(c). In this example, the Activity ICCSource binds and sends a messages to the Service ICCService. The important steps performed by DroidSafe to resolve this flow are as follows:

1. Our manifest parser maps the Intent action string "ICCServiceAction" to ICCService.
2. DroidSafe resolves the Intent object on line 15 as an Implicit Intent with action string "ICCServiceAction". Consulting the implicit IntentFilter registration map, we see the Intent's destination is ICCService.
3. The call to bindService(...) on line 16 is transformed to a linkage call to the harness object representing ICCService. This linkage call is a new method we define in our ADI for Service, droidSafeOnBind(Intent, ServiceConnection). The linkage method performs the following (some details omitted):
   (a) Invoke ICCService's (the receiver's) onBind(Intent) method to retrieve the Binder object. The ADI model for android.os.Messenger.getBinder() creates a Binder that references the Messenger object which created it.
   (b) Invoke onServiceConnected(ComponentName, Binder) on the passed ServiceConnection object, passing the android.os. Binder returned from ICCService's onBind() method.

With the linkage methods called, the Binder object used to create the Messenger in ICCSource line 7 is connected to the IncomingHandler of line 8 of ICCService. The method android.os. Messenger.send(Message) has a stub to call the handleMessage(Message) method of its Handler object. Thus, the call mService.send(msg) on line 23 of ICCSource will deliver the message to the handleMessage(Message) method of ICCService's Messenger object.

This is just one example of binding and message communication in Android that we support. The DroidSafe ICC model supports precision increasing transformations for common forms of ICC, and handles uncommon cases conservatively.

4.7 Information-Flow Analysis

Our information-flow analysis computes an over-approximation of all the memory states that occur during the execution of a program. The analysis is designed as a forward data-flow analysis. For each type of statement, we define a transfer function in terms of how it changes the state of memory.

We divide memory into four separate areas that store local variables, instance fields, static fields, and arrays, reflecting the semantics of the Java programming language:

$$
\begin{aligned}
\text{Memory} &= \text{Local} \times \text{Instance} \times \text{Static} \times \text{Array} \\
\text{Local} &= \text{Ctx} \times \text{Var} \rightarrow \text{InfoVal} \\
\text{Instance} &= \text{Loc} \times \text{Field} \rightarrow \text{InfoVal} \\
\text{Static} &= \text{Class} \times \text{Field} \rightarrow \text{InfoVal} \\
\text{Array} &= \text{Loc} \rightarrow \text{InfoVal} \\
\text{Ctx, Loc} &= \text{AllocSite}^{k}
\end{aligned}
$$

Each of the memory areas is modeled as a function whose codomain consists of a set of *information value*s. An information value is a tuple of the type of information and the source code location where the information was first injected. Our analysis can identify not only the kind of information being exfiltrated but the code location of the source.

In the local variable area, Local, each method's local variables are parameterized by their calling contexts (i.e. the heap location of a receiver object), so the precision of the analysis does not decrease when a method is called in various contexts. In other words, our information-flow analysis analyzes local variables in a flow-insensitive and object-sensitive fashion.

The instance field area, Instance, is a function that takes as its arguments an abstract heap location and an instance field. The return value is information values that flow into the instance field of objects at the heap location. Note that an abstract heap location consists of a series of allocation sites (see Section 4.5). This area corresponds to what is colloquially called "context-sensitive (or object-sensitive) heap" or "heap cloning" in the literature [23]. Each static field of each class has an entry in the static field area, Static. Unlike the memory area for instance fields, the static field area is not parameterized by heap locations because, in Java, all objects of each class share the static fields of the class.

The analysis collects all information values that are assigned to the elements of an array and stores the result at a single heap location of the array area, Array. That is, we analyze arrays in an array-index-insensitive fashion.

An information value is injected into an appropriate memory area when a source API method is invoked from application code. More specifically, for the statement `r = o.source(a)` where `r` is of primitive type, the analysis puts an information value into an entry in the local variable area that

corresponds to the r variable in the current calling context; the stored information value consists of the statement's location and the type of information associated with the source method. If r is a reference, the information value is stored in the special taint field of an instance that r refers to in the current calling context.

For each o. sink (a, ...) statement that invokes a sink method, DroidSafe reports the information values for *accessed memory addresses* in the sink. For each argument (and the receiver o), DroidSafe reports all the information values that are attached to memory addresses (and their taint field) read during the execution of the body of the sink method (among memory addresses reachable from the argument).

In Figure 4.3, the analysis injects an information value into the sensitive variable when Location.getLatitude (), a source API method, is invoked (line 3 of Activity2). The information value consists of the line number and the type of information (for this case, a user's location). For the invocation of Bundle.put() method (line 5 of Activity2), the transfer functions of the statements in the body of Bundle.put() convey the information value from the sensitive variable to the value field of an Entry object whose heap location is (Ⓔ ← Ⓗ ←Ⓢ). At the call to a sink API method (line 7 of Activity2), the analysis reports 1) a user's location information is reachable from the bundle2 argument, 2) the information was generated first at line 3, and 3) whether the body of the sink method actually uses the information by reading the Entry.value field. On the other hand, for the call to a sink API method in Activity1 (line 6 of Activity1), the analysis correctly reports that a user's location information is not reachable from the bundle1 argument. That is, even though the Entity objects in Activity1 and Activity2 are both created at the same program location Ⓔ, the analysis properly distinguishes information flows into them because each of them has its own heap location ((Ⓔ ← Ⓗ ← Ⓝ) and (Ⓔ ← Ⓗ ← Ⓢ), respectively).

4.8 Implicit Flows

The last section covers our information flow analysis with respect to explicit flows. This chapter focus on our implementation of tracking of *implicit* flows. An implicit information flow occurs when some sensitive data, though not directly leaked to output, can be inferred from the program's control flow. For example, in the following program fragment:

```
int secret;

void check ( int x) {
    if ( secret > x)
        print (‘‘ greater than ‘‘ + x);
    else
    print (‘‘ less than or equal to ‘‘ + x);
}
```

The security-sensitive integer secret is not leaked via an explicit flow. However, information about it is leaked, i.e., whether it is greater than a given integer x. When invoked repeatedly with different values of x, the function check can ultimately leak the value of secret.

We have enhanced the information flow analysis in DroidSafe to track implicit flows as well as explicit flows. The implicit flow tracking can be turned on or off via a configuration parameter. Below we will describe how DroidSafe incorporates implicit flow tracking into its information flow analysis.

4.8.1.1 Compute Control Flow Graphs

Implicit flows leaks information through the program control flow. Therefore our implicit flow analysis first computes control flow graphs for the reachable methods in the given program.

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line code sequence with no branching. Directed edges are used to represent branching in the control flow.

4.8.1.2 Augment Analysis State with Implicit Flow Data

We used additional state variables to store the implicit flow data collected during the analysis:

- blockIFlows = Block × Ctx → InfoVal

  Where Block represents a node in a program's control flow graph, i.e. a basic block in the program body

  For each block that is implicitly tainted in a given context, this state variable stores the corresponding sources of taint.

- locIFlows = Loc → InfoVal

  For each abstract heap location that is implicitly tainted, this state variable stores the corresponding sources of taint.

4.8.1.3 Compute Implicit Taints of Blocks on the Control Branches

The information flow analysis is implemented as a forward data-flow analysis. For each type of statement, we apply a transfer function that modifies the analysis state. The implicit flow analysis added a transfer function for control transfer statements including the Java If statements and Switch statements. If the conditional or switch expression is tainted, the transfer function marks the blocks on the control branches and method blocks reachable from these blocks as implicitly tainted. Here

are the steps:

```
1  curBlock ← the basic block containing the if/switch statement
   { Compute the set of blocks on the control branches 'siblings' as follows: }
2  subGraph ← directSubGraph(curBlock)
   { The directed subgraph of the control flow graph starting from 'curBlock', with an entry block (through
     which control enters into the subgraph) and an exit block (through which all control flow leaves) added: }
3  postDominatorTree ← the post dominator tree from the subGraph
   { In a control flow graph, a block M postdominates block N if every path from N to the EXIT block has to
     pass through block M. A block M immediately postdominates block N if M postdominates N, and there is
     no intervening block P such that M postdominates P and P postdominates N. }
   { The post dominator tree is a tree depicting the postdominator relationships of a given control flow graph.
     There is an arc from Block M to Block N if M is an immediate postdominator of N. This tree is rooted at
     the exit block. }
4  siblings ← the sibling (and grand-sibling) blocks of curBlock in the post dominator tree. These are the
     blocks on the branches that will receive implicit taints
5  foreach method context do
6  |  taints ← collectTaints(<if/switch expression>)
7  |  if taints /= empty then
8  |  |  foreach sibling ∈ siblings do
9  |  |  |  add taints to blockIFlows(sibling, context)
10 |  |  end
11 |  |  foreach method context reachable from the sibling blocks do
12 |  |  |  foreach block in the method do
13 |  |  |  |  add taints to blockIFlows(block, context)
14 |  |  |  end
15 |  |  end
16 |  end
17 end
```

4.8.1.4 Propagate Implicit Taints from Blocks to Abstract Heap Locations

We have modified the transfer function for assignment statements to propagate implicit taints from blocks to abstract heap locations as follows:

```
1  lhs = LHS of the assignment statement
2  if lhs is not of primitive type then
3  |  curBlock = the basic block containing the assignment statement
4  |  foreach method × context do
5  |  |  implicitTaints = blockIFlows(curBlock, context)
6  |  |  if implicitTaints /= empty then
7  |  |  |  foreach loc ∈ pointsToSet(lhs, context) do
8  |  |  |  |  add implicitTaints to state.locIFlows(loc)
9  |  |  |  end
10 |  |  end
11 |  end
12 end
```

4.8.1.5 Propagate Implicit Taints from Abstract Heap Locations to Method Blocks

We have modified the transfer function for statements containing method calls to propagate implicit taints from the receiver to the blocks in the called methods as follows:

```
1  receiver = receiver of the method call
2  foreach callerContext do
3  |  locs = pointsToSet(receiver, callerContext)
4  |  implictTaints = union of state.locIFlows(loc) for loc in locs
5  |  if implicitTaints is not empty then
6  |  |  foreach calleeContext do
7  |  |  |  foreach calleeBlock do
8  |  |  |  |  add implicitTaints to state.blockIFlow(calleeBlock, calleeContext)
9  |  |  |  end
10 |  |  end
11 |  end
12 end
```

4.8.1.6 Incorporate Implicit Taints into Explicit Taints via Assignment Statements

We have modified the transfer functions for assignment statements to add the implicit taints of the containing blocks to the explicit taints of the LHS as follows:

```
curBlock = the basic block containing the assignment statement
foreach method × context do
    implicitTaints = state.blockIFlows(curBlock, context)
    if implicitTaints is not empty then
        lhs = LHS of the assignment statement
        if lhs is a primitive variable then
            add implicitTaints to state.locals(context, lhs)
        end
        else if lhs is a primitive instance field then
            base = instanceFieldGetBase(lhs)
            field = instanceFieldGetField(lhs)
            foreach loc ∈ pointsToSet(base, context) do
                add implicitTaints to state.instances(loc, field)
            end
        end
        else if lhs is a primitive static field then
            field = staticFieldGetField(lhs)
            add implicitTaints to state.statics(field)
        end
        else if lhs is primitive array element then
            foreach loc ∈ pointsToSet(base, context) do
                base = arrayElementGetBase(lhs)
            end
            foreach loc ∈ pointsToSet(base, context) do
                add implicitTaints to state.arrays(loc)
            end
        end
    end
end
```

4.8.1.7 Incorporate Implicit Taints into Explicit Taints via Return Statements

We have modified the transfer functions for return statements to add the implicit taints of the containing blocks to the explicit taints of the LHS of their caller assignment statements as follows:

```
curBlock = the basic block containing the return statement
foreach calleeMethodContext do
    implicitTaints = state.blockIFlows(curBlock, calleeContext)
    if implicitTaints /= empty AND return type is primitive then
        foreach callEdge: incomingEdges(calleeMethodContext) do
            if callStatement is an assignment statement then
                lhs = LHS of callStatement
                add implicitTaints to state.locals(callerContext, lhs)
            end
        end
    end
end
```

4.8.1 Other Implicit Flows

- In Java, the X instanceof Y operator is a boolean operator that returns true if the object X is of class (or subclass of) Y. If implicit flows tracking is enabled in DroidSafe, we modified the transfer function for statement, local = X instanceof Y such that the taints of the added taint field for all locations reachable from X (for all contexts) are transferred to local:

```
foreach method × context do
    locs = pointsToSet(X, context)
    forall loc ∈ locs do
        taints = union of state.instances(loc, taintfield)
        add taints to state.locals(context, local)
    end
end
```

- For the statement, local = length_of(a), we transfer the taint on the taint field of arrays referenced by a to local:

```
foreach method × context do
    locs = pointsToSet(a)
    forall loc ∈ locs do
        taints = union of state.instances(loc, taintfield)
        add taints to state.locals(context, local)
    end
end
```

4.8.2 Reporting Implicit Flows

As described above, for many types of implicit flows, in the statement transfer function the flow is added to a set that includes both implicit and explicit flows. Thus our existing means of reporting flows will capture many types of implicit flows.

However, for sink calls that are predicated on tainted control flow, we have added a separate report on implicit information flows. For each sink method call that is in an implicitly tainted block, the implicit taints into this sink will be listed in this report.

4.8.3 Tests

DroidSafe repository includes 17 small system test apps to test implicit flow tracking and reporting. Twelve of the tests were written by our team, the other 5 were developed by the creators of FlowDroid [14].

Currently the DroidSafe analysis passes 15 of 17 tests, with 96% precision (1 false flow reported out of 25 flows reported). The analysis does not currently track implicit flows introduced by exceptional control flow, nor does it track implicit flows initiated by indexing into an array of references.

4.9 Evaluation

This section presents experimental results that characterize the effectiveness of DroidSafe's information-flow analysis. Our results indicate:

1. DroidSafe achieves both higher precision and accuracy than FlowDroid [14] + IccTA [20] a current state-of-the-art Android information-flow analysis. [14] and [20] demonstrate that FlowDroid + IccTA achieve both higher precision and accuracy than commercially available tools such as IBM's AppScan Source [38] (which was specifically designed to analyze Android apps) and HP's FortifySCA [39].
2. DroidSafe successfully reports all malicious leaks of sensitive information in a suite of malicious Android applications developed by independent, motivated, and sophisticated attackers from three hostile Red Team organizations.
3. DroidSafe successfully scales to analyze large Android applications analyzed in the context of our ADI.

4.9.1 Methodology

DroidSafe is developed on top of the Soot Java Analysis Framework [40]. We implemented our object-sensitive points-to analysis on top of Soot's SPARK PTA framework [35]. DroidSafe comprises approximately 70K lines of Java code. Figure 4.5 presents the architecture of DroidSafe

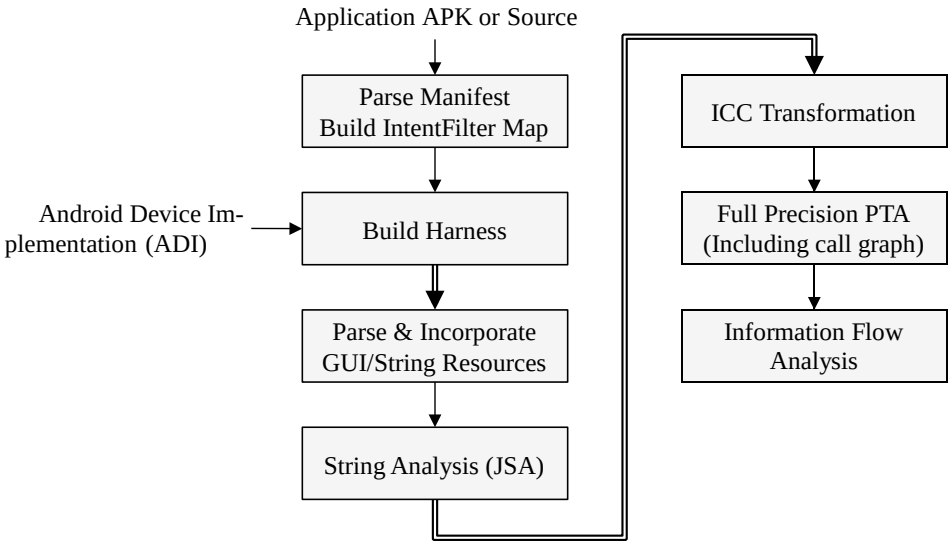

Figure 5: Phases of the DroidSafe Tool. Double lines denote an update of the PTA result is calculated for the next phase.

Table 1: DROIDBENCH results for DroidSafe and FlowDroid.

Tool	Missed Flows Explicit / Implicit	Accuracy	False Positives	Precision
DroidSafe	0/6	93.9%	13	87.6%
FlowDroid	12/7	80.6%	30	72.5%

(previous sections discuss the different phases of DroidSafe). DroidSafe can analyze an APK and Java source code.

We compare DroidSafe's information-flow results to FlowDroid + IccTA [14, 20], a flow-sensitive and object-sensitive static information-flow analysis system developed by academic researchers. This system incorporates Epicc [41] to resolve values for Intent objects used in ICC calls. For the remainder of this section we refer to this system as *FlowDroid* since FlowDroid is the underlying information-flow analysis. We run FlowDroid with our list of sources and sinks described in Section 4.4. Both tools report flows at a fine granularity as the generating source method call expression and the sink method call expression (Soot IR representation of the analyzed application).

We executed FlowDroid and DroidSafe on an Intel® Xeon® CPU E5-2690 v2 @ 3.00GHz running Ubuntu 12.04.5 with 64GB of heap memory for the JVM. We executed on a single core of the processor. We compare results for three application sets:

DROIDBENCH: DROIDBENCH[14] is an evolving set of Android micro-applications designed by the authors of FlowDroid to test precision and accuracy (recall) of static information-flow analyses. For our evaluation of DroidSafe, we employ DROIDBENCH version 1.2 plus the benchmarks developed for IccTA [20], which includes 94 benchmarks. One of the goals of DROIDBENCH is

Application	Lines of Code	Malicious Flow	
		Source	Sink
AgentSmith	1,481	Clipboard	Network
AndroidGame	63,755	Image Metadata	Network
AndroidMap	8,491	Location	Network
AndroidsFortune	14,621	Device ID	Network
AudioSidekick	2,444	Mic	Network
AWeather	1,837	Network	Network
BatteryIndicator	5,319	Image	Network
Butane	2,506	SMS	Network

Application	Lines of Code	Malicious Flow	
		Source	Sink
CalcF	861	User Input	Network
DeviceAdmin2	2,289	System Info	Network
FillInFun	82,602	Contact	SMS
KitteyKittey	962	Image Metadata	Network
PicViewer	221	Image Metadata	Network
Quickdroid	6,155	Contact, Bookmark	IPC
RunningApp	1,785	User Input	NFC
ShareLoc	372	Location	Network

Application	Lines of Code	Malicious Flow	
		Source	Sink
ShyGuyCRM	3,811	Contact	Email
SmartWebCam	1,176	Camera	AIDL
SMSBackup	387	SMS, Image, Browser	File
SMSBlocker	3,775	SMS	Network
SMSPopup	17,953	SMS	SMS
SnapshotShare	13,461	Screenshot	Network
SourceViewer	208	Device ID	Network
UltraCoolMap	2,658	Location	Network

Figure 6: APAC Information-Flow Applications: Size and malicious flows details.

to exercise potentially problematic Java idioms such as exceptions, callbacks, reflection[3], Android Inter-Component Communication (ICC), static initializers, and arrays. It is also designed to test the flow-, field-, and object sensitivity of the analysis. The largest DROIDBENCH application is under 200 lines of code. Reachable code including our analyzed Android ADI is always at least 80,000 lines of code. The maximum FlowDroid analysis time across the benchmark suite is 11 seconds. Despite the fact that it analyzes substantially more code than FlowDroid, the maximum DroidSafe analysis time is 222 seconds.

Additional Android Micro-Applications: To get better test coverage of Android, we developed our own set of 40 information-flow Android micro-applications. These applications test common Java and Android idioms for which coverage is missing in the current version of DROIDBENCH. The largest app is 255 lines of code. Benchmarks include tests of: Parcel, Activity saved state, Fragment, asynchronous event orderings, global Application object, String to char conversion, callback context, SharedPreferences, and dynamic dispatch precision. Fourteen (14) of the benchmarks test Android ICC mechanism coverage such as: components not in manifest, event ordering between components, programmatic IntentFilter registration, Intent passed through API objects, and various mechanisms for creating explicit Intents. We are working with the developers of DROIDBENCH to incorporate our benchmarks into the DROIDBENCH suite.

APAC: The APAC applications are complete real-world Android applications that, as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program, have been augmented with information-flow leaks by three independent hostile Red Team organizations. The goal of these organizations was to develop information leaks that would either evade detection by static analysis tools or overwhelm static analysis tools into producing unacceptable results (by, for example, manipulating the tool into reporting an overwhelming number of false positive flows). The APAC applications are unique because they are aggressive malware for which the ground truth for malicious information flows is defined. Furthermore, unlike much of the current generation of malware in the wild, which is delivered in over-privileged applications [42], many of the APAC applications contain malicious leaks to unauthorized destinations of sensitive data that the application is authorized to access based on its stated functionality.

The APAC applications comprise 24 Android applications. The application sizes range from 200 to 80,000 lines of Java code (not including library or Android run-time code). Reachable code including our analyzed Android ADI ranges from at least 80,000 to 180,000 lines. DroidSafe

[3]DroidSafe handles reflection using its String and points-to analysis to replace reflective calls in application code with direct calls to the target method, when applicable. The mechanisms of this transformation are similar to ICC transformation discussed in Section 4.6.

analysis times for the APAC applications range from 4.4 to 27.5 minutes.

4.9.2 DroidBench and Additional Micro-Applications

Figure 4.1 presents results from the DroidSafe and FlowDroid analysis of the DROIDBENCH suite. The second column contains entries of the form X/Y, where X is the number of missed explicit flows across all 94 applications and Y is the number of missed implicit flows. Implicit flows have no direct flow of data from the source to the sink; the flow is instead created via control flow that depends on sensitive data. DroidSafe detects all of the 90 explicit flows in the benchmark suite. FlowDroid detects 78 of 90 explicit flows. Inaccuracies in the FlowDroid tool cause it to miss the 10 flows: static initializers not modeled properly with respect to flow sensitivity, String to character conversions with unlinked flows, life-cycle event ordering inaccuracies, inaccurate fragments modeling, missing ContentProvider flows, and unresolved flows linked through reflected calls.

The fourth column of Figure 4.1 presents the number of false positives — i.e., the number of reported flows that do not actually exist in the 94 benchmark programs. Despite its greater accuracy, DroidSafe exhibits a lower false positive rate than FlowDroid. The final column presents the corresponding precision numbers for these false positives.

DroidSafe reports 13 false positives. DroidSafe reports two false flows due to conflating all elements of an array with tainted and untainted elements and two false flows due to accesses of a container (Map and List) with tainted and untainted elements. DroidSafe also reports four false flows due to flow insensitivity. We report two false positives because of our conservative (though accurate) model of life-cycle event orderings; in one benchmark the flow was connected by an order that could never occur at runtime (onCreate() called after onDestroy()). Two false flows are caused by conflating accesses of the Intent state map based on a string key.

Eighteen (18) of the 30 FlowDroid false positives are due to a conservative "Callback" source type that FlowDroid always injects for callback handler arguments because it does not model callback context accurately. Conversely, DroidSafe accurately and precisely models callback context in the ADI, and does not have to conservatively inject these flows. The remaining 12 false positives in FlowDroid are due to various causes such as conflating array and container elements, imprecisions in analyzing callback de-registration, and flow insensitivity on field assignments.

We now present results for the forty additional Android micro-applications developed by the authors. There are 42 leaks of sensitive information in the 40 applications. DroidSafe achieves 100% precision and accuracy for the suite. DroidSafe's comprehensive and precise Android model, the ADI, enables DroidSafe to capture all flows. FlowDroid (plus IccTA) achieves 34.88% accuracy and 79.0% precision. The low accuracy for FlowDroid is caused by the incomplete model of the Android environment that is included in the system. Examples of common Android idioms that FlowDroid currently does not model accurately nor conservatively include: API calls that access state (such as SharedPreferences and Application); taint transferred from an array index; String to char conversion; accounting for all legal asynchronous event orderings; callback context modeling (flows through API callbacks); accounting for components not defined in manifest; event ordering between components; Service binding and messages; and programmatic IntentFilter registrations.

Application	Mali-cious	DroidSafe						FlowDroid		
		Reachable Lines (including ADI)	Analysis Time (sec)	Reachable Source Calls	Reachable Sink Calls	Total Flows	Missed Malicious Flows	Analysis Time (sec)	Total Flows	Missed Malicious Flows
AgentSmith	1	123,881	434	53	60	167	0	60	123	1
AndroidGame	1	82,170	499	11	18	37	0	Did not complete		1
AndroidMap	2	102,236	698	78	41	132	0	54	25	2
AndroidsFortune	1	130,003	752	72	183	304	0	159	208	0
AudioSidekick	2	126,223	507	62	50	89	0	41	28	2
AWeather	1	126,218	491	35	30	72	0	116	57	1
BatteryIndicator	1	122,132	846	64	135	113	0	106	176	1
Butane	4	173,934	625	73	102	392	0	68	109	2
CalcF	2	117,414	374	11	21	11	0	33	5	0
DeviceAdmin2	2	137,046	358	17	33	5	0	47	6	2
FillInFun	2	123,016	601	22	64	14	0	75	25	1
KitteyKittey	1	110,584	271	4	1	2	0	47	1	1
PicViewer	3	118,019	360	7	3	8	0	20	0	3
Quickdroid	19	119,427	399	103	65	278	0	64	231	19
RunningApp	1	126,629	579	51	34	59	0	75	94	1
ShareLoc	4	119,771	1,051	6	4	7	0	28	7	4
ShyGuyCRM	1	177,853	1,255	105	99	463	0	78	82	1
SmartWebCam	1	126,029	1,649	101	267	21	0	50	30	1
SMSBackup	10	108,317	269	25	7	26	0	20	0	10
SMSBlocker	1	125,531	419	12	105	23	0	42	12	1
SMSPopup	3	149,824	1,477	180	182	918	0	298	304	3
SnapshotShare	1	130,111	590	89	29	108	0	92	71	1
SourceViewer	1	118,943	384	13	11	8	0	23	4	1
UltraCoolMap	4	121,507	407	14	9	12	0	34	42	4
Total	**69**					**3,269**	**0**		**1,640**	**63**

Figure 7: APAC Information-Flow Applications: DroidSafe and FlowDroid evaluation results.

4.9.3 APAC Malicious Applications

We next present analysis results for 24 real-world applications with malicious flows. Figure 4.6 presents details on the source lines of code and the malicious flows. The most common type of sink is the network (when the flow exfiltrates sensitive information via the network). The developers of the APAC applications attempted to intentionally hide malicious flows. Some of the patterns they used include flows through raised exceptions, application native methods, reflection, and character and string manipulation. They also employ Android API methods that are difficult to model precisely and accurately such as Object.clone, System.arraycopy, and primitive to string (and vice versa) conversion. One application (SmartWebCam) leaks sensitive information (camera) via an AIDL-defined RPC call that could be called by a colluding application. Android-specific implementation decisions that complicate information-flow analysis include:

1. **ICC**: In 17 of the applications, malicious flows cross components via Intents, messages, or RPCs.
2. **Callbacks**: In 17 of the applications, either a) sources or sink calls are reachable via control flow initiated via a callback, or b) malicious flows are linked through callback arguments passed via the runtime.
3. **API Inter-method flows**: In 6 of the applications malicious flows are linked through API method combinations with flows that are challenging to model. More specifically, method a is called to assign a reachable field or static memory location to a tainted value, then method *b* is called that accesses this memory via static memory or a path of fields from its arguments. SharedPreferences are an example: one component could write taint to a preference, and an-

other component could read from that preference.

4. **Non-argument sink flows**: In 7 of the applications, a malicious flow is not linked via the arguments of the sink call. Instead the source taint is accessed beginning with a) memory reachable from a field of the receiver, or b) memory reachable via a static access.

Figure 4.7 presents information-flow results. The column "Malicious Flows" gives the total number of malicious flows in the application. Many apps leak multiple sources of sensitive data; for example, UltraCoolMap leaks location information from four sources to the network.

DroidSafe achieves an accuracy of 100%, reporting all 69 of the malicious flows in the APAC apps. The high accuracy of our Android model enabled this result; the malicious flows in these applications often attempt to exploit Android behavior that is difficult to model. For DroidSafe, the ratio of reported malicious flows to total flows is 2.1%.

FlowDroid, in contrast, misses 63 of the 69 malicious flows, for an accuracy of 8.7%. Reasons for the low accuracy include missing ICC modeling, not considering all valid event orderings (exacerbated by FlowDroid's flow-sensitive analysis), inaccurate modeling of many Android API calls (causing disconnection of flows), and inaccurate callback context modeling. FlowDroid's ratio of found malicious flows to total flows is 0.3%. FlowDroid timed out after 2 hours for the AndroidGame application.

The "Total Flows" column in Figure 4.7 reports the total number of flows from sources to sinks that the tools report. Even though these flows involve sensitive information, the majority are not malicious as the flow is part of the intended functionality of the application. For example, some legitimate flows in UltraCoolMap send location information via the network to a trusted server. The malicious flows, in contrast, send the location via the network to an untrusted destination.

The APAC applications, as with all unknown applications, do not come with specifications that would enable DroidSafe or FlowDroid to distinguish malicious from legitimate flows. Instead, someone must understand the intended functionality of the application and make a subjective determination of which flows are malicious. Because all of the malicious flows that we report were inserted by the hostile Red Team, there is no doubt that at least these flows are malicious. We did not attempt to analyze all of the remaining reported flows to determine if the flows actually exist in the application or not.

4.9.4 Intent Resolution in APAC Applications

Section 4.6 presents our ICC modeling and defines *resolved* Intent objects. A resolved Intent is an Intent for which an analysis concludes that at least one of its field values is only a set of constants; resolved values can be used to reduce the number of true target components. Over all of the APAC applications, DroidSafe calculates that there are 213 Intent-based communication calls. DroidSafe resolves Intent objects in 95.8% of the calls. Of the calls with resolved Intents, 59.2% of calls employ explicit Intent objects, and 40.8% calls employ implicit Intent objects. Of the resolved Intent-based calls, 74.0% target at least one component of the app; DroidSafe concludes that the other 26.0% cannot target a component of the app. On average, across the

APAC apps, calls with resolved Intent objects that target an in-application component resolve to 1.03 destination components.

FlowDroid relies on Epicc to resolve values for Intent. We inspected the output of Epicc to determine the percentage of Intent objects that are resolved based on our definition. We ignore the Uri data field since Epicc does not reason about Uri objects. Epicc finds 177 Intent-based ICC calls in the APAC applications, Epicc resolves the Intent object arguments for 85.9% of the calls (versus DroidSafe's 95.8%). The lower number of total calls versus DroidSafe may be because of Epicc's model of Android callbacks and reachable code. Causes for unresolved Intent objects include lack of support for some explicit Intent construction mechanisms and Intent objects passed through API methods.

Across the APAC applications, there are 131 ContentProvider operations. Of the operations, 66.4% use Uri objects that DroidSafe resolves. Of the resolved operations, 35.6% target a components of the application, and each resolved operation targets 1.0 components. Epicc does not resolve Uri values, and consequently FlowDroid does not link flows through ContentProvider operations.

4.9.5 Implicit Flows

Tracking implicit flows will increase the number of sensitive flows reported by DroidSafe. We employed DroidSafe to analyze the 24 APAC applications with malicious leaks of sensitive information. Enabling implicit flow tracking increased the number of sensitive flows reported by 2.2x and the information flow analysis time by2.5x.

4.10 Related Work

Object-Sensitive Points-To Analysis: For robustness and flexibility, typical whole-program object-sensitive analysis implementations reduce program facts into representations appropriate for general solvers; examples include logic relations [23], constraints [36], and binary decision diagrams [34, 43]. Our implementation differs from these systems in that it operates directly on the pointer assignment graph (PAG) representation of the program [35], an explicit representation of the program. Previous work has demonstrated that direct implementations of points-to analysis problems, when they fit in memory, are typically faster than general solvers [34, 36]. Today main memory sizes are large enough to accommodate our direct implementation of a context sensitive analysis of large programs.

Tuning context-sensitivity of an analysis for precision and scalability has also received much work. *Hybrid* context sensitivity treats virtual and static method calls differently, and in addition to object sensitivity, attempts to emulate call-site sensitivity for static calls [44]. Our analysis implements hybrid context sensitivity by cloning static method calls for calls to application methods, and certain API factory methods. *Type sensitivity* is a form of object sensitivity that merges contexts based on types [23]. We tried type sensitivity for our client, but it did not provide adequate precision. An *introspective analysis* drops context sensitivity from program elements that could blow-up the

analysis [24], without regard for precision of the client. In client-driven approaches [45], a client analysis asks for more precision from the points-to analysis when needed. In contrast, our technique pre-calculates the set of classes (and thus allocations and methods calls) for which precision is historically not helpful for our problem.

Information-Flow Security Analysis: DroidSafe follows a long history of information-flow analysis (sometimes called taint analysis) systems for security. Livshits and Lam [46] present an approach for taint analysis of Java EE applications that is demand-driven, uses call-site context sensitivity, and shallow object sensitivity via inlining. TAJ [47] focuses on Java web application and employs a program slicing technique combined with a selective object-sensitive analysis. F4F [22] is a taint analysis for Java applications built on web frameworks that uses a specification language to describe the semantics of the underlying framework.

Focusing on information-flow analysis for Android, FlowDroid [14] is a sophisticated, open-source static information flow analysis for Android applications. FlowDroid's analysis is flow-sensitive, and thus, is more precise than DroidSafe, however the FlowDroid model of Android is not nearly as complete as DroidSafe's. FlowDroid attempts to compensate with inaccurate blanket flow policies on unmodeled API methods. From testing, we discovered that FlowDroid does not accurately model all possible combinations of life-cycle or callback events, demonstrating the difficulty of modeling Android execution in a flow-sensitive system. FlowDroid's analysis is on-demand and flow-sensitive as opposed to DroidSafe. However, each instantiation of the analysis is expensive; in preliminary experiments running FlowDroid with our ADI, the analysis completed only 7 of 24 applications given a 2 hour timeout for each application.

Epicc [41] is a tool that resolves Intent destinations in an application. Epicc developed a model of commonly-used classes and methods involved in the Android Intent implementation. Their analysis is on-demand and flow-sensitive. The DroidSafe system includes a more comprehensive model of classes and mechanisms used in inter-component and inter-application communication (for example Uri and Service messages). DroidSafe's resolution can also reason about values created in and passed through API methods.

IccTA [20] combines FlowDroid with Epicc and seeks to identify sensitive inter-component and inter-application information flows. DidFail [48] also combines FlowDroid and Epicc to discover sensitive flows across applications. Though not discussed here, DroidSafe includes an analysis to capture inter-application flows via a database of previously resolved Intent values and reachable source flows. This database is consulted and appropriate flows are injected before information analysis.

There are other many other examples of static information flow analyses for Android. CHEX [13] detects information flow vulnerabilities between components. ScanDal [15] is a static analysis implemented as an abstract interpretation of Dalvik bytecode. CHEX and ScanDal employ analysis with $k = 1$ call-site context sensitivity. SCanDroid [17] resolves data flows between components using a limited model of Android, and conservative flow policies for API methods. Leak- Miner [18] tracks flows with a context-insensitive analysis. AndroidLeaks [19] combines both context-sensitive and context insensitive analyses, but models flows through API methods with a blanket policy that reduces precision. DroidSafe includes a more precise analysis and has a more accurate and precise model of the Android API than these other tools.

Dynamic testing and monitoring approaches engender different tradeoffs compared to static analysis. Examples include the sophisticated dynamic taint-tracking tool TaintDroid [11], and Tripp and Rubin [49] who describe an approach for classifying information leakages by considering values that flow through sources and sinks. They do not have issues with reflection and dynamic class loading. But, if employed for triage, they require adequate test coverage. If used for dynamic monitoring they are susceptible to denial-of-service attacks if malware is activated during execution and the application is killed or functionality is disabled. This might be unacceptable for mission-critical applications. Similar to static analysis, they require user-mediated judgment for reported sensitive flows.

DroidSafe's list of sources and sinks was compiled manually. SuSi [29] employs supervised machine learning to automatically designate source and sink methods in the Android API. Merlin [50] is a probabilistic approach that employs a potentially incomplete list of sources, sinks, and sanitizers to calculate a more comprehensive list. Merlin automatically infers an information flow specification for an application from its *propagation graph* using probabilistic inference rules. While SuSi's list proved incomplete for the APAC applications, Merlin's technique is complementary to ours and a possible next step for helping the results of DroidSafe.

4.11 Conclusion

Malicious leaks of sensitive information pose a significant threat to the security of Android applications. Static analysis techniques offer one way to detect and eliminate such flows. The complexity of modern application frameworks, however, can pose a major challenge to the ability of static analyses to deliver acceptably accurate and precise analysis results.

Our experience developing DroidSafe shows that 1) there is no substitute for an accurate and precise model of the application environment, and 2) using the model to drive the design decisions behind the analysis and supporting techniques (such as accurate analysis stubs) is one effective but (inevitably) labor-intensive way to obtain an acceptably precise and accurate analysis. As long as there are complex application frameworks, we anticipate that making an appropriate set of design decisions (such as the use of a scalable flow insensitive analysis) to successfully navigate the trade-off space that the application framework implicitly presents will be a necessary prerequisite for obtaining acceptable accuracy and precision.

Our results indicate that the final DroidSafe system, with its combination of a comprehensive model of the Android runtime and an effective set of analyses and techniques tailored for that model, takes a significant step towards the final goal of an information flow analysis that can eliminate malicious information leaks in Android applications.

5.0 COOKBOOK FOR INFORMATION FLOW POLICIES

5.1 Overview

The DroidSafe Android application analysis system performs a static, global information flow analysis that tracks possible sensitive sources and sinks. This chapter details how an organization can create an exhaustive list of unauthorized or sensitive information flows in an Android application. The chapter is organized like a "cookbook" recipe with steps detailing how to create the information flow policy.

Information flow sources and sinks are defined with respect to the Java Android API. Each source API call is labeled with one of 42 source categories. Each sink API call is labeled with one of 31 sink categories. The author of the information flow policy classifies each source-to-sink flow as either unauthorized, non-sensitive, or sensitive. *Unauthorized* flows are never allowed in an application. *Non-sensitive* flows are always allowed in an application. *Sensitive* flows are sometimes allowed in an application and the context of the sensitive flow defines whether it is authorized or unauthorized.

An information flow policy can be utilized by DroidSafe to report flows that are sensitive and unauthorized for the organization, ignoring flows that are non-sensitive. A trusted analyst representing the organization can utilize the information flow policy while vetting an application to investigate each flow reported by DroidSafe. If an application does not include the possibility of any unauthorized or sensitive flows, then it is trivially authorized. If DroidSafe reports that the application could possible trigger an unauthorized flow, then it the analyst can either reject the application or investigate the implementation in more detail. Sensitive flows require the analyst to understand the context of the information flow to make the decision as to authorized.

This chapter provides the reader with an understanding of how to classify information flows from the API with respect to the security needs of a particular organization.

5.2 DroidSafe Information Flow Analysis Overview

The DroidSafe [1] system (covered in Chapter 4) is a state-of-the-art static analysis system for Android applications. DroidSafe presents a report of possible security sensitive-information flows and actions in an application under test (AUT) for review by a trusted analyst. DroidSafe does not attempt to classify behaviors as intentionally malicious; DroidSafe presents possible sensitive behaviors for review. The trusted analyst must consider the reported actions and flows of the

application with respect to the security and privacy policies of the organization she represents. Based on the policies of the organization, the application's possible behaviors may be authorized or unauthorized. If an unauthorized behavior is found, the application can be rejected, and either sent back to the developer for re-implementation, and / or investigation into the intent and nature of the implementation can be initiated if malice is expected.

The most prominent analysis in DroidSafe is information flow analysis. Information flow analysis tracks how sensitive information sources (such as the location sensors) flow through a program. For each value in a program (primitive locals and fields) the information flow analysis report if the value could be influenced by a particular sensitive information flow source. Having this result, it is then possible, for an application call to a sink API method (such as a network write), to know if a particular sensitive source could flow to a sink (such as, *Does location flow to the network?*).

The analysis can conclude two types of facts: 1) A source could possibly flow to a value (or sink), and 2) A source will not flow to a value. The analysis **cannot** answer if a source will always flow to a value (or sink).

Furthermore, DroidSafe's information flow analysis reports on the *context* of a source-to-sink flow. Context provides the analyst with more information for how the source-to-sink flow is used in the application, under what conditions and with what parameters. Context for a source-to-sink flow include:

- The user or system event that triggered the sink call. For example, was the sink call initiated by a user button press.
- Possible values for primitive or string arguments passed to the source call. For example, if the source is a file read, the path of the file read.
- Possible values for primitive or string arguments passed to the sink call. For example, if the sink is a network write, the IP address of the socket connection.

Context can be employed to further understand flows and to restrict or authorize sensitive flows. For example, it may be authorized to send location to the network address www.google.com, but not any other address.

5.3 Defining an Information Flow Policy

Now that we have background on information flow analysis, it is time to discuss how to define an information flow policy for sensitive flows of the Android API.

The Android API guards access to sensitive information, actions, and sinks. Sensitive information sources include location information, sensor data, device identification, etc. Sinks include any call that can export data beyond the boundaries of the application, e.g., inter-process communication, network write, file write, and other devices like audio or video output.

The task of the author of the information flow policy is to decide what information sources are sensitive and what sinks are sensitive. Each source and sink should be categorized as one of the following:

Table 2: Reviewing strategy for combinations of source–to-sink classifications

		Sink		
		Non-sensitive	**Sensitive**	**Unauthorized**
Source	**Non-sensitive**	Non-sensitive	?	Unauthorized
	Sensitive	?	Sensitive	Unauthorized
	Unauthorized	Unauthorized	Unauthorized	Unauthorized

1. **Non-sensitive**: The source or sink is not sensitive with respect to the security policy of the organization.
2. **Sensitive**: The source or sink is sensitive with respect to the security policy of the organization and all possible uses of the source and sink need to be manually inspected and authorized by an analyst. This inspection could include source inspection and / or inspection of the context of the use of the source or sink.
3. **Unauthorized**: The source or sink is never authorized for use. Any possible use of the source or sink is an automatic rejection of the AUT.

DroidSafe has defined 4,051 source API calls and 2,116 sink API calls in the Android API version 19. Obviously, it would not be feasible for the policy author to classify each of these calls. Thus, the DroidSafe system labels each call with a single high-level category. There are 42 source categories and 31 sink categories. The source categories are listed (with descriptions) in Table 5.2. The sink categories and descriptions are listed (with descriptions) in Table 5.3.

Again, it is the task of the information flow policy author to understand each of the high-level categories, and classify each as either non-sensitive, sensitive, or unauthorized.

Next, it is the task of the information flow policy author to decide on the strategy for reviewing combinations of source-to-sink classifications that are ambiguous in terms of need for review. Table 5.1 provides a matrix of the source-to-sink combinations. The information policy author must decide on the policy for the combinations marked with "?": 1) sensitive source to non-sensitive sink, and 2) non-sensitive source to sensitive sink. The possible decisions are: non- sensitive (meaning ignore and DroidSafe should not report) and sensitive (meaning DroidSafe should report and analyst review is required).

For an example of a sensitive source to a non-sensitive sink, consider location information written to the log. If the information flow policy author has designated location as sensitive but the device log as insensitive, it still might be the case that this flow, should it appear, should be reviewed. By designating case 1) above as sensitive, DroidSafe will offer all sensitive source to non-sensitive sink flows for review.

The next two sections provide the complete list of source and sink categories.

5.4 Source Category Descriptions

Table 3: Source categories and their descriptions

Source Category	Data Derived From
GUI	A user-interface (UI) component that is not text (for example, the size of a button or the color of a text-box).
ACCOUNT_INFORMATION	The user account on the device such as user name.
BLUETOOTH	A Bluetooth connection's data received on the device.
BLUETOOTH_INFORMATION	The settings or control of a Bluetooth connect (but not the data received over the connection).
BROWSER_INFORMATION	Web browser data accessed through the global browser databases or inter-application communication (for example, history).
CALENDAR	Default calendar application data on the device.
CLIPBOARD	Global device clipboard data.
CONTENT	Information describing the context of an application's Android runtime environment (for example, application name and permissions).
CONTACT_INFORMATION	Default contact application data (for example, phone number of a contact on the phone).
DATABASE_INFORMATION	Settings or schema information for a database (for example, column names).
DATABASE	Information stored in an application's database.
EMAIL	Default email application data (for example, sender of most recent email message received).
EXIF_INFO	Information embedded in an image on the device.
FILE	Information read from a file on the device.
FILE_INFORMATION	Settings and control data for file system, including file names (but not include data read from a file).
IMAGE	Image data (either captured from the camera or on the file system).
LOCATION	Location information (could be current location, historical location, or a calculated location).
MEDIA	Information read from a default media location on the device (could include files at default media locations like music).
NETWORK	Values read from a network connection (could be local or remote).
NFC	Values read from the Near Field Communication device.
PREFERENCES	Values stored in the local application preferences.
SHARED_PREFERENCES	Values read from a global preferences file.
RESOURCE	Values read from an XML resource included with the application, e.g., strings or user-interface declarations.
SCREEN	Screen capture data.
SMS_MMS	Values received in an SMS or MMS message.

Source Category	Data Derived From
SECURITY_INFO	Values related to default security settings for the application or device, for example encryption or key schemes employed or available.
SYNCHRONIZATION_DATA	Values derived from automated synchronization frameworks such as Google Sync.
SYSTEM_SETTINGS	Values that define various system settings such as default is roaming enabled. These are setting that are found in the "Settings" app on the device.
SYSTEM_PROPERTY	Values read from system environment variables.
UNIQUE_IDENTIFIER	Device unique identifiers such as IMEI.
SENSOR	Values read from a sensor on the phone that does not have its own category (such as accelerometer).
IPC	Values read from an external application.
IO	General category for values read from a Linux device, includes files and network streams.
AUDIO	Values derived from an audio clip, this could include audio recorded from the microphone.
USER_INPUT	User input into a text-based UI component.
OS_PROCESS	Values read from the output of a running process on the phone. This could include a process started by the AUT.
ANY_MEMORY	Values read from any memory location accessible from the Java app. These are values that are read by the sun. misc. unsafe package calls. These calls allow for reads of any memory address (unsafe) from Java.
CAMERA	Information captured directly from the camera through camera callbacks.
DATE_TIME	Information regarding data and time. Could include the current data and time.
AD	Values received from the Google advertisement framework.
GOOGLE_SERVICES	Values read from various Google services such as Google Drive.
WEB	Values read from a WebView embedded in the application including JavaScript console and HTTP control messages.

5.5 Sink Category Descriptions

Table 4: Sink categories and their descriptions

Sink Category	Data Flows To
ACCOUNT_SETTINGS	Writes to the centralized registry of user accounts on the device, for example, Google and Microsoft Exchange.

Sink Category	Data Flows To
AD	External library calls that send data to remote advertisement servers.
ANY_MEMORY	Writes in sun. misc. Unsafe package that allow writes to any memory address accessible by application.
AUDIO	Writes to setting that could affect audio controls like volume, audio focus, and current clip play.
BLUETOOTH	Writes to devices connected via Bluetooth.
CLIPBOARD	The system clipboard.
CONTACT_INFORMATION	The default contact information application's database.
DATABASE	A write of information into a database owned by this application. The application could be exported via a Content-Provider to other apps.
EMAIL	Commands that set fields of an email message including destination and message.
EXIF_INFO	Calls that alter EXIF data embedded in an image.
FILE	Write calls for files either shared or private, internal storage or external storage.
GOOGLE_SERVICES	Google services calls that change parameters or write data to connections to local or remote Google services such as Google Drive.
IO	Write calls to an I/O stream that could include files, serialization, network, or assets.
IPC	Communication between components of the AUT or to components of an external application. The payload or target field destinations for Intent-based communication, Service messages or AIDL (RPC) calls.
LOCATION	Set location provide or set current location on device or a location-aware stock application.
LOG	The Android logging framework. Since Android 15, applications cannot read the log messages from other apps.
NETWORK	Network write calls to either a local address or remote address.
NFC	Near-field Communication (NFC) write.
OS_COMMAND	Calls that set the command strings for starting command-line processes.
PHONE_CONNECTION	Commands that can modify phone connection actions such as initiating a call, hanging up from a call, or sending Dual-tone multi-frequency (DTMF).
PHONE_STATE	Commands that change setting of the device related to phone operation such as ringtone and connection notifications.

Sink Category	Data Flows To
PROCESS	Calls that execute command-line processes. The data flows to command-line arguments or the command name.
REFLECTION	Arguments passed to a reflected method invoke.
SCREEN	Calls that modify text displayed to user in UI widgets and views.
SERIALIZATION	Serialization routines for converting objects to bytes and writing bytes to IO streams.
SHARED_PREFERENCES	Writes to shared and private persistent key / value stores.
SMS_MMS	Payload, control, and destination fields for SMS and M M S messages.
SYNCHRONIZATION_DATA	Payload data sent to Android service for app data synchronization.
SYSTEM_SETTINGS	Writes to a variety of system settings that are found in the "Settings" app on the device include call forwarding, roaming, radio state, etc.
USB	Writes to a USB-connected device.
VOIP	Commands that affect connection state and settings for VOIP and SIP protocols.

5.6 Defining Authorized Context for Sensitive Source and Sink Flows

When reporting on a sensitive source to sensitive sink flow, DroidSafe also reports the context of the flow. The context includes three pieces of information:

- The user or system event that triggered the sink call. For example, was the sink call initiated by a user button press.
- Possible values for primitive or string arguments passed to the source call. For example, if the source is a file read, the path of the file read.
- Possible values for primitive or string arguments passed to the sink call. For example, if the sink is a network write, the IP address of the socket connection.

The information flow policy can additionally decide for each sensitive source and sink, what context values are non-sensitive, sensitive, or unauthorized. The next two subsections provide more information.

5.6.1 Triggering Event Context

DroidSafe reports on the triggering runtime event that triggered the sink call. For example, a location-to-network flow could be triggered by a button press and an Activity lifecycle onResume() event. DroidSafe currently does not categorize triggering events into high-level categorizations, so

it is the task of the policy writer to consider all possible event types in the API, and decide if any required special attention.

Also, for certain events addition context information is provided. For example, for a button press, DroidSafe attempts to resolve the name of the button (as defined in the XML user-interface resources or set programmatically), and DroidSafe presents the button name to the analyst.

For example, it might be non-sensitive for a network write to be initiated by a button press that signals that data will be written to the network, the button is named "Send Location". However, if the button name does not denote that it performs a network write, such as "Back", the flow could be unauthorized, and the application rejected.

As DroidSafe continues to mature, it may in the future include high-level classifications for triggering event types to help an information flow policy author consider all possible triggering event types for sensitive sinks.

5.6.2 Source and Sink Argument Value Context

The effects of a source or sink API call is often controlled by the arguments to the call (including the state of the receiver object). For example, a read of a file stream is affected by the file object to which the stream is linked. For sensitive and unauthorized source-to-sink flows, DroidSafe reports additional context on the source arguments and the sink state.

Table 5.4 provides an overview of the types of context resolved and reported by DroidSafe for sources and sinks that include the tracked state.

The information flow policy writer should define restrictions or whitelists for any of the object types of importance to the security policy of the organization. For example, for network reads and writes, it might be required to limit them to a certain IP address range or hostname regular expression. This whitelist can be documented and provided to the analyst such that all network-related source and sink calls reported by DroidSafe are checked to target only addresses that are in the white list.

Another illustrating example is to define a limit on Intent state. The policy writer should decide if applications should have the authorization to communicate with other applications, and if so, what are the actions (and data) and target applications that are authorized.

It is strongly recommended that the policy author create whitelists of any of the values in Table 5.4 (as opposed to blacklists).

5.7 Using DroidSafe to Check an Application Against an Information Flow Policy

To reiterate, it is the task of the information flow policy writer to: 1) provide the classification for each of the sources and sink categories, 2) provide classifications for the two combination cases, and 3) provide whitelists for sensitive object state types. Once this is complete, the policy can

Table 5: Context object types and their state fields

Object	State Name	Notes
URL	Protocol	Each state component is a String regular expression (RE). See Java URL documentation for more information.
	Authority	
	Host	
	File	
	User Info	
	Path	
	Query	
URI	Value	String RE for possible dynamic values.
Socket	InetAddress	String RE for remote host name.
	Remote Port	Remote port number.
	Local Port	Local port number.
Datagram Packet	Address	String RE for remote host name.
Class (Reflection)	Class Name	String RE for dynamic class type
File Streams	File Name	String RE for file path
Intent Filter	Actions	String RE for actions filtered
Component Name	Package	String RE for package of component.
	Class	String RE for class of component.
Process Builder	Command	String RE for command executed.
Database Cursor	URI	String RE for database URI linked to cursor.
Intent	Categories	List of String RE for categories.
	Action	String RE for Action.
	Data	String RE for URI describing data.
	Type	String RE for Type.
	Package	String RE for target class's Package.
	Component Name	Component Name for target Component
	Class	String RE for target Class.
Method (Reflection)	Declaring Class	String RE for declaring class.
	Name	String RE for name of method.

be distributed to analysts for review and incorporation. Also, the policy can be translated into a DroidSafe reporting policy.

Now, for each AUT analyzed by DroidSafe, DroidSafe will provide a list of all sensitive and unauthorized flows, with context when resolved. If a context state element is unresolved, it means that the analysis could not calculate values for the context element, and it should be interpreted that the state could take on any value.

Screencast walkthroughs covering the procedure for vetting an application have been delivered during the APAC program by the DroidSafe team. These videos are the best training material for using DroidSafe to vet an application with respect to an information flow policy.

5.8 Conclusion

This chapter covers how to create an information flow policy for Android applications. The information flow policy classifies exfiltrations of sensitive information derived from the Android API as either non-sensitive, sensitive (requiring further analyst review), or unauthorized. For sensitive flows, context state can be utilized to make the determination as to whether the sensitive flow should be authorized. DroidSafe provides context state for source-to-sink flows, and the information flow policy author has to reason about and create a whitelist for state that is authorized for sensitive flows. Finally, once the information flow policy is concretized, it can be employed by the analyst to interpret the DroidSafe results, and guide the decision as to whether to approve 3rd-party Android applications.

6.0 ANDROID APPLICATION CODING STYLE GUIDE

6.1 Overview

The goal of this chapter is to provide an overview of the effects of coding styles and implementation choices on the precision, scalability, and accuracy of the DroidSafe analysis system. The chapter is organized by focusing on various important analyses, transformations, and semantic modelings in the DroidSafe system; and for each, a discussion of Android API coverage and how application implementation choices could affect the analysis result.

This chapter should be read by both application developers and security analysts. For application developers, this chapter provides guidance on application implementation such that the developer can support the DroidSafe analysis and human analyst; the security analyst is likely to reject an application if implementation choices lead to imprecision or inaccuracies that confuse or look malicious, even if the application is benign.

For security analyst, this chapter helps them to understand sources of imprecision and inaccuracy such that guidance can be given to application developers and informed decisions can be made regarding the intent and behavior (malicious or benign) of an application with respect to its DroidSafe analysis result.

6.2 Coding Style Overview

The DroidSafe [1] system (covered in Chapter 4) relies on static analyses to resolve security sensitive behaviors and context for those behaviors. The underlying static analysis is conservative. It must be able to statically prove the conclusions that it draws. In practice, this means security-relevant code should be "straight-forward", e.g., little control flow, limited global variables, utilize constant values. It is often the case that existing Android applications follow these practices because API calls accept values that are known at compile time: Intent action values, onClick handlers, etc. If application code cannot be accurately analyzed, it could lead to unresolved context for API behaviors, or control flows / data flows that cannot be realized at runtime.

6.3 DroidSafe Analysis Report

The DroidSafe analysis system is a complex analysis system that includes a comprehensive and large model of the Android API and runtime. It is not possible to *exhaustively* describe the analysis

precision and accuracy effects of coding style and implementations decisions. Though this chapter covers the important implementation and coding-style choices that can affect precision and accuracy of the analysis, to best understand the specifics of a given application, the application should be analyzed with DroidSafe and the analyst or developer should inspect the output.

For each run of the analysis, DroidSafe produces a report of implementation choices that may affect precision and accuracy. This report is produced in the working directory at droidsafe-gen /analysis-warning-report.txt. The report includes a listing of source code locations (and line numbers when available) that may trigger precision and accuracy issues. The report is prioritized by the severity of the issue.

For the remainder of this chapter, the analysis report is given as the final reference for style and implementation decisions for each category with respect to an application under test (AUT).

6.4 Android Device Implementation Model

The DroidSafe system includes a comprehensive, precise, and accurate model of the Android API and runtime. Each AUT is analyzed in the context of the model, to exercise the application event callbacks and to account for the semantics of the API operations. The model is termed the Android device implementation (ADI), and it can be thought of as representing the semantics of the Android API and runtime for a theoretical device or phone. The ADI currently supports most commonly-used Android API operations and components of Android API level 19. However, it is not a complete model. Furthermore, it is not realistic to produce a list of the methods, classes, and packages that are modeled in this chapter.

For an application under test, the analysis report includes output that describes possibly reachable Android API calls that are not modeled, and thus their use could produce precision or accuracy problems with the analysis of the AUT. For full confidence in the analysis conclusions, it is recommended that any un-modeled API calls be removed or reimplemented to use modeled API components.

6.4.1 Fallback Modeling

The ADI includes runtime modeling that exercises callbacks in the application. However, since the ADI is not a complete model of the Android system (due to its nature as a proof-of-concept), the analysis includes mechanisms to detect possible inaccuracies in the ADI and correct them automatically. One such mechanism will search for API callbacks in an application, and if they are not exercised by the modeling, the mechanism will exercise them in a way that is not completely accurate to the Android runtime, but will expose much of the semantics of the callback. Callbacks that had to be exercised by this *fallback modeling* mechanism are reported by the analysis report.

Additionally, for API calls in the application that return a reference, but the points-to analysis concludes that the reference is always null are a good signal that the ADI model is inaccurate. This heuristic is used by the fallback modeling, and if a such a call is found, the fallback modeling will insert a generated object of the appropriate type (with a search for casts on the reference)

to attempt to account for the inaccuracy. The analysis report will list the code locations in the application where this transformation occurred.

For both of the above heuristic transformations, if the analyst sees these warnings, the remediation is either 1) the application should be rewritten to use calls that are modeled by ADI or 2) inspect and correct the modeling issue in the ADI (a discussion of this is beyond the scope of this chapter).

6.5 Application Resources and XML User-Interface Declarations

The DroidSafe analysis system reasons about user-interface and string resources defined in XML files. The semantics of the declarations are incorporated into the AUT for commonly-used idioms for Android resource declarations. The next two sections give more specifics.

6.5.1 User-Interface Resources

DroidSafe parses XML files that define user-interface elements, and inserts semantics to into the AUT representation to capture the semantics of the objects created by the XML declarations.

Common forms of the calls that inflate and return references to user-interface elements declared in XML are supported, e.g., findViewByID() and setContentView(). These calls must be passed resource identifiers that are constants. If a non-constant value is passed, the inflation will not be inserted, and semantics might be missed (thus leading to possible analysis inaccuracy). Also, the resource identifier passed as an argument must be either an application resource or a resource of the ADI.

The analysis report will list all user-interface resource methods that could not be understood.

6.5.2 String Resources

Calls in the AUT that access string resources are replaced with the string declared in the US locale. DroidSafe does not support string replacement for non-constant accesses to resources. Non-constant accesses to strings resources will be reported in the analysis report. Currently, the DroidSafe Android resource transformation does not support localization.

6.5.3 Application Manifest

DroidSafe supports parsing of the Android manifest as defined in Android API 19. The component of the manifest that are important for the analysis are parsed and the semantics incorporated into the representation of the AUT. If there is a component of the manifest that is not supported, the analysis report will contain an entry detailing the issue.

6.6 String Values and DroidSafe's String Analysis

In many cases, string arguments to API calls determine the semantics of security-sensitive operations such as inter-component communication targets and file operations. String values that are passed to Android API calls should be built in a straightforward manner preferably from program constants.

DroidSafe includes a powerful string value resolution analysis. This analysis attempts to resolve arguments to API calls from the application code. For each string argument in an API call, a backwards analysis attempts to resolve a constant or a regular expression describing the language of the string. The string analysis result is then used by the inter-component resolution and file system precision transformation passes (see below) to increase precision while maintaining accuracy.

If for some reason the pass does not finish, the analysis report will contain an entry denoting this condition.

Briefly, the string analysis reasons about commonly-used string operations like concatenation. It can create an internal model of two strings that are concatenated (for example) when the result is used as an argument to an API call. If one of the strings is a program constant, then the analysis will "know" something about the string and could use that knowledge to disambiguate certain operations.

In general, create and use strings arguments to API calls locally (within one method) and do not use containers to store and retrieve string arguments for API calls.

If there exists security sensitive API calls for which one or more string arguments could not be resolved to a required level for precision transformations, there will be an entry in the analysis report. See below for more details on the individual transformations that are clients of the string analysis.

6.7 Precision Increasing Transformations

The designers of the Android API decided to favor flexibility and extendibility over analyzability. In the Android API dynamic values dictate many potentially security sensitive actions such as inter-component communication (via Intents), file operations, network operations, and database operations.

To increase the precision of the DroidSafe result, it is necessary to apply the string value analysis resolve to dynamic values that affect security sensitive operations. For example, the *action* string of an Intent could dictate the target component for the Intent, and the string could be built through multiple string operations and user input.

6.7.1 Inter-component Communication

The Android API and runtime includes various calls for communicating between the building-block components of an application; we call these idioms *inter-component communication* (ICC).

Most of the same API calls are also used to communicate between different applications, and DroidSafe does include experimental support for resolving flows *across* applications, however, a discussion of these features is beyond the scope of the chapter.

The DroidSafe analysis reasons about communication (flows) between components of a single application (AUT). Without sophisticated analysis of target resolution, an analysis must connect flows into the payload of a communication statement with all possible target components based simply on type. The DroidSafe system includes aggressive string analysis and Android-specific transformations that seek to increase the precision of target-component resolution. This section details the implementation decisions that affect precision of ICC target resolution.

The target of ICC calls are in most cases defined by the android.content.Intent (hereafter referred to as Intent) passed to the call. Thus resolving values for Intents are crucial.

6.7.1.1 Intent Guidelines

There are two types of Intents in Android, *explicit* and *implicit* Intents. Explicit intents explicitly denote the target component via a class designation for the target. Implicit Intents denote an action and it is up to the runtime system to decide the target of the Intent based on the components that register to handle the action and arguments.

The developer should strongly favor explicit Intents. If the component name or class fields are set in code, an Intent is considered explicit. If setting the explicit Intent target with a Class object, use the class constant returned from Clz.getClass(). If using android.content.ComponentName, use string constants to create the component name.

If an explicit Intent cannot be resolved by inspecting the component name or class fields, then it will conservatively target all appropriate components of the application given the ICC call.

Implicit Intents require more machinery to resolve to targets. Resolution requires resolving potential runtime values for the action, categories, and type string fields; and also the data field which is a uniform resource identifier (URI). The string fields should be set from program constants (or local string operations on program constants). The URI type, if present, should be constructed from program constants. A failure to resolve all of the fields to constants (or not assigned) will result in a conservative estimate that the Intent can target all components of ICC call in which it is passed.

For a resolved implicit Intent and an associated ICC call, the targets are defined by the application components that register *Intent Filters*. Intent filters are registered in the manifest of the application, but can also be defined and registered dynamically for BroadcastReceivers. It is strongly recommended to define all Intent filters in the application manifest, though DroidSafe does try to aggressively resolve dynamic Intent filter registrations.

6.7.2 File System Operations

It is very difficult for a static analysis to disambiguate flows through file system read and write operations. For example, to conclude that a value written to a file cannot be read by a particular file read, the file name used by the read and write operations must be resolved, and soundly concluded to be different. Luckily, Android guards file operations via its API (for applications that do not include native code). By default, DroidSafe does not attempt to disambiguate file system operations, and reports the flows through the file system as if all operations target the same file and offset.

However, DroidSafe includes a mode, disabled by default, that attempts to add precision by resolving file names and file objects and soundly disambiguating flows through file system operations. The following properties must hold for the high-precision file operation mode to be applicable to an AUT:

- No use or allocation in the app of the following API classes: android.util.AtomicFile, java.lang.ProcessBuilder, java.io.FileDescriptor, java.io.RandomAccessFile, and android.content.res.AssestFileDescriptor.
- All file operations that open a file or file input / output stream must have a target parameter that resolves to a constant. For example, for a call to Context.openFileInput(String), the string analysis must be able to resolve the argument to a string constant. Thus the argument must be constructed from program constants (or string resources).

If a target parameter (file name) for a file operation cannot be resolved by the analysis, then high-precision file operation mode will not be enabled (and the offending statement will be reported in the analysis report).

The high-precision file operation mode includes support for predefined file locations accessed via the android.content.Context class, e.g., getFilesDir() and getObbsDir(). These predefined locations can be concatenated with strings or other file objects (e.g., via File constructors that accept a parent File object).

6.7.3 Reflection Guidelines

Reflection is an inherently dynamic idiom that has the potential to introduce significant loss of precision if used. It is strongly recommended not to use any of Java's (Android's) reflection library. However, DroidSafe does try to aggressively resolve precise targets for reflected methods and reflected allocations.

If the developer must use reflected invokes, follow these guidelines:

- The method must be invoked with java.lang.reflect.Method: invoke(Object, Object ...).
- The method must be retrieved with java.lang.Class.getMethod(...).

- The Class object that is the receiver to getMethod() must be resolvable, e.g., the getClass() method called on a static class name or a resolvable string.

If the developer must use reflected allocation, follow these guidelines:

- The object must be created using the java.lang.Class: newInstance() method.
- The Class object that is the receiver to newInstance() must be resolvable, e.g., the getClass () method called on a static class name or a resolvable string.

The analysis report will report on all uses of reflection, and highlight the cases where the reflection is not resolved. By default, unresolved reflection is not handled accurately.

6.8 Conclusion

This chapter provides an overview of the important implementation choices that could affect the precision, accuracy, and/or scalability of the DroidSafe Analysis. We hope that it is useful for application developers seeking to support application vetting via DroidSafe and analysts that employ DroidSafe to increase the accuracy and throughput of application vetting.

7.0 FORMAL FUNCTIONAL CORRECTNESS PROOFS OF APPLICATIONS

7.1 Introduction

Android devices [51] are vulnerable to security compromises carried out by rogue apps that may abuse the user's trust by masquerading as benign apps [52, 53]. The Android security mechanisms are coarse and complex [54, 55] and may be bypassed via exploitable flaws in the platform [56, 57].

A more detailed characterization of an app's behavior, especially its access to user data, can enable users to make more informed decisions about trusting and installing the app. A suitable formal specification of the app can be used for this purpose, and trust can be established via a formal proof that the app's code satisfies the specification. This requires a formal model of the platform that the app runs on—both language and API.

The work described in this chapter contributes to the goal of establishing trust in apps based on formal specifications and proofs. We used the ACL2 theorem prover [58] to build a formal model of a subset of the Android platform that supports non-trivial apps. We developed a proof methodology based on induction and symbolic execution of the app's event handlers, showing that each handler preserves the app's invariant, which includes all properties of interest, including functional correctness.

We applied this proof methodology to verify the full functional correctness of a slightly simplified version of a calculator app written by one of the APAC Red Teams. For a version of the app that contains malware, the correctness proof fails in a way that reveals the malware. In the process of verifying the app, we also uncovered a subtle functional bug that may be representative of malware that is triggered by complex conditions on an app's state and whose malicious action is the calculation of incorrect results. This "functional malware" differs from more explicit, and potentially more easily detectable, malware that, for example, sends private user data to a remote server when the device is in a certain location at a certain time. The latter kind of malware makes API calls to test the trigger conditions and perform the malicious actions, while functional malware may not make any suspicious API calls. For example, functional malware in a navigation app could deliberately lead users off course, perhaps even directing them to dangerous places.

Our approach is sound, precise, and high-assurance. It complements DroidSafe's static analysis approach, which focuses on undesired information flows. Our approach can prove virtually any true property about an app, with high assurance. Its main disadvantage is that it requires significant user effort, but we are working to improve the automation of the proof process.

Our work makes the following contributions:

- A formal model of a non-trivial subset of the Android platform.
- A formal proof methodology for Android apps.

7.2 Background

7.2.1 Android

Most Android apps are written in Java [59]. Besides using a subset of the standard Java API, these apps use the Android API, which provides access to hardware devices (camera, GPS, etc.), GUI elements (buttons, text boxes, etc.), inter-app communication (e.g., to open a given URL in a web browsing app), and so on. In addition to the Java source files, an app contains other resources, which often take the form of XML files. An app's Java source code is compiled to Java Virtual Machine (JVM) bytecode [60] using a standard Java compiler. The Android development tools are used to convert the JVM bytecode to Dalvik bytecode [51], which is assembled with the XML and other resource files (e.g., images) into an installable app package.

An Android app is structured in terms of 'activities', each of which is a single "screen" in the app's GUI. Within an activity are various 'views'—rectangular regions of the screen that represent GUI elements, such as text boxes and buttons that can be clicked. Events in Android include click events for these views. An app can register listeners for such events, either statically in its layout XML or programmatically by calling setOnClickListener(). When these events occur, the Android GUI thread invokes the appropriate methods of the registered listeners. An app's XML 'manifest' indicates, among other things, the initial activity to be created when the app starts.

Android also includes lifecycle events (*Create*, *Start*, *Resume*, *Restart*, *Pause*, *Stop*, and *Destroy*) that can be dispatched to the app. The sequencing of these events must be consistent with the activity lifecycle state machine [59] in Figure 4.1 (a typical flow is: *Create*, *Start*, *Resume*, *Pause*, *Stop*, *Destroy*) but can otherwise occur at any time. For example, a *Pause* event may occur when another app opens in front of the current app. Apps typically implement handlers to respond to these events (e.g., to save data when the app is paused) by overriding methods of the Activity class, such as onPause().

Various entities belonging to the app are identified using numeric resource IDs. These resource IDs are defined in special classes, namely the R class ('resource' class) and its inner classes, which are generated by the Android development tools. For example, an XML layout entity <Buttonandroid:id="@+id/btnSeven"...> will cause the R\$id class to contain a final static field called btnSeven whose ConstantValue attribute is some large, unpredictable number, e.g., 2131034114. In the Java source code of the app, the button object can be obtained by the method call findViewById(R.id.btnSeven), but in the bytecode only the numeric ID is present.

Android includes a permission mechanism to limit apps' access to hardware and other resources. For example, an app must possess the INTERNET permission to open network sockets and the CALL_PHONE permission to initiate phone calls. An app declares, in its XML manifest, the set of permissions that it requests. When an app is about to be installed, the requested permissions are shown to the user, who decides whether to proceed with the installation, and thus grant the

app all the requested permissions. This permission mechanism is coarse-grained: for instance, the INTERNET permission gives an app carte blanche to connect to any host at any time to send any data.

7.2.1.1 Malware

Several kinds of malware affect Android devices [52, 53]. Tools like [61] can be effective at detecting malware that exfiltrates private user data by (necessarily) making suspicious API calls. The mere presence of certain API calls may be suspicious, e.g., an app that opens a network connection, when the app's purported functionality does not involve the network. The presence of an API call may be legitimate, but the information that flows to the API calls may be suspicious, e.g., an app reads a user's contacts and sends them over the network, when the app's purported functionality does not include that.

A more stealthy kind of "functional malware" may not exfiltrate private user data, and instead intentionally calculate incorrect results. The severity of this kind of malware depends on how much the user relies on the app calculating correct results: it may range from an annoyance to loss of life, e.g., if a military navigation app sends a squad off-course to a dangerous place. Functional malware may be triggered under complex conditions on an app's state variables, eluding detection via code inspection. Functional malware may involve API calls, but not necessarily suspicious ones; or it may not involve any API calls.

Unlike many other approaches, our work addresses functional malware. Of course, it also addresses inadvertent errors. The difference between functional malware and an unintentional bug is one of developer's intent; but the impact may be similar. Our app verification approach establishes functional correctness, ruling out both intentional and unintentional bugs.

7.2.2 ACL2

The ACL2 theorem prover [58] consists of a first-order specification language based on side-effect-free Common Lisp and automated proof methods for reasoning about programs and models written in the language. Two strengths of ACL2 are its sophisticated term rewriter and its heuristic application of induction [62]. ACL2 supports reasoning about programs written in languages other than its native Common Lisp dialect via embeddings that capture the languages' semantics in terms of ACL2's native language. Below we describe how we use this approach to reason about JVM bytecode representing Android apps.

7.3 Platform Modeling

Since our motivation for modeling the Android platform is app verification, our formal model describes not the internal structure and layers of the platform stack, but the top-level interface that the platform provides to apps. This interface consists of the language that apps are written in and the API calls exchanged between apps and platform, including callbacks.

7.3.1 Formal JVM Bytecode Model

To reason about an Android app, we intercept its JVM bytecode during compilation (before Dalvik bytecode is generated). To assign semantics to this bytecode, we defined in ACL2 a formal model that is an executable interpreter of the Java Virtual Machine [63]. Our model is similar to the M5 model developed by J Moore and others [64], but covers more features (e.g., exceptions, string interning, and class initialization). Theorems about JVM bytecode programs are expressed using this formal model; we prove that when the program of interest is executed on the model, starting from a state where certain properties hold, then certain other properties always hold on the resulting state. This follows the style pioneered in [65].

While we do not consider our JVM model to be a novel contribution of our work, we summarize its behavior here for concreteness. The state of the JVM in our model includes the Java heap, static area (where static fields are stored), and, for each thread, a call stack that includes invocation frames for each method that the thread is currently executing. Also included are auxiliary data structures for synchronization and locking, string interning, etc.

Each JVM instruction is modeled by specifying the effect on the JVM state when that instruction is executed. For example, the iadd instruction for integer addition is modeled as follows:

```
(defun execute-IADD (th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (push (bvplus 32
                               (top (pop (stack (top-frame th s))))
                               (top (stack (top-frame th s))))
                       (pop (pop (stack (top-frame th s)))))))
```

The function execute-IADD modifies the data structures of thread th in the JVM state s. In particular, it pops two operands off of the operand stack in the top invocation frame of the call stack, adds them, and pushes the sum back onto the operand stack. It then increments the program counter :pc by 1, which is the length of the iadd instruction.

To run an entire program, we repeatedly step the machine state by fetching and dispatching on the next instruction. We use ACL2's defpun utility to soundly introduce the JVM interpreter as a partial function [66].

A crucial feature of our JVM model is that, in addition to running bytecode programs on concrete inputs, it can be used for symbolic execution of bytecode programs on arbitrary inputs. A typical theorem says, in essence, "When we run the JVM model on this bytecode program, for any input satisfying this predicate, the resulting state has the following properties." The symbolic execution is performed using the ACL2 rewriter to repeatedly step and simplify the state, symbolically executing one instruction at a time and building up a symbolic representation of the current state in terms of the symbolic inputs. This technique is standard in the ACL2 community. In this way, our formal JVM model captures the semantics of the JVM bytecode language and allows us to reason about the code that constitutes Android apps.

7.3.2 Formal Android Model

We extended the formal JVM model described above to a formal model of the Android platform, capable of executing and reasoning about simple Android apps. A state in the Android model contains a JVM state and several additional Android-specific state components More precisely, our model of the Android state contains:

- A JVM state, as discussed above. This contains the persistent data used by the app, including its heap and static fields.
- The app's activity stack, including the current activity on top of the stack, and any activities that are currently paused, below the top activity.
- The set of currently allowed events (e.g., button clicks) for which the app has registered event handlers.
- A parsed representation of the app's manifest—see Section 7.2.
- The app's layout information, parsed from the app's XML layout files and indexed by the layouts' numeric IDs. This includes information about the views (e.g., buttons) in the app's GUI and their associated event handlers (e.g., onClick listeners) and is used by our model of the setContentView() API method when it constructs the GUI for an activity.
- A map from the addresses of View objects to their listeners, used to dispatch control when handling events. A listener is a pair of a method (often, but not always, the onClick() method of some class) and an object on which to invoke the method (often this is an Activity object or an instance of an anonymous class whose sole purpose is to define the listener). This map is updated by our model of the setOnClickListener() API method.
- A map from symbolic string names of views, used in the layout XML, to the corresponding numeric resource IDs. This is used to translate events from user-meaningful form to internal form. We build this map by inspecting the names and values of the static fields of the R\$Id resource class generated when the app is built.
- A map from resource IDs to the addresses of their corresponding View objects. This is used to determine the actual objects on which to dispatch events (e.g., click events) and by our model of the findViewById() API method.
- The API call history, a ghost variable that lets us reason about the API calls that the app has (and, critically, has not) made, including a record of the event whose handler made each API call.
- The event history, a ghost variable that lets us talk about the sequence of events given to the app so far. If we are verifying that the app implements an abstract state machine, we can abstract this event history and feed it to the abstract state machine. The resulting abstract state should then be the abstraction of the machine's current concrete state. Proving that this property is preserved by all event handlers in the app is the core step of our app proof methodology described below.
- The event currently being handled, if any, so that we can record in the API history which event was being handled when the API call was made. API calls may be allowed for some events but not others. For example, a sound recorder app may be allowed to start recording only when the user presses the *Record* button.

7.3.2.1 Event Handling

Our Android model supports running an app on a sequence of input events, by executing their event handlers in order. This can be done on a concrete sequence of events, to test an app. More importantly, it can be used for proof. We prove that, for any sequence of events, running the app's handlers for those events preserves the app's invariant. At this level, events are represented in terms that are meaningful to the user. For example, (:resume) represents the event that resumes the current activity, and (:click"myButton") represents a click of the button whose name in the layout is myButton. In order to actually handle these events, our model must determine the objects on which the handler methods should be invoked, so it first converts the events into an internal form. For lifecycle events, this adds to the event the heap address of the topmost activity object on the activity stack, giving something like (:resume12345). Click events are internalized by mapping the symbolic name of the button to a numeric resource ID and then to the actual address of the View object with that ID, giving something like (:click6789). Currently our model only handles lifecycle events and click events, but adding support for other events should be straightforward.

Once the event has been elaborated to internal form, we dispatch it to the appropriate handler by executing the code for the handler using the underlying JVM model. For a lifecycle event, we execute an invokevirtual instruction for the appropriate handler method (e.g., onResume()) on the given Activity object, which causes the app's onResume() handler method to run. Such methods almost always begin by calling through to the corresponding method of the parent class, e.g., super.onResume(). This causes code from the Android API implementation to run, e.g., android.app.Activity.onResume(). Our model includes special modeling for these lifecycle API calls. For example, the model for onResume() causes the onClick listeners in the resuming activity to again be added to the set of allowed events. To handle a click event, assuming it is already in internal form, we look up the onClick listener for the given View object and call the indicated method. In our model, handlers execute to completion and cannot be interrupted. This corresponds to Android's use of an app's main 'UI thread' to execute its handlers. Future work would include adding support for background services, which an app can use to offload expensive computation from its UI thread.

The sequential processing of events in our model corresponds to the way in which the Android platform internally enqueues events and delivers them to an app's unique UI thread. By proving properties over all possible event sequences, we ensure that the properties hold no matter how the Android platform enqueues and delivers the events.

Events that are not currently allowed by the app (according to the set of allowed events in the Android state) are ignored, e.g., a click on a view that has no registered onClick listeners, or an illegal lifecycle event, such as stopping an activity that has not been started. Every event is also recorded in the event history, so that the invariant can refer to the state that the app should be in, given the events seen so far.

7.3.3 Formal API Model

A major challenge in reasoning about Android apps is to properly model calls to API methods. We are following a "demand-driven" approach in which we add models of API methods as we encounter calls to them in apps that we want to verify. Some methods such as sendTextMessage() do not really need to be modeled because they affect only the external world, not the state of the app itself: we simply record them in the API history, so that we can express properties such as "the app has not sent any text messages", and continue with execution. When the API call does affect the app's state, if possible we simply execute on our model the actual code of the API from the Android implementation. API calls treated this way include many calls in java.lang (e.g., dealing with Strings and Enums) and setters and getters such as Activity.setTitle() and View.isClickable(). There are situations where simply executing the API call does not work, either because the code is unavailable (e.g., native methods) or too complicated, or because it affects parts of the Android state that we model. To model such methods, we define executable ACL2 functions and include them in our Android model. Methods that are modeled in this way include setOnClickListener(), findViewById(), setContentView(), and the activity lifecycle event handlers onStart(), onResume(), etc.

Our model of running an app begins by building an initial Android state for the app (where many components, such as the API history, are initially empty) and then calling the app's onCreate() method. Further events are then handled in order.

7.4 App Verification

Our platform model provides a formal semantics for non-trivial Android apps. This allows us to formally prove that apps satisfy their functional specifications, which implies the absence of the kind of functional malware discussed in Section 7.1.

Our methodology is based on formulating an invariant for the app: a predicate over states of the Android model that is preserved as the app runs. The invariant characterizes correct behavior, often using an abstraction to a high-level state machine, and also makes many Android-specific assertions, such as specifying the set of currently active event listeners. Each event is proved to preserve the invariant, using the ACL2 rewriter to perform symbolic execution, as described below. Failed proofs may require the invariant to be strengthened. Once an inductively-strong invariant is obtained, an induction over event sequences establishes that the invariant holds for all possible event sequences. This section discusses the app verification process in more detail, using the running example of verifying a calculator app.

7.4.1 Calculator App

The Red Teams of the DARPA APAC Program [67] developed several apps, including a calculator that applies the four arithmetic operations to floating-point numbers. Since our JVM bytecode model does not include floating-point numbers yet, we modified the app to operate on integers instead, using Java's normal modular arithmetic. We also slightly simplified the GUI of the app to

not use features that are currently not covered by our model. The malware in the app replaces the running result with a random number under certain conditions described later, but we simplified it to return a fixed result of 88888888 instead, because we do not yet model random numbers. These simplifications do not fundamentally change the structure of the app.

7.4.2 Representation

Our Android model includes a parser, written in ACL2, that turns an app's JVM bytecode class files and XML files into an S-expression-based ACL2 representation usable by our platform model.

A parsed app, with the platform underneath, forms a state machine. The initial state S_0 is defined by our model of app initialization discussed above. Each transition is triggered by a platform-initiated event (e.g., pause app, resume app) or a user-initiated event (e.g., click a button). The deterministic transition function T maps an input event E and a state S to the next state $T(E, S)$; it is lifted to sequences of events by defining $T^*((E_1, \ldots, E_n), S) = T(E_n, \ldots T(E_1, S)\ldots)$, and $T^*(\varepsilon, S) = S$, where ε is the empty sequence. Our platform model currently supports a single app (state machine) at a time, but can be extended to support multiple apps.

For the calculator app, the state machine has an input event for each calculator button (0123456789+ −*/=C) and each app lifecycle event. The state includes a TextView GUI object whose content is the string shown on the calculator display. The main correctness theorem for the app says that the contents of the display are always correct, given the sequence of input events supplied to the app so far. We defined an output function O that maps a state S to this display string $O(S)$. Different output functions could be defined for different apps, each extracting from the state the app-specific observables of interest.

7.4.3 Specification

The execution of the parsed app on the platform model corresponds to a low-level state machine whose states are states of our Android model, as described above, and whose transitions are expressed in terms of the execution of JVM bytecode and API calls. Often a functional specification for an app is naturally expressed as a higher-level state machine, whose states and transitions are defined in user-oriented terms rather than code-oriented terms. The correctness of the code with respect to the specification can then be expressed as a simulation [68] of the high-level machine by the low-level machine.

A state machine specification for the calculator app is sketched in Figure 7.1. Each state has a name (in bold, e.g., value) and one or more state variables (in italics, e.g., *val*); the underlined state variable is the one shown on the calculator display. In each state, *val* is the latest result, which is 0 when the calculator starts or when C (clear) is entered. In value-op and value-op-value, *op* is the latest operator entered. In value-op-value, entering = or an operator *op'* combines *val2* with *val* by applying *op*, completing the pending operation and replacing the latest result; if *op'* was entered, it becomes the latest operator. Figure 7.1 does not show the expressions assigned to state variables when transitions are taken, e.g., a *digit* transition from value-op-value to value-op-value assigns $10 \times val2 + digit$ to *val2*. Exploiting that 0 is identity for addition,

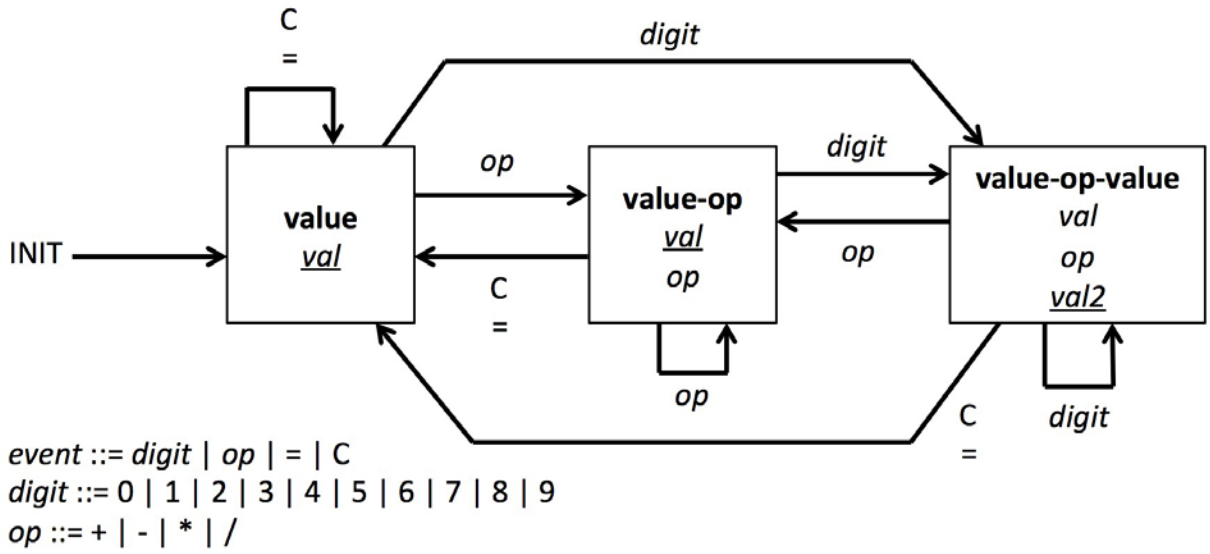

Figure 8: A state machine specification for the calculator app

entering a *digit* in value sets *val* to 0, *op* to +, and *val2* to *digit* , as if there were a pending $0 + \ldots$ operation.

We formalized this state machine specification in ACL2. The formalization includes a constant s_0 for the initial state, a deterministic transition function t that maps an input event e and a state s to the next state $t(e, s)$ (and is lifted to t^* over sequences of events, analogous to T^* above), and an output function o that maps a state s to the content of the calculator display $o(s)$.

7.4.4 Invariants and Proofs

Often the simulation relation between a low-level and a high-level state machine is defined as an abstraction function [69] from the low-level inputs and states to the high-level inputs and states. For the calculator app, the abstraction function α maps each calculator button press event to the corresponding input in Figure 7.1 and each app lifecycle event to no input in Figure 7.1; it also maps each app/platform state to a state in Figure 7.1.

In our Android platform model, the app/platform state S includes the history of input events. Thus, given an abstraction function to a high-level state machine specification, the correctness of the app with respect to the specification can be expressed as a predicate over the low-level app/platform states. Intuitively, the app's invariant says that the app is in fact in the state that it should be in, given the sequence of inputs seen so far. If $H(S)$ is the history of input events in S, the predicate is $\Omega(S) \equiv [O(S) = o(t^*(\alpha^*(H(S)), s_0))]$, i.e., the observable outputs that result from executing the app's code on the inputs $H(S)$, which take the initial state S_0 to S, are the same that result from running the high-level state machine on the corresponding abstract inputs $\alpha^*(H(S))$, where α^* is the homomorphic lifting of α from events to event sequences. If Ω includes all the states S reachable from S_0, i.e., if $\Omega(T^*((E_1, \ldots, E_n), S_0))$ holds for every event sequence $E_1, \ldots, E_n$, then the app's code is observationally equivalent to the specification, i.e., it yields the same outputs

for the same inputs. For the calculator app, the code is observationally equivalent to Figure 7.1. $\Omega(T^*((E_1, \ldots, E_n), S_0))$ is provable by induction if Ω is an invariant, i.e., if Ω holds on the initial state (base case: $\Omega(S_0)$) and is preserved by each transition (induction step: $\Omega(S) \Longrightarrow \Omega(T(E, S))$). Since Ω alone does not provide a sufficiently strong induction hypothesis, the following invariants are defined, and proved together:

1. A stronger correctness predicate that involves not only outputs (the calculator display) but also states: $\Sigma(S) \equiv [\alpha(S) = t^*(\alpha^*(H(S)), s_0) \wedge O(S) = o(\alpha(S))]$, from which the weaker $\Omega(S)$ is easily proved. While α, t, and S_0 are specific to the app under verification, Σ has the same form for every app whose specification is a state machine with an abstraction function, e.g., the calculator app.

2. Code-level predicates on the app's state, e.g., that a Java int field is never negative or is always within a certain range. Formulating these predicates requires an understanding of the app's code, but failed proof attempts in ACL2 often suggest them.

3. Platform-level structural predicates about the Java heap containing the objects that form the app under verification, the Android GUI objects being consistent with the XML files, Java fields having values of the right types, and so on. These constraints are largely boilerplate and we believe that they could be automatically generated at the same time as the app is parsed into its ACL2 representation. For the calculator, we manually defined several predicates of this kind, because their automatic generation is not implemented yet.

Once a sufficiently strong invariant has been defined, proving its establishment in the initial state and preservation by each transition can be carried out by symbolic execution using the ACL2 rewriter. To prove preservation, we start with an arbitrary Android state assumed to satisfy the invariant. We then show that the execution of an arbitrary event results in a state that still satisfies the invariant. The proof naturally splits into cases for each possible allowed event (disallowed events have no effect on the state), and we usually prove each event separately. Some application-specific rewrite rules are often needed (e.g., rules about bit-vector math for the calculator app), and the proofs also use our growing library of rewrite rules about the Android model itself. Otherwise, proofs for simple apps are largely automatic; for the calculator app, the proof corresponding to each button click event is a single line of ACL2 code that invokes our tactic called def-event-proof. This tactic unfolds the application of the invariant to the initial state (to expose necessary assumptions for symbolic execution), performs the symbolic execution, often resulting in several cases, and finally, in each case, unfolds the invariant applied to the final state and simplifies the result. In successful proofs, everything simplifies to 'true'.

A key intermediate formula that arises in the proof of the preservation of the invariant is $\alpha(T(E, S)) = t(\alpha(E), \alpha(S))$, i.e., each low-level transition has a corresponding high-level transition—a typical commuting diagram in simulation. If an app's code has no loops (as is the case for the calculator app), ACL2 can automatically prove the invariant's establishment and preservation, provided that an appropriate set of rewrite rules is enabled. The absence of loops is not so uncommon in simple Android apps, where the platform already provides a GUI loop that reads inputs and invokes app code to process them. Verifying apps whose event handlers contain loops is future work and will likely involve formulating and proving appropriate loop invariants; Σ and the other invariants discussed above apply to the platform GUI loop.

We found it convenient to verify the calculator app in two stages. We defined an intermediate state machine whose structure closely resembles the Java code, but without involving any Java or Android concepts. Its states are records whose components correspond to the app's Java fields, and its transitions are defined in terms of record component updates that correspond to the Java code. This intermediate state machine is an abstraction of the code in the ACL2 logic, which in particular does not involve the platform-level structural invariants discussed above. It may be possible to obtain this intermediate machine automatically, using the techniques in [63]. We prove that the app's code simulates the intermediate machine and that the intermediate machine simulates the high-level machine. The two theorems are composed to obtain a proof of correctness of the calculator app with respect to Figure 7.1.

7.4.5 Malware Discovery

The calculator app keeps a count of the operations performed since the last = was entered (or since the app started), e.g., after entering ...=1+2*3 the count is 2. The malware (in our simplified version of) the app replaces the running result with 88888888 when the count reaches 3. This is functional malware, which does not involve API calls.

We attempted to prove that the calculator app with malware satisfies the specification in Figure 7.1. As it should, the verification fails. The output from the failed ACL2 proof exposes the malware: a proof subgoal that cannot proved is that when the operation count is 3, the correct running result is 88888888. In general, failed proof subgoals can expose the conditions that trigger an app's malware and the malicious computations that violate the functional specification.

This is a very simple example of functional malware, which is also fairly easy to detect by the user. However, it is suggestive of more serious, and hard to detect, kinds of functional malware. An example is a military navigation app whose intentional miscalculations send a squad off-course to a dangerous place.

7.4.6 Functional Bugs

After manually removing the malware from the calculator app, we found two functional bugs in the app that prevented a successful proof. The bugs are also present in the original, unsimplified version.

The operation count is stored in a Java int, which wraps around and becomes negative if 2^{31} operations are entered without entering =. Since the condition under which the display is updated includes that the count is larger than 1, the display stops updating as the count becomes negative (until it wraps around again to become positive). Since it is impractical to enter 2^{31} operations, this bug has arguably only theoretical significance (some may argue that it is in fact not a bug). Nonetheless, we fixed this bug in the app code.

The other bug may occur in practice: under certain easily achievable conditions, the display is not updated to show the running result. For example, starting the calculator and entering -12345+ shows 12345 instead of -12345 on the display (the + should show the partial result $0 - 12345$,

where 0 is the initial display). The details of this bug are unimportant, but are caused by what we regard as an unnecessarily complicated implementation of the calculator: this bug eluded our manual code inspection. While this bug was not malware planted by the Red Team, and is not earth-shattering in its significance, it may be representative of functional malware where a cleverly crafted, non-straightforward implementation may sometimes produce an incorrect result under conditions that cannot be easily detected by manual inspection. After fixing this last bug, we proved the correctness of the app with respect to Figure 7.1.

7.5 Related Work

In [70], JML [71] is used to specify contracts for API and application methods, and the KeY theorem prover [72], which is based on dynamic logic [73], is used to verify that the Java code of those methods satisfies the contracts. Our formal model of the Android API is more comprehensive, e.g., we model callbacks, which are not modeled in [70]. The app specifications in [70] consist of contracts for various app methods, which are implicitly informally "composed" into an overarching correctness argument for the apps. In contrast, our app verification is carried out with respect to an explicit overarching app specification expressed in user-oriented terms (not code-oriented terms like contracts). The translator from Java/JML to KeY in [70] embodies the dynamic logic semantics of Java and JML and is thus a critical component of that approach; in our approach, all the semantics is explicated in ACL2.

In [74], a pencil-and-paper concrete and symbolic operational semantics for Dalvik and for a few Android API methods is defined, and used as the foundation to implement a symbolic executor of Android apps. The symbolic executor is connected to an SMT solver. The tool is shown to infer the conditions under which an example app performs certain privileged actions. Our approach also uses symbolic execution, but our semantics is mechanized inside a theorem prover, and we use ACL2's rewriter for symbolic execution. It is not clear whether their approach can verify the full functional correctness of apps, due to the use of an SMT solver rather than a more general (but likely less automatic) theorem prover such as ACL2.

In [75], a pencil-and-paper operational semantics for a few Dalvik instructions and a few Android API methods is defined, and a progress property is proved. The paper mentions work in progress on a symbolic executor, but no app verification results are reported. Our Android model is mechanized inside a theorem prover and covers more features of the Android platform.

Other formal models of the Android platform [76, 77, 78, 79] are more abstract than ours, focused on security aspects and properties. These formal models are in a sense complementary to ours: it should be possible to formalize abstraction mappings from our model to those models, ensuring that the security properties of the more abstract models apply to the more concrete model.

Static analysis of app code to help detect malware (e.g., [80, 81, 61]) is complementary to our approach. It is more automated (e.g., no functional specification is needed) but less precise; it cannot prove deep properties like functional correctness.

In [82], post-conditions of API method calls are calculated from pre-conditions via an algorithm that processes propositional formulas. It may be possible to use our API model and the ACL2

theorem prover for that purpose, which may lead to higher precision in the malware detection tool described in that paper.

Proposals to improve the Android security mechanisms (e.g., [83, 84, 85]) or to add on-device virtualization (e.g., [86]) require extensions to the platform, which the developers of all the fragmented versions of Android would have to agree on. If implemented, these extensions may prevent certain classes of malware, but not the kind of functional malware that our approach addresses.

Collecting data at run time and analyzing it to detect malware patterns (e.g., [87]) is likely to be more automatic than our approach but may allow malware to execute before it is detected. It also may raise privacy concerns if the analysis is performed off-device.

Dynamic analysis in off-device sandboxes prior to deployment (e.g., [88]) has similar coverage limitations as conventional testing. In addition, some malware may detect when it is being run in an emulator and behave differently than when it is run on a device.

Automatically transforming app code to enforce security policies (e.g., [89]) may affect performance and potentially functionality and may not be agreeable to app developers. This approach may thwart certain classes of malware, but not the kind of functional malware that our approach addresses.

7.6 Takeaways

7.6.1 App Verification Methodology

Many aspects of the app verification work described in Section 7.4 are not specific to the calculator app. We expect that the same proof methodology can apply to a large class of apps:

- Automatically parse the app's code and XML files into a deeply embedded representation inside the theorem prover, obtaining a low-level state machine based on the formal semantics of the JVM and of the Android platform, as in Section 7.4.2.
- Formalize the app's specification as a high-level state machine, expressed in user-oriented terms (not in internal Android-oriented terms), as in Section 7.4.3.
- Define an abstraction function from the low-level state machine to the high-level state machine, as in Section 7.4.4.
- Formulate a sufficiently strong state invariant on the low-level state machine (like Σ in Section 7.4.4) that implies the desired relation between the high-level state machine and the low-level state machine (like Ω in Section 7.4.4). The invariant includes not only simulation conditions, but also code-level invariants and platform-level invariants, as explained in Section 7.4.4.
- Use symbolic execution to prove that the low-level state machine's invariant is established by initialization and preserved by each event.
- If convenient, formalize intermediate state machines (between the low-level one and the high-level one), staging the abstraction functions accordingly. Prove simulations of each machine by the one immediately below it, and finally compose the simulation theorems into

one overarching simulation of the high-level state machine by the low-level state machine. As mentioned in Section 7.4.4, for the calculator app we used an intermediate state machine.

7.6.2 State Invariants vs. Trace Invariants

By keeping suitable history (e.g., the sequence of events processed so far) in our model of the Android state, we are able to express properties of interest (such as Σ and Ω in Section 7.4.4) as state invariants instead of more complex trace invariants, which involve multiple successive states of execution.

7.6.3 Iterative Invariant Strengthening

It may be difficult to formulate a sufficiently strong invariant in one attempt. The first attempt typically results in an invariant that is too weak. However, the failed proof output from ACL2 often readily suggests how to strengthen the invariant. The failed proof output consists of one or more proof subgoals, each consisting of a number of hypotheses and a conclusion. When these hypotheses express some impossible condition (e.g., that an integer variable is outside it possible range of values), the invariant must be strengthened to exclude that impossible condition (e.g., the range of the variable must be part of the invariant). Several iterations may be needed before reaching a sufficiently strong invariant.

7.6.4 Bugs Uncovered by Failed Proof Attempts

Bugs in the app (i.e., the fact that the app does not satisfy the specification) are often exposed by failed proof attempts. In some cases, the hypotheses of a failed proof subgoal, when they do not correspond to an impossible situation (i.e., the failed proof is not due to the invariant being too weak), reveal corner cases in which the invariant is broken. This may indicate either a bug in the app or perhaps a need to reformulate the invariant.

7.6.5 An ACL2 Trick

There are cases in which failed ACL2 proof subgoals do not explicitly expose the problem, because the ACL2 rewriter rewrites an untrue conclusion to 'false' and replaces it with the negation of one hypothesis—the untrue conclusion has disappeared from the proof subgoal. This happens, for instance, when attempting to prove that some term x equals a certain constant c, when instead the term equals some other constant c': The goal $x = c$ is rewritten to 'false' and it disappears. To debug this, we can introduce an uninterpreted nullary function f and attempt to prove $x = f()$. The new proof attempt will of course fail, but the rewriter will rewrite x to the correct constant c', displaying the failed proof subgoal $c' = f()$. Then we can revise our original proof attempt to prove $x = c'$ instead.

7.6.6 Android Platform Modeling

The Android documentation informally describes the interaction of apps with the Android platform, without explicitly describing most of the internal state of the platform, aside from app lifecycle states and similar aspects. Formalizing the Android platform involves creating an explicit model of the internal Android state. In order to do that, we tried to imagine how the implementation could support the behaviors described in the documentation (e.g., maintain a mapping from resource IDs to references to View objects), and defined our state (and transition) model accordingly.

7.6.7 Android API Modeling

The large size of the Android API makes its formal modeling challenging. We believe that the best approach to address this challenge is to model the API in a demand-driven fashion, i.e., formalize the API classes and methods as they are needed to verify apps. API methods written entirely in Java need not be explicitly modeled; instead, their code can be symbolically executed along with the app code. However, it may be beneficial to explicitly model API methods that have complex code that may complicate symbolic execution. It should be also noted that, as suggested in [82], typical apps use a relatively small "popular" subset of the Android API: thus, it is not necessary to model most of the Android API in order to verify interesting apps.

7.7 Conclusion and Future Work

We have described our ongoing work on formally modeling the Android platform and verifying Android apps. Compared to existing research, our Android model has the highest coverage of Android features, and our Android app verification goes deeper to include proofs of full functional correctness. A major motivation for this work is to ensure the absence of functional malware in apps, which other detection approaches to do not address. Our approach can be used to prove deep properties of apps with high assurance.

The proof methodology described in this chapter, based on state machines and simulations, can verify a large class of app properties. But the ACL2 logic and our Android model can express other kinds of assertions over the deeply embedded apps. Examples are program-level properties such as the fact that certain API calls are made only under certain conditions and with certain data, which enables much finer distinctions than coarse Android permissions such as INTERNET. Other examples are hyperproperties (i.e., predicates over multiple executions) [90], including security policies like non-interference [91], which could express the non-leakage of private user data to network sockets, text messages, and other destinations. To verify these kind of properties, extensions to our proof methodology may be needed, e.g., invariants over multiple states from different execution traces.

We are extending our formal model to cover more Android features and are tackling the verification of larger and more complex apps. We would also like to extend our approach to support reasoning about multiple apps, including their communication via Android's 'intent' mechanism.

Another direction for future research is the modeling and proof of non-functional aspects of apps, e.g., to reason about resource usage or covert channels.

8.0 CONCORD - VERIFYING MEMORY SAFETY

In this part of the project, we set out to explore how to move very precise static analysis and verification techniques from the realm of specialized research tools to an approach that can feasibly adopted by programmers in the real world. The critical issue is the lack of precision that such analyses inevitably encounter when analyzing the programs that occur in practice, no matter how advanced the automated techniques are. This lack of precision causes either 1) unacceptable numbers of false positive alarms or 2) the use of unsound techniques that may leave errors uncovered.

Our hypothesis we set out to test is that, in existing programs, only a small percentage of the code is responsible for this lack of precision. If this hypothesis is true, it opens up the possibility that we are much closer to a practical program analysis for verifying important security (and potentially other properties) than it currently appears.

For this, we developed a focused prototype version of a highly precise verifier based on the Compass tool [92] we call Concord. While our tool is only a prototype, we were able to test and validate our initial hypothesis and concretise the key challenges currently limiting the impact of program verification as well as formulate initial solutions to these challenges.

We have validated on real code that loss of analysis precision is generally caused by a small subset of the code. But more specifically, this insight leads to two technical challenges:

1. Where the precision loss starts is often far from where a spurious error is reported and very hard to identify by hand.
2. Once the piece of code too difficult to analyze is identified, non-expert programmers need to specify a correct alternate description or implementation.

In our project, we propose using *logical abduction* to identify the parts of a code base that introduce analysis imprecisions. In order for non-expert programmers to annotate, replace and/or describe the behavior of code segments that are beyond automatic analysis capabilities, we propose a technique we call *property programming* where programmers rewrite the small sections of code that are too hard to analyze. Where this is not possible (e.g. stubbing libraries, or very critical code segments), a small set of intuitive annotation primitives are embedded into regular program constructs (such as loops) to allow non-expert programmers to easily and succinctly express difficult constraints and invariants.

8.1 Logical Abduction

8.1.1 General Introduction

The fundamental ingredient of automated logical reasoning is *deduction*, which allows deriving valid conclusions from a given set of premises. For example, consider the following set of facts:

$$(1)\quad \forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x))$$
$$(2)\quad \forall x.\ ((\text{duck}(x) \vee \text{goose}(x)) \Rightarrow \text{waddle}(x))$$
$$(3)\quad \text{duck}(\text{donald})$$

Based on these premises, logical deduction allows us to reach the conclusion:

$$\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})$$

This form of forward deductive reasoning forms the basis of all SAT and SMT solvers as well as first-order theorem provers and verification tools used today.

A complementary form of logical reasoning to deduction is *abduction*, as introduced by Charles Sanders Peirce [93]. Specifically, abduction is a form of backward logical reasoning, which allows inferring likely premises from a given conclusion. Going back to our earlier example, suppose we know premises (1) and (2), and assume that we have observed that the formula $\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})$ is true. Here, since the given premises do not imply the desired conclusion, we would like to find an explanatory hypothesis ψ such that the following deduction is valid:

$$\frac{\begin{array}{c}\forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x)) \\ \forall x.\ ((\text{duck}(x) \vee \text{goose}(x)) \Rightarrow \text{waddle}(x)) \\ \psi\end{array}}{\text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})}$$

The problem of finding a logical formula ψ for which the above deduction is valid is known as *abductive inference*. For our example, many solutions are possible, including the following:

$$\psi_1 :\ \text{duck}(\text{donald}) \wedge \neg\text{quack}(\text{donald})$$
$$\psi_2 :\ \text{waddle}(\text{donald}) \wedge \text{quack}(\text{donald})$$
$$\psi_3 :\ \text{goose}(\text{donald}) \wedge \text{quack}(\text{donald})$$
$$\psi_4 \quad \text{duck}(\text{donald})$$

While all of these solutions make the deduction valid, some of these solutions are more desirable than others. For example, ψ_1 contradicts known facts and is therefore a useless solution. On the other hand, ψ_2 simply restates the desired conclusion, and despite making the deduction valid, gets us no closer to explaining the observation. Finally, ψ_3 and ψ_4 neither contradict the premises nor restate the conclusion, but, intuitively, we prefer ψ_4 over ψ_3 because it makes fewer assumptions.

At a technical level, given premises Γ and desired conclusion φ, abduction is the problem of finding an explanatory hypothesis ψ such that:

$$(1)\quad \Gamma \wedge \psi \models \varphi$$
$$(2)\quad \Gamma \wedge \psi \models \text{false}$$

Here, the first condition states that ψ, together with known premises Γ, entails the desired conclusion φ, and the second condition stipulates that ψ is consistent with known premises. As illustrated by the previous example, there are many solutions to a given abductive inference problem, but the most desirable solutions are those that are as simple and as general as possible.

Recently, abductive inference has found many useful applications in verification, including inference of missing function preconditions [94, 95], diagnosis of error reports produced by verification tools [96], and for computing under-approximations [97]. Furthermore, abductive inference has also been used for inferring specifications of library functions [98] and for automatically synthesizing circular compositional proofs of program correctness [99].

In the context of the Concord project, our goal is to utilize abduction to identify the *smallest* and *most general* annotations required to verify a program. Assume that everything a static analysis could automatically learn about a program is encoded in constraint ψ and we are trying to prove a property encoded in constraint φ. By definition, if our tool reports a potential error, it must be that

$$\psi \models \varphi$$

Therefore finding a smallest root cause of the error reported can be directly mapped into logical abduction

$$(1)\ \Gamma \wedge \psi \models \varphi$$
$$(2)\ \Gamma \wedge \psi \models \text{false}$$

where ψ is a smallest piece of information that, if true and annotated by the user, will make the original condition provable. Of course the smallest such fact may not actually be true, therefore we need to generate a sequence of abductive solutions of increasing difficulty until the programmer using the tool confirms one of them by placing the right annotations.

8.1.2 Algorithm for Performing Abductive Inference

In this section, we describe the algorithm used in for performing abductive inference at a high level. First, let us observe that the entailment $\Gamma \wedge \psi \models \varphi$ can be rewritten as $\psi \models \Gamma \Rightarrow \varphi$. Furthermore, in addition to entailing $\Gamma \Rightarrow \varphi$, we want ψ to obey the following three requirements:

1. The solution ψ should be consistent with Γ because an explanation that contradicts known premises is not useful
2. To ensure the simplicity of the explanation, ψ should contain as few variables as possible
3. To capture the generality of the abductive explanation, ψ should be no stronger than any other solution ψ' satisfying the first two requirements

Now, consider a *minimum satisfying assignment* (MSA) of $\Gamma \Rightarrow \varphi$. An MSA of a formula ϕ is a partial satisfying assignment of ϕ that contains as few variables as possible. The formal definition of MSAs as well as an algorithm for computing them are given in [100]. Clearly, an MSA σ of $\Gamma \Rightarrow \varphi$ entails $\Gamma \Rightarrow \varphi$ and satisfies condition (2). Unfortunately, an MSA of $\Gamma \Rightarrow \varphi$ does not satisfy condition (3), as it is a logically strongest solution containing a given set of variables.

```
abduce(φ, Γ) {
1.   ϕ = (Γ ⇒ φ)
2.   Set X = find_mus(ϕ, Γ, free(ϕ), 0)
3.   χ = elim(∀X.ϕ)
4.   ψ = simplify(χ, Γ)
5.   return ψ
}

find_mus(ϕ, Γ, V, L) {
6.   If V = 0′ or |V| ≤ L   return  0′

7.   U = free(ϕ) − V
8.   if( UNSAT (Γ ∧ ∀U.ϕ))   return ∅

9.   Set best = 0′
10.   choose x ∈ V

11.   if(SAT(∀x.ϕ)) {
12.     Set Y = find_mus(∀x.ϕ, Γ, V ∖ {x}, L − 1);
13.     If (|Y| + 1 > L) { best = Y ∪ {x}; L = |Y| + 1 }
      }
14.   Set Y = find_mus(ϕ, Γ, V ∖ {x}, L);
15.   If (|Y| > L) { best = Y }

16.   return best;
}
```

Figure 9: Algorithm for performing abduction

Given an MSA of $\Gamma \Rightarrow \varphi$ containing variables V, we observe that a logically weakest solution containing only V is equivalent to $\forall V.\ (\Gamma \Rightarrow \varphi)$, where $V = \text{free}(\Gamma \Rightarrow \varphi) - V$. Hence, given an MSA of $\Gamma \Rightarrow \varphi$ consistent with Γ, an abductive solution satisfying all conditions (1)-(3) can be obtained by applying quantifier elimination to $\forall V.\ (\Gamma \Rightarrow \varphi)$.

Thus, to solve the abduction problem, what we want is a largest set of variables X such that $(\forall X.(\Gamma \Rightarrow \varphi)) \wedge \Gamma$ is satisfiable. We call such a set of variables X a *maximum universal subset* (MUS) of $\Gamma \Rightarrow \varphi$ with respect to Γ. Given an MUS X of $\Gamma \Rightarrow \varphi$ with respect to Γ, the desired solution to the abductive inference problem is obtained by eliminating quantifiers from $\forall X.(\Gamma \Rightarrow \varphi)$ and then simplifying the resulting formula with respect to Γ using the algorithm from [101].

Pseudo-code for our algorithm for solving an abductive inference problem defined by premises Γ and conclusion φ is shown in Figure 8.1. The abduce function given in lines 1-5 first computes an MUS of $\Gamma \Rightarrow \varphi$ with respect to Γ using the helper find_mus function. Given such a maximum universal subset X, we obtain a quantifier-free abductive solution χ by applying quantifier elimination to the formula $\forall X.(\Gamma \Rightarrow \varphi)$. Finally, at line 4, to ensure that the final abductive solution does not contain redundant sub-parts that are implied by the premises, we apply the simplification algorithm from [101] to χ. This yields our final abductive solution ψ which satisfies our criteria of minimality and generality and that is not redundant with respect to the original premises.

LOC analyzed	11,678
Analysis Time	143s
Number of lines changed	76
Annotations placed	7

Figure 10: Statistics on the OpenSSH analysis

The function find_mus used in abduce is shown in lines 6-16 of Figure 8.1. This algorithm is based directly on the find_mus algorithm we presented earlier in [100] but excludes universal subsets that contradict Γ. At every recursive invocation, find_mus picks a variable x from the set of free variables in ϕ. It then recursively invokes find_mus to compute the sizes of the universal subsets with and without x and returns the larger universal subset. In this algorithm, L is a lower bound on the size of the MUS and is used to terminate search branches that cannot improve upon an existing solution. Therefore, the search for an MUS terminates if we either cannot improve upon an existing solution L, or the universal subset U at line 7 is no longer consistent with Γ. The return value of find_mus is therefore a largest set X of variables for which $\Gamma \wedge \forall X.\phi$ is satisfiable.

8.1.3 Using Abduction to Identify Imprecision Root Causes on Open SSH

In this work, we created a self-contained version of OpenSSH that can be verified automatically, and used abductive inference to identify the missing pieces of information that we needed to add. This can range from simple additional assumptions to rewrites of code pieces to stubbing the behavior of sub-components. Table 8.2 gives a high-level overview of the changes needed for our tool to establish absence of null pointer dereference errors as well as array/buffer overflow or underflow errors.

As mentioned in the last section, the first key challenge is to identify possible small and relevant root causes from a spurious error report. All changes and annotations were identified using logical abduction, and we found this approach to be critical for any non-expert in identifying and remedying relevant precision losses.

To give the reader an understanding of how abduction helps programmers identify relevant annotations for error reports, consider the following sliced and condensed excerpt from OpenSSH. Observe that for keeping this example concise, we manually added one safety property that we want to prove, marked with static_assert on line 358 (In our full analysis, all necessary such checks are synthesized automatically).

```
1 /* Fatal messages.  This function never returns. */
2 void
3 fatal(const char *fmt,...)
4 {
5   exit(1);
6 }
7
8 void *
9 xmalloc(size_t size)
```

```
10 {
11   void * ptr ;
12
13   if ( size == 0)
14     fatal("xmalloc: zerosize");
15   ptr = malloc(size);
16   if (ptr == NULL)
17     fatal("xmalloc:uout of memory (allocating %lu bytes)", (u_long)
         size);
18   return ptr;
19 }
20
21 typedef struct {
22   unsigned int num_host_key_files;     /*  Number of files for host keys.  */
23   int      rhosts_rsa_authentication;  /*  If true, permit rhosts RSA
24                                         * authentication .  */
25   int      rsa_authentication;     /* If true, permit RSA authentication */
26   int challenge_response_authentication;
27   int password_authentication;
28 } ServerOptions;
29
30
31
32
33
34 int errno;
35 /*  import  */
36 extern ServerOptions options;
37 extern char *__progname;
38 extern uid_t original_ real_ uid ;
39 extern uid_t original_ effective_ uid;
40 extern pid_t proxy_ command_ pid ;
41
42
43
44 void *
45 xcalloc(size_t nmemb, size_t size)
46 {
47   void * ptr ;
48
49   if ( size == 0 || nmemb == 0)
50     fatal("xcalloc:uzerousize");
51   if ( SIZE_ T_ MAX / nmemb < size)
52     fatal("xcalloc:unmembu*usizeu>uSIZE_T_MAX");
53   ptr = calloc ( nmemb , size );
```

```
54    if (ptr == NULL)
55      fatal("xcalloc:uout of memory (allocating %lu bytes)",
56            (u_long)(size * nmemb));
57            return ptr;
58  }
59
60  void
61  xfree(void *ptr)
62  {
63    if (ptr == NULL)
64      fatal("xfree: NULL pointer given as argument");
65    free(ptr);
66  }
67
68
69
70
71  static Authctxt *authctxt;
72  /* message to be displayed after login */
73  Buffer loginmsg;
74
75  /*
76  * Any really sensitive data in the application is contained in this
77  * structure. The idea is that this structure could be locked into memory so
78  * that the pages do not get written into swap. However, there are some
79  * problems. The private key contains BIGNUMs, and we do not (in principle)
80  * have access to the internals of them, and locking just the structure is
81  * not very useful.      Currently, memory locking is not implemented.
82  */
83  struct {
84    Key    *server_key;     /* ephemeral server key */
85    Key    *ssh1_host_key; /* ssh1 host key */
86    Key    **host_keys;     /* all private host keys */
87    int    have_ssh1_key;
88    int    have_ssh2_key;
89    u_char         ssh1_cookie[SSH_SESSION_KEY_LENGTH];
90  } sensitive_data;
91
92  void
93  mm_ssh1_session_id(u_char session_id[16])
94  {
95    Buffer m;
96    int i;
```

```
97
98    debug3("%s entering", __func__);
99
100   buffer_init(&m);

101   for (i = 0; i < 16; i++)
102     buffer_put_char(&m, session_id[i]);
103
104   mm_request_send(&m);
105   buffer_free(&m);
106 }
107
108
109 /*
110 * Decrypt session_ key_ int using our private server and host key
111 * (key with larger modulus first).
112 */
113 int
114 ssh1_session_key(BIGNUM *session_key_int)
115 {
116   int rsafail = 0;
117
118   if (BN_cmp(sensitive_data.server_key->rsa->n,
119     sensitive_data.ssh1_host_key->rsa->n) > 0) {
120     /* Server key has bigger modulus. */
121     if (BN_num_bits(sensitive_data.server_key->rsa->n) <
122       BN_num_bits(sensitive_data.ssh1_host_key->rsa->n) +
123       SSH_KEY_BITS_RESERVED) {
124       fatal("do_connection:u%s:u"
125       "server_key %d <uhost_key %d +uSSH_KEY_BITS_RESERVED %d",
126       get_remote_ipaddr(),
127       BN_num_bits(sensitive_data.server_key->rsa->n),
128       BN_num_bits(sensitive_data.ssh1_host_key->rsa->n),
129       SSH_KEY_BITS_RESERVED);
130     }
131     if (rsa_private_decrypt(session_key_int, session_key_int,
132       sensitive_data.server_key->rsa) <= 0)
133       rsafail++;
134     if (rsa_private_decrypt(session_key_int, session_key_int,
135       sensitive_data.ssh1_host_key->rsa) <= 0)
136       rsafail++;
137   } else {
138     /* Host key has bigger modulus (or they are equal). */
139     if (BN_num_bits(sensitive_data.ssh1_host_key->rsa->n) <
140       BN_num_bits(sensitive_data.server_key->rsa->n) +
141       SSH_KEY_BITS_RESERVED) {
142       fatal("do_connection:u%s:u"
```

```
143         "host_keyu%du<userver_keyu%du+uSSH_KEY_BITS_RESERVEDu%d",
144         get_remote_ipaddr(),
145         BN_num_bits(sensitive_data.ssh1_host_key->rsa->n),
146         BN_num_bits(sensitive_data.server_key->rsa->n),
147         SSH_KEY_BITS_RESERVED);
148       }
149       if (rsa_private_decrypt(session_key_int, session_key_int,
150         sensitive_data.ssh1_host_key->rsa) < 0)
151         rsafail++;
152       if (rsa_private_decrypt(session_key_int, session_key_int,
153         sensitive_data.server_key->rsa) < 0)
154         rsafail++;
155     }
156     return (rsafail);
157 }
158
159
160 /* session identifier, used by RSA-auth */
161 u_char session_id[16];
162 /* variables used for privilege separation */
163 int use_privsep;
164
165 /* Destroy the host and server keys. They will no longer be needed. */
166 void
167 destroy_sensitive_data(void)
168 {
169     int i;
170
171     if (sensitive_data.server_key) {
172       key_free(sensitive_data.server_key);
173       sensitive_data.server_key = NULL;
174     }
175     for (i = 0; i < options.num_host_key_files; i++) {
176       if (sensitive_data.host_keys[i]) {
177         key_free(sensitive_data.host_keys[i]);
178         sensitive_data.host_keys[i] = NULL;
179       }
180     }
181     sensitive_data.ssh1_host_key = NULL;
182     memset(sensitive_data.ssh1_cookie, 0, SSH_SESSION_KEY_LENGTH);
183 }
184
185 /*
186 * SSH1 key exchange
187 */
188 static void
```

```
189 do_ssh1_kex(void)
190 {
191   int i, len;
192   int rsafail = 0;
193   BIGNUM *session_key_int;
194   u_char session_key[SSH_SESSION_KEY_LENGTH];
195   u_char cookie[8];
196   u_int cipher_type, auth_mask, protocol_flags;
197
198   /*
199   * Generate check bytes that the client must send back in the user
200   * packet in order for it to be accepted; this is used to defy ip
201   * spoofing attacks.       Note that this only works against somebody
202   * doing IP spoofing from a remote machine; any machine on the local
203   * network can still see outgoing packets and catch the random
204   * cookie.     This only affects rhosts authentication, and this is one
205   * of the reasons why it is inherently insecure.
206   */
207   arc4random_buf(cookie, sizeof(cookie));
208
209   /*
210   * Send our public key.      We include in the packet 64 bits of random
211   * data that must be matched in the reply in order to prevent IP
212   * spoofing.
213   */
214   packet_start(SSH_SMSG_PUBLIC_KEY);
215   for (i = 0; i < 8; i++)
216     packet_put_char(cookie[i]);
217
218   /* Store our public server RSA key. */
219   packet_put_int(BN_num_bits(sensitive_data.server_key->rsa->n));
220   packet_put_bignum(sensitive_data.server_key->rsa->e);
221   packet_put_bignum(sensitive_data.server_key->rsa->n);
222
223   /* Store our public host RSA key. */
224   packet_put_int(BN_num_bits(sensitive_data.ssh1_host_key->rsa->n));
225   packet_put_bignum(sensitive_data.ssh1_host_key->rsa->e);
226   packet_put_bignum(sensitive_data.ssh1_host_key->rsa->n);
227
228   /* Put protocol flags. */
229   packet_put_int(SSH_PROTOFLAG_HOST_IN_FWD_OPEN);
230
231   /* Declare which ciphers we support. */
232   packet_put_int(cipher_mask_ssh1(0));
233
234   /* Declare supported authentication types. */
235   auth_ mask = 0;
```

```
236   if(options.rhosts_rsa_authentication)
237     auth_mask |= 1 << SSH_AUTH_RHOSTS_RSA;
238   if (options.rsa_authentication)
239     auth_mask |= 1 << SSH_AUTH_RSA;
240   if(options.challenge_response_authentication == 1)
241     auth_mask |= 1 << SSH_AUTH_TIS;
242   if (options.password_authentication)
243     auth_mask |= 1 << SSH_AUTH_PASSWORD;
244   packet_put_int(auth_mask);
245
246   /* Send the packet and wait for it to be sent. */
247   packet_send();
248   packet_write_wait();
249
250   debug("Sentu%dubituserverukeyuandu%dubituhostukey.",
251   BN_num_bits(sensitive_data.server_key->rsa->n),
252   BN_num_bits(sensitive_data.ssh1_host_key->rsa->n));
253
254   /* Read clients reply (cipher type and session key).          */
255   packet_read_expect(SSH_CMSG_SESSION_KEY);
256
257   /* Get cipher type and check whether we accept this.          */
258   cipher_type = packet_get_char();
259
260   if (!(cipher_mask_ssh1(0) & (1 << cipher_type)))
261     packet_disconnect("Warning:uclientuselectsuunsupporteducipher.");
262
263   /* Get check bytes from the packet.      These must match those we
264   sent earlier with the public key packet. */
265   for (i = 0; i < 8; i++)
266     if (cookie[i] != packet_get_char())
267       packet_disconnect("IP Spoofing check bytes do not match.");
268
269     debug("Encryption type: %.200s", cipher_name(cipher_type));
270
271   /* Get the encrypted integer. */
272   if ((session_key_int = BN_new()) == NULL)
273     fatal("do_ssh1_kex: BN_new failed");
274   packet_get_bignum(session_key_int);
275
276   protocol_flags = packet_get_int();
277   packet_set_protocol_flags(protocol_flags);
278   packet_check_eom();
279
280   /* Decrypt session_key_int using host/server keys */
281   rsafail = PRIVSEP(ssh1_session_key(session_key_int));
282
```

```
283     /*
284     * Extract session key from the decrypted integer. The key is in the
285     * least significant 256 bits of the integer; the first byte of the
286     * key is in the highest bits.
287     */
288     if(!rsafail) {
289        (void) BN_mask_bits(session_key_int, sizeof(session_key) * 8);
290       len = BN_num_bytes(session_key_int);
291       if (len < 0 || (u_int)len > sizeof(session_key)) {
292       error("do_ssh1_kex: bad session key lenufrom %s: "
293         "session_key_int %d > sizeof(session_key) %l ",
294         get_remote_ipaddr(), len, (u_long)sizeof(session_key));
295         rsafail++;
296       } else {
298         BN_bn2bin(session_key_int,

299         session_key + sizeof(session_key) - len);
300
301       derive_ssh1_session_id(
302                   sensitive_data.ssh1_host_key->rsa->n,
303                   sensitive_data.server_key->rsa

                                                  ->n,
304                                               cookie, session_id);
305                                               /*
306                                               * Xor the first 16 bytes of the
                                                     session key with the
307                                               * session id.
308                                               */
309                                               for (i = 0; i < 16; i++)
310                                                 session_key[i] ^= session_id[
                                                      i];
311       }
312     }
313     if (rsafail) {
314       int bytes = BN_num_bytes(session_key_int);
315       u_char *buf = xmalloc(bytes);
316       MD5_CTX md;
317
318       logit("do_connection: generating a fake encryption key");
319       BN_bn2bin(session_key_int, buf);
320       MD5_Init(&md);
321       MD5_Update(&md, buf, bytes);
322       MD5_Update(&md, sensitive_data.ssh1_cookie,
            SSH_SESSION_KEY_LENGTH);
323       MD5_Final(session_key, &md);
324       MD5_Init(&md);
```

```
325       MD5_Update(&md, session_key, 16);
326       MD5_Update(&md, buf, bytes);
327       MD5_Update(&md, sensitive_data.ssh1_cookie,
              SSH_SESSION_KEY_LENGTH);
328       MD5_Final(session_key + 16, &md);
329       memset(buf, 0, bytes);
330       xfree(buf);
331       for (i = 0; i < 16; i++)
332         session_id[i] = session_key[i] ^ session_key[i + 16];
333     }
334     /* Destroy the private and public keys. No longer. */
335     destroy_sensitive_data();
336
337     if (use_privsep)
338       mm_ssh1_session_id(session_id);
339
340     /* Destroy the decrypted integer.          It is no longer     needed. */
341     BN_clear_free(session_key_int);
342
343     /* Set the session key. From this on all communications will be encrypted . */
344     packet_set_encryption_key(session_key, SSH_SESSION_KEY_LENGTH,
            cipher_type);
345
346     /* Destroy our copy of the session key.          It is no longer needed. */
347     memset(session_key, 0, sizeof(session_key));
348
349     debug("Received session key; encryption turned on.");
350
351     /* Send an acknowledgment packet. Note that this packet is sent encrypted . */
352     packet_start(SSH_SMSG_SUCCESS);
353     packet_send();
354     packet_write_wait();
355
356
357     for (i = 0; i < options.num_host_key_files; i++) {
358       static_assert(sensitive_data.host_keys[i] == NULL);
359     }
```

Observe that even in this sliced and self-contained code fragment, it is far from clear why the analysis is reporting this error and what information is missing or where precision was lost. Using logical abduction, our tool automatically identifies the following fact:

" If function destroy_sensitive_data() sets every element of sensitive_data.host_keys in range[0, options.num_host_key_files] to NULL, this will prove the assertion."

More formally, we identify a simplest and most general solution to the abduction problem as:

$$\psi := \forall i.(0 \le i < options.num\_host\_key\_files \wedge \text{call}(destroy\_sensitive\_data)) \rightarrow (\text{array}(\text{sensitive\_data.host\_keys}, i) = 0)$$

While the desired property clearly holds for the loop in destroy_sensitive_data(), the verification tool, for internal reasons that are obscure to the tool user, fails to understand this loop properly. In this case, this is clearly the case, and after annotating this missing piece of information, the assertion is now provable. Specifically, we add the following simple annotation to line 176:

```
assume( (!(0<=_t && _t <= i)) || ( sensitive_data.host_keys[_t] == NULL));
```

where the syntax _t marks _t as a universally quantified variable.

Observe that logical abduction helped us immediately pin down the root cause of this spurious report without any need for the user to be familiar with the internal reasoning and limitations of the analysis tool used.

8.1.4 Code Changes and Annotations

In order to complete the verification of memory safety properties, we also added a few key annotations, as well as modified some lines of code in order to facility automated analysis of the code. This mostly involved expressing global object invariants as annotations, as well as replacing built-in macros that copy memory contents with explicit store statements.

Here is a code snippet from clientloop.c to illustrate this:

```
1
2  // Annotated global invariants
3  assume(options.num_send_env <= MAX_SEND_ENV);
4  assume(options.num_send_env >=0);
5  assume(env_len*sizeof(char*) <= buffer_size(env));
6
7  /* Transfer any environment variables from client to server */
8  if (options.num_send_env != 0 && env != NULL) {
9          int i, j, matched;
10         char * name , * val ;
11
12         debug("Sendinguenvironment.");
13         for (i = 0; i<env_len; i++) {
14                 // here we replaced pointer arithmetic used in
15                 // the original code with an explicit array reference style
16
17                 /* Split */
18                 name = env [i];
19
```

```
20                      if (( val = foo ( name , '=')) == NULL ) {
21                              xfree(name);
22                              continue;
23                      }
24
25                      *val++ = '\0';
26
27                      matched = 0;
28                      for (j = 0; j < options.num_send_env; j++) {
29                              if (match_pattern(name, options.send_env[
                                j]))
                                  {
30                                      matched = 1;
31                                      break;
32                              }
33                      }
34                      if (!matched) {
35                              debug3("Ignoreduenvu%s", name);
36                              xfree(name);
37                              continue;
38                      }
39
40                      debug("Sending env %s = %s", name, val);
41                      channel_request_start(id, "env", 0);
42                      packet_put_cstring(name);
43                      packet_put_cstring(val);
44                      packet_send();
45                      xfree(name);
46              }
47      }
```

Note that the key object invariants relating global values are annotated in order to allow the successful analysis of this code segment.

8.2 Dark Corners

Sound static program analysis promises exhaustive detection of many classes of program errors and, ideally, verification that the program is free of these errors. Despite decades of effort investing in developing powerful static analyses, we are still far from having a static analysis that can analyze existing programs precisely enough to make verifying the absence of important errors (such as memory safety vulnerabilities) feasible in practice. The critical issue is the lack of precision that such analyses inevitably encounter when analyzing the programs that occur in practice. This lack of precision causes either 1) unacceptable numbers of false positive alarms or 2) the use of unsound techniques that may leave errors uncovered.

Our hypothesis is that, in existing programs, only a small percentage of the code (the code's dark corners) is responsible for this lack of precision. If this hypothesis is true, it opens up the

possibility that we are much closer to a practical program analysis for verifying important security (and potentially other properties) than it currently appears. Our work on analyzing the Java system libraries in DroidSafe [1] supports this. In many cases, small changes to the library code significantly increased the precision of the analysis

We believe that once the small percentage is identified that there are reasonable techniques for addressing the complex code. For example: (1) Applying more expensive analysis techniques to this code; (2) Making manual changes to the code and/or adding annotations/dynamic checks to reduce the complexity; (3) Replacing the code with similar code from other projects (e.g., DARPA MUSE); (4) Providing an alternative implementation that is easier to analyze but can be shown to match the existing implementation (with respect to memory safety).

We investigated this hypothesis on a set of widely used open source C programs from coreutils. Specifically mv.c and chroot.c Our goal was to create versions of the programs for which we can verify memory safety (out of bounds accesses), null pointer dereferences, and the use of uninitialized variables.

We used a focused version of the Compass tool [102, 103, 92] (CONCORD) adapted and streamlined for robustness and coverage for this study.

8.2.1 Concord Features

Concord analyzes C programs and processes some additional functions to aid in the analysis. It can check assertions added by the programmer and can automatically check for buffer overflows, null pointer dereferences, and the use of uninitialized variables.

Concord supports a number of special functions. These are most commonly used in library routine summaries though there are use cases (primarily for static_assert) in the application as well.

The primary functions supported by Concord are:

- void static_assert (<expr>) - Statically checks the specified expression.
- void assume (<expr>) - Assume that the specified expression is true
- <type> set_nonnull_<type> - Function that returns a value that is non-null (non-zero) for the specified type. This function must be declared with the correct type. Concord does not operate correctly if there are type mismatches between the return value of set_nonnull and the variable it is assigned to.
- int unknown() - Returns an arbitrary initialized integer value
- void buffer_safe (void *buffer, int offset) - Statically checks whether or not it is safe to index into buffer at offset
- void assume_size (void *buffer, int len) - Assume that the size of buffer is len

8.2.2 Coding practices and analysis

We discovered a number of coding practices that make a significant difference to the analysis. In most cases, the alternative easier to analyze version is basically equivalent in terms of coding effort and efficiency.

8.2.2.1 Obscure loop iterations / indexing

Concord attempts to find invariants as part of analyzing loops. One of its main approaches is to attempt to find a relationship between the loop counter (K) and various loop variables. This works well when the loop is straightforward (such as a loop over an array of values), but may fail to provide interesting information or timeout when the relationship is obscure.

For example the C library function getopt() processes command line arguments. Each call to getopt() returns the next valid option character. A typical loop that invokes getopt() looks like:

```
1   while ((c = getopt (argc, argv,"bfint:uvS:T") != -1) {
2     // switch on option character
3     switch (c)
4     ...
5   }
```

To support this usage model, the internals of getopt() uses the external variable optind to keep track of the current argument and the static variable nextchar to keep track of the next argument and next option character within that argument. These variables are updated on each call. A greatly simplified version of getopt() shows roughly how this works:

```
1  int getopt ( int argc , char ** argv,  char *options) {
2  {
3    if ( nextchar == NULL ) || (* nextchar == '\0 ') {
4
5      if( optind == 0) optind = 1;
6          while ((optind < argc) && (argv[optind][0] != '-'))
7          optind++;
8      if ( optind >= argc ) return -1;
9      nextchar = argv [ optind ] + 1;
10   }
11   return (*nextchar++);
12 }
```

There is no straightforward relationship between the loop counter and any of the variables. In order to create a simple interface for the caller, getopt() turns what might naturally be two nested loops into a single loop which significantly complicates the analysis. In this case, Concord times out while attempting to simplify the loop.

If this were not a utility routine it could be implemented in a much more straightforward fashion as two nested loops. The first loop is over each of the arguments and then within each option

argument (those that begin with a dash), a loop over each option character in the argument. For example:

```
1   for ( i = 0; i < argc ; i ++) {
2     char * arg = argv [i];
3     if (* arg ++ == '-') {
4       while (*arg != 0) {
5         ch = * arg;
6         switch (c)
7         ...
```

This version works fine in Concord. (and is arguably just as easy to use as the original) But some of the complexity is no longer hidden in a library routine. This could be resolved by creating two library routines (one to loop through the arguments and the other to process an argument).

Another approach is to provide a simpler summary of the library method rather than analyzing the method itself. This has the downside of not proving correctness of the library routine, but can create a much more analyzable version for the application. In this case, we replaced getopt() by:

```
1  int getopt ( int argc , char ** argv ,   char *options) {
2
3    static_ assert ( argv != NULL);
4
5    // optind is guaranteed to point into argv
6    optind = unknown ();
7    assume (( optind >= 1 > && ( optind < argc ));
8
9    // optarg is a pointer to the value for an argument. An argument
10   // value may either immediately follow the option character
11   // or be in the next argument
12   optarg = unknown();
13
14   // Each of the strings in argv should be valid
15   int ii;
16   for ( ii = 0; ii < argc ; ii ++) {
17     buffer_safe (argv, ii);  // check that ii is a safe index
18     check_str_nn (argv[ii]);
19   }
20
21   // The return value is either a character or -1
22   int rval = unknown ();
23   assume ((rval == -1) || ((rval > 0)  && rval < 255));
24   return rval;
25 }
```

This code checks all of its arguments for validity and sets up reasonable return values. This allows the caller to be verified by Concord.

This version is also handled correctly by Concord and shows that there are no errors in the client (mv, chroot, etc.)

Unfortunately, the summary approach may miss some nuanced problems with the use of getopt() such as when there are arguments associated with options. The optarg value is only set when such an argument is encountered. The summary has to set optarg to a valid value on each call (because it does not know which calls will have an option value) It is possible that a caller may access optarg when it is not set which would lead to a memory error.

This problem can be handled when options are handled as two nested loops.

```
1   for ( i = 0; i < argc ; i ++) {
2     char * arg = argv [i];
3     if (* arg == '-') {
4       arg++;
5       while (*arg++) {
6         ch = * arg;
7         switch (c){
8         case 'b':
9           make_ backups = true;
10          break;
11        case 'f':
12          x.interactive = I_ALWAYS_YES;
13          break;
14        case 'i':
15          x.interactive = I_ASK_USER;
16          break;
17        case 'n':
18          x.interactive = I_ALWAYS_NO;
19          break;
20        case 't':
21          if (* arg ) {
22            target_ directory = arg ;
23          } else { // directory is in the next argument
24            target_directory = argv[i++]
25          }
26          // move arg pointer to the end of the argument to break the loop
27          arg += strlen(arg);
28          break;
29        case 'T':
30          no_ target_ directory = true;
31          break;
32        case 'u':
33          x.update = true;
34          break;
35        case 'v':
36          x.verbose = true;
37          break;
```

```
38          case 'S':
39            make_backups = true;
40           // backup_suffix_string = optarg;
41            break;
42          case_GETOPT_HELP_CHAR;
43          case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
44          default:
45            usage (EXIT_FAILURE);
46          }
47          ...
```

8.2.2.2 Additional Checks

Any static analysis can be confused by complex relationships between variables. It may be that at a particular point in the program, that a certain condition must be true and the program is safe to assume that. The static analysis may not be able to reason about this and may produce a false positive.

One simple solution to this problem is to add a redundant check for the condition where it is assumed. For example, getopts() handles command line options that take an argument. If the argument is missing, getopts() will, be default, issue an error message and terminate. The application can thus safely assume that the argument is not null. This correspondence would be extremely difficult to determine analyzing the getopt() code and impossible when using the summary.

However, it is easy to avoid the problem by simply adding an additional null check for the argument at its point of use. This was necessary only a few times in our example programs.

8.2.2.3 Pointer Arithmetic

Complex pointer arithmetic can be difficult to analyze and lead to inaccuracies in analysis. In many cases, this can be replaced by an array reference. This can make the relationship between the loop variable and the array reference more explicit.

This is the change that we made in clientloop.c to allow the analysis to remove false positives.

8.2.3 Verification Approach

We verified several coreutils programs for memory safety as part of the project. The verification concentrated on the main file of the program. Calls to methods in the standard C library and the coreutils libraries were not included. These methods were summarized with respect to memory safety. Each summary checks the validity of each of the arguments to the function and sets any return values to the correct range of valid values. For example, the summary of stat() is:

```
1 int stat (char const *file, struct stat *st) {
2
```

```
3     // Make sure arguments are not null
4     static_ assert ( file != NULL );
5     static_ assert ( st != NULL );
6
7     // Initialize fields to non-zero ints
8     st->st_dev  = set_nonnull_int();
9     st->st_ino  = set_nonnull_int();
10    st->st_mode = set_nonnull_int();
11
12    // Return normally or error
13    if (set_nonnull_int() == 1)
14      return 0;
15    else { // simulated error
16      errno = set_nonnull_int();
17      return -1;
18    }
19  }
```

Note that this initializes fields to non-zero values. This ensures that accesses to these field will not result in uninitialized read errors.

8.2.4 Concord limitations

While we addressed a number of limitations in Concord, there is additional work that would allow a wider range of programs to be verified and ease the work of doing so. Some of these are listed below.

Valid Strings. Null terminated strings are a very basic type in C. It is difficult to define these efficiently using the building blocks currently available in Concord. A valid null-terminated string should consist of an array of initialized characters followed by the null character. The array of characters (including the null) should be less than or equal to the size of the buffer that contains them. In concord, such a check could be coded something like:

```
1  void valid_ string ( char * str ) {
2    int i;
3    for (int i = 0; str[i] != '\0'; i++)
4      char ch = str [i];
5    static_assert (buffer_size (str) > i);
6  }
```

Unfortunately, it is not as easy to specify that an unknown string (such as one read from a command line argument or a file) is a valid one. And Concord doesn't currently have a mechanism to assign an arbitrary size to a buffer (such as might be returned from getenv())

Given the prevalence of strings and string manipulations in C, it seems worthwhile to support them directly in Concord. This could be accomplished by adding some new functions. The function

check_str() would check for a valid string, the function set_str() would mark the string as valid or NULL, and the function set_str_nonnull() would mark the strings as valid and not null. This would allow functions that accept strings as arguments to perform checks on their inputs and return valid outputs. It would also make it straightforward to provide summaries for the standard C string functions.

Object invariants. Object invariants are constraints on an object that should be true at entry/exit to all of the public methods of the object. The valid string checks described above are a special case of these. Object invariants can simplify static checking by providing assumptions that will always hold over an object of a particular type. If Concord were to be enhanced with sup- port for object invariants (possibly over C structures), it would be easier to verify more complex programs.

Assumes. Currently Concord does not propagate information about variables specified by assumes (or implied by conditional statements) as cleanly as it propagates sets of possible values. This can lead to false positives when the necessary information to verify a property is available but not fully propagated.

Loop Invariants. Concord attempts to learn relationships between the iterations of a loop and any variables that are manipulated in the loop. Allowing loop invariants to be specified would make it possible to handle more complex loops.

Varargs. Concord doesn't currently support variable argument lists, but this would be any easy enhancement.

Memory Management Checks. With respect to memory safety, Concord checks for out-of-bounds buffer accesses, reading uninitialized values, and null dereferences. It could be enhanced to check for common memory management errors such as Memory leaks, double frees, etc.

Non-deterministic. Due to low level implementation choices, some of the basic data structures used within Concord do not yield deterministic results (one run may timeout or show false positives while a subsequent run does not). It is a straightforward fix to change the underlying sets, lists, and maps to have a repeatable order.

8.2.5 Debugging

Debugging false positives (and differentiating between false positives and real problems) when verifying code can be challenging to the non-expert. However, as part of this project we discovered a relatively straightforward process for quickly finding the source of problems.

In some sense, debugging a verification error is very much like debugging a normal coding error. The static analysis indicates that a particular operation is not necessarily correct (e.g., it is dereferencing a possibly null variable or reading an uninitialized variable). If the assertion were checked at run-time, it would fail in a very similar fashion. A standard debugging technique would be to add debug statements to earlier points in the program to determine where the unexpected value came from.

A very similar approach can be taken with Concord. One can add assertions earlier in the data-flow of the variable making the same check. Iteratively re-running the analysis and adding additional assertions can quickly narrow down the problem (perhaps using a rough binary search). Once the root cause of the problem is found, it is usually pretty straightforward to fix.

This basic approach can be applied more generically as well. It is useful to add assertions at standard program points (such as function entry and exit points). They can help make the assumptions of the function and its results more clear. The systematic presence of such asserts will often uncover problems much closer to the source. We found that to be the case when working with our library summaries. Calls to the libraries often triggered failures quite close to the actual problem that would otherwise have appeared much later.

Not surprisingly, this approach works best when dealing with false positives. It is much less helpful in debugging problems that cause timeouts in the solver.

9.0 CONCLUSION

Application marketplaces provide a centralized location for application developers to place applications for potential users. A critical weakness with current application marketplaces, however, is that users have no way to be sure that the applications in the marketplace are free of malware. The potential presence of malicious malware and the resulting possibility of widespread security vulnerabilities can even eliminate the ability of service organizations (such as the United States Department of Defense) with stringent security needs to use application marketplaces.

The DroidSafe project developed effective program analysis techniques and tools to uncover malicious code in Android mobile applications. The core of the system is a static information flow analysis that reports the context under which sensitive information is used. The DroidSafe project invested significant time developing a comprehensive semantic model of Android run-time behaviors alongside the analysis to achieve acceptable precision, accuracy, and scalability for real-world Android applications. To address subtle functional correctness bugs and vulnerabilities, we have produced a formal model of (part of) the Android system capable of supporting proofs of functional correctness of simple but non-trivial Android apps.

The combined system has been demonstrated to be the most precise and accurate information flow analysis for Android applications. The analysis results can be used to automatically check applications for security policy violations, and the results can help a human analyst inspect sensitive behaviors of an app, increasing accuracy and throughput of application vetting. For each of the last six APAC engagements, the DroidSafe team was unsurpassed in malware diagnosis accuracy and human-analysis diagnosis throughput.

We also explored how to move precise static analysis and verification techniques from specialized research tools to an approach that can feasibly be adopted by programmers in the real world. Our hypothesis is that, in existing programs, only a small percentage of the code (the code's dark corners) is responsible for this lack of precision. We found that in many cases relatively straightforward code modifications can yield code that is analyzable and still practical.

10.0 REFERENCES

[1] Gordon, Michael I, Kim, Deokhwan, Perkins, Jeff, Gilham, Limei, Nguyen, Nguyen, and Rinard, Martin, "Information-flow analysis of Android applications in DroidSafe," in *Network and Distributed System Security (NDSS)*

[2] Smith, Eric and Coglio, Alessandro, "Android Platform Modeling and Android App Verification in the ACL2 Theorem Prover," in *7th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*

[3] Chen, Kevin Zhijie, Johnson, Noah, Silva, Vijay D, Dai, Shuaifu, Macnamara, Kyle, Magrino, Tom, Wu, Edward, Rinard, Martin, and Song, Dawn, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *NDSS*

[4] Petters, Dmitrij, **Efficient Resolution of Security-Sensitive Values in Android using Abstract Interpretation**, Master's thesis, Massachusetts Institute of Technology, 2014

[5] Rubin, Julia, Gordon, Michael I., Nguyen, Nguyen, and Rinard, Martin, *Non-Essential Communication in Mobile Applications*, Technical Report MIT-CSAIL-TR-2015-015, Massachusetts Institute of Technology, 2015

[6] Rubin, Julia, Gordon, Michael I., Nguyen, Nguyen, and Rinard, Martin, "Covert Communication in Mobile Applications (T)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, Washington, DC, USA, ASE '15, pp. 647–657, URL http://dx.doi.org/10.1109/ASE.2015.66

[7] Felt, Adrienne Porter, Finifter, Matthew, Chin, Erika, Hanna, Steven, and Wagner, David, "A survey of mobile malware in the wild," *Security*, **55**, June 2011, p. 3

[8] Grace, Michael C, Zhou, Wu, Jiang, Xuxian, and Sadeghi, Ahmad-Reza, "Unsafe exposure analysis of mobile in-app advertisements," in *WISEC*

[9] Felt, Adrienne Porter, Chin, Erika, Hanna, Steve, Song, Dawn, and Wagner, David, "Android Permissions Demystified," *CCS*, 2011

[10] Percoco, N. J. and Schulte, S., "Adventures in Bouncerland," , 2012

[11] Enck, William, Gilbert, Peter, Chun, BG, and Cox, LP, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," in *OSDI*

[12] Reina, Alessandro, Fattori, Aristide, and Cavallaro, Lorenzo, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," in *EuroSec*

[13] Lu, Long, Li, Zhichun, Wu, Zhenyu, Lee, Wenke, and Jiang, Guofei, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *CCS*

[14] Arzt, Steven, Rasthofer, Siegfried, Fritz, Christian, Bodden, Eric, Bartel, Alexandre, Klein, Jacques, Le Traon, Yves, Octeau, Damien, and McDaniel, Patrick, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *PLDI*

[15] Kim, Jinyung, Yoon, Yongho, Yi, Kwangkeun, and Shin, Junbum, "Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *MoST*

[16] Chin, Erika, Felt, AP, Greenwood, Kate, and Wagner, David, "Analyzing inter-application communication in Android," in *MobiSys*

[17] Fuchs, Adam P, Chaudhuri, Avik, and Foster, Jeffrey S, *ScanDroid: Automated Security Certification of Android Applications*, Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, MD, November 2009

[18] Yang, Zhemin and Yang, Min, "Leakminer: Detect information leakage on android with static taint analysis," in *WCSE*, p. 104

[19] Gibler, Clint, Crussell, Jonathan, Erickson, Jeremy, and Chen, Hao, "AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale," *Trust and Trustworthy Computing*, 2012

[20] Li, Li, Bartel, Alexandre, Klein, Jacques, Traon, Yves Le, Arzt, Steven, Rasthofer, Siegfried, Bodden, Eric, Octeau, Damien, and McDaniel, Patrick, "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," *CoRR*, 2014

[21] Google, "Android Open Source Project," URL https://source.android.com/

[22] Sridharan, Manu, Artzi, Shay, Pistoia, Marco, Guarnieri, Salvatore, Tripp, Omer, and Berg, Ryan, "F4F: taint analysis of framework-based web applications," in *OOPSLA*

[23] Smaragdakis, Yannis, Bravenboer, Martin, and Lhoták, Ondrej, "Pick Your Contexts Well: Understanding Object-Sensitivity," in *POPL*

[24] Smaragdakis, Y, Kastrinis, George, and Balatsouras, George, "Introspective analysis: context-sensitivity, across the board," in *PLDI*

[25] Google, "Intent and Intent Filters," URL http://developer.android.com/guide/ components/intents-filters.html

[26] King, Dave, Hicks, Boniface, Hicks, Michael, and Jaeger, Trent, "Implicit flows: Can't live with 'Em, can't live without 'Em," in *ICISS*

[27] Smaragdakis, Yannis, Kastrinis, George, Balatsouras, George, and Bravenboer, Martin, *More Sound Static Handling of Java Reflection*, Technical report, 2014

[28] Livshits, Benjamin, Whaley, John, and Lam, Monica S, "Reflection Analysis for Java," in *APLAS*

[29] Rasthofer, Siegfried, Arzt, Steven, and Bodden, Eric, "A machine-learning approach for classifying and categorizing android sources and sinks," *NDSS*, 2014

[30] Andersen, Lars O., **Program Analysis and Specialization for the C Programming Language**, Ph.D. thesis, U. of Copenhagen, 1994

[31] Bravenboer, Martin and Smaragdakis, Yannis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA*

[32] Lhotak, Ondrej, **Program analysis using binary decision diagrams**, Ph.D. thesis, McGill University, Montreal, 2006

[33] Sridharan, Manu, Chandra, Satish, Dolby, Julian, Fink, Stephen J., and Yahav, Eran, **Aliasing in Object-Oriented Programming**, Springer Berlin Heidelberg, 2000

[34] Berndl, Marc, Lhoták, Ondrej, Qian, Feng, Hendren, Laurie, and Umanee, Navindra, "Points-to analysis using BDDs," *PLDI*, 2003

[35] Lhotak, Ondrej, **SPARK: A Flexible Points-To Analysis Framework for Java**, Ph.D. thesis, McGill University, Montreal, 2002

[36] Kodumal, John and Aiken, Alex, "Banshee: A scalable constraint-based analysis toolkit," in *SAS*

[37] Christensen, Aske Simon, Møller, Anders, and Schwartzbach, Michael I., "Precise Analysis of String Expressions Static Analysis," in *SAS*

[38] IBM, "IBM Security AppScan," URL http://www-03.ibm.com/software/products/de/appscan

[39] HP, "Enterprise Security Intelligence," URL http://www8.hp.com/us/en/ software-solutions/enterprise-security.html

[40] Vallée-Rai, R, Gagnon, Etienne, and Hendren, Laurie, "Optimizing Java bytecode using the Soot framework: Is it feasible?" *CC*, 2000

[41] Octeau, Damien, McDaniel, Patrick, Jha, Somesh, Bartel, Alexandre, Bodden, Eric, Klein, Jacques, and Le Traon, Yves, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Usenix Security*, Washington D.C., USA

[42] Jordan, Alex, Gladd, Alex, and Abramov, Alex, *Android Malware Survey*, Technical Report April, Raytheon BBN Technologies, 2012

[43] Liang, Percy and Naik, Mayur, "Scaling abstraction refinement via pruning," *ACM SIGPLAN Notices*, **47**, 6, 2012, p. 590

[44] Kastrinis, George and Smaragdakis, Y, "Hybrid Context-Sensitivity for Points-To Analysis," in *PLDI*

[45] Guyer, SZ and Lin, Calvin, "Client-driven pointer analysis," *SAS*, 2003

[46] Livshits, V Benjamin and Lam, Monica S, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *USENIX Security*

[47] Tripp, Omer, Pistoia, Marco, Fink, Stephen J, Sridharan, Manu, and Weisman, Omri, "TAJ: Effective Taint Analysis of Web Applications," in *PLDI*

[48] Klieber, William, Flynn, Lori, Bhosale, Amar, Jia, Limin, and Bauer, Lujo, "Android taint flow analysis for app sets," in *SOAP*

[49] Tripp, Omer and Rubin, Julia, "A Bayesian Approach to Privacy Enforcement in Smartphones," in *USENIX Security*

[50] Livshits, Benjamin, Nori, Aditya V, Rajamani, Sriram K, and Banerjee, Anindya, "Merlin: Specification Inference for Explicit Information Flow Problems," in *PLDI*

[51] Open Handset Alliance, "Android Open Source Project," http://source.android.com

[52] Felt, Adrienne Porter, Finifter, Matthew, Chin, Erika, Hanna, Steve, and Wagner, David, "A Survey of Mobile Malware in the Wild," in *Proc. ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*

[53] Vidas, Timothy, Votipka, Daniel, and Christin, Nicolas, "All Your Droid Are Belong To Us: A Survey of Current Android Attacks," in *Proc. 5th USENIX Workshop on Offensive Technologies (WOOT)*

[54] Enck, William, Ongtang, Machigar, and McDaniel, Patrick, "Understanding Android Security," *IEEE Security & Privacy Magazine*, **7**, 1, 2009

[55] Shabtai, Asaf, Fledel, Yuval, Kanonov, Uri, Elovici, Yuval, Dolev, Shlomi, and Glezer, Chanan, "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy Magazine*, **8**, 2, 2010

[56] Armando, Alessandro, Merlo, Alessio, Migliardi, Mauro, and Verderame, Luca, "Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures)," in Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, Springer, volume 376 of *IFIP Advances in Information and Communication Technology*, 2012

[57] Lineberry, Anthony, Richardson, David Luke, and Wyatt, Tim, "These Aren't the Permissions You're Looking for," DEF CON 18, 2010

[58] University of Texas at Austin, "The ACL2 Theorem Prover," http://www.cs.utexas.edu/~moore/acl2

[59] Open Handset Alliance, "Android Development Resources," http://developer.android.com

[60] Lindholm, Tim, Yellin, Frank, Bracha, Gilad, and Buckley, Alex, **The Java Virtual Machine Specification – Java SE 8 Edition**, March 2014, http://docs.oracle.com/javase/specs/jvms/se8/html

[61] Gordon, Michael I. Kim, Deokhwan, Perkins, Jeff, Gilham, Limei, Nguyen, Nguyen, and Rinard, Martin, "Information-Flow Analysis of Android Applications in DroidSafe," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS)*

[62] Boyer, Robert S. and Moore, J Strother, **A Computational Logic**, Academic Press, 1979

[63] Smith, Eric W. **Axe: An Automated Formal Equivalence Checking Tool for Programs**, Ph.D. dissertation, Stanford University, 2011

[64] Moore, J, "Proving Theorems about Java and the JVM with ACL2," http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html

[65] McCarthy, John, *A Formal Description of a Subset of Algol*, Technical Report Stanford Artificial Intelligence Project Memo No. 24, Stanford University, 1964

[66] Manolios, Panagiotis and Moore, J Strother, "Partial Functions in ACL2," *Journal of Automated Reasoning*, **31**, p. 2003

[67] DARPA Information Innovation Office, "Automated Program Analysis for Cybersecurity (APAC) Program," http://www.darpa.mil/program/ automated-program-analysis-for-cybersecurity

[68] Milner, Robin, *An Algebraic Definition of Simulation between Programs*, Technical Report CS-205, Stanford University, 1971

[69] Hoare, C. A. R., "Proof of Correctness of Data Representations," *Acta Informatica*, **1**, 4, 1972, pp. 271–281

[70] Masoumeh Al. Haghighi Mobarhan, **Formal Specification of Selected Android Core Applications and Library Functions**, Master's thesis, Chalmers University of Technology, University of Gothenburg, 2011

[71] "The Java Modeling Language (JML)," http://jmlspecs.org

[72] "The KeY Project," http://www.key-project.org

[73] Harel, David, Kozen, Dexter, and Tiuryn, Jerzy, **Dynamic Logic**, MIT Press, 2000

[74] Jeon, Jinseong, Micinski, Kristopher, and Foster, Jeffrey, *SymDroid: Symbolic Execution for Dalvik Bytecode*, Technical Report CS-TR-5022, University of Maryland, College Park, 2012

[75] Payet, Etienne and Spoto, Fausto, "An Operational Semantics for Android Activities," in *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*

[76] Armando, Alessandro, Costa, Gabriele, and Merlo, Alessio, "Formal Modeling and Reasoning about the Android Security Framework," in *Proc. 7th International Symposium on Trustworthy Global Computing (TGC)*, volume 8191 of *Lecture Notes in Computer Science*

[77] Chaudhuri, Avik, "Language-based Security on Android," in *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS)*

[78] Fragkaki, Elli, Bauer, Lujo, Jia, Limin, and Swasey, David, "Modeling and Enhancing Android's Permission System," in *Proc. 17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459

[79] Shin, Wook, Kiyomoto, Shinsaku, Fukushima, Kazuhide, and Tanaka, Toshiaki, "A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework," in *Proc. IEEE Second International Conference on Social Computing (SOCIALCOM)*

[80] Chin, Erika, Felt, Adrienne Porter, Greenwood, Kate, and Wagner, David, "Analyzing Inter-Application Communication in Android," in *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*

[81] Fuchs, Adam, Chaudhuri, Avik, and Foster, Jeffrey, *SCanDroid: Automated Security Certiïn˘A¸cation of Android Applications*, Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, 2009

[82] Chen, Kevin Zhijie, Johnson, Noah, D'Silva, Vijay, Dai, Shuaifu, MacNamara, Kyle, Magrino, Tom, Wu, Edward, Rinard, Martin, and Song, Dawn, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS)*

[83] Enck, William, Gilbert, Peter, Han, Seungyeop, Tendulkar, Vasant, Chun, Byung-Gon, Cox, Landon P. Jung, Jaeyeon, McDaniel, Patrick, and Sheth, Anmol N. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, **32**, 2, 2014

[84] Nauman, Mohammad, Khan, Sohail, and Zhang, Xinwen, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proc. 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*

[85] Smalley, Stephen and Craig, Robert, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS)*

[86] Lange, Matthias, Liebergeld, Steffen, Lackorzynski, Adam, Warg, Alexander, and Peter, Michael, "L4Android: A Generic Operating System Framework for Secure Smartphones," in *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*

[87] Shamili, Ashkan Sharifi, Bauckhage, Christian, and Alpcan, Tansu, "Malware Detection on Mobile Devices Using Distributed Machine Learning," in *Proc. 20th International Conference on Pattern Recognition (ICPR)*

[88] Bläsing, Thomas, Batyuk, Leonid, Schmidt, Aubrey-Derrick, Camtepe, Seyit Ahmet, and Albayrak, Sahin, "An Android Application Sandbox System for Suspicious Software Detection," in *Proc. 5th International Conference on Malicious and Unwanted Software (Malware)*

[89] Xu, Rubin, Saïdi, Hassen, and Anderson, Ross, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proc. USENIX Security Symposium*

[90] Clarkson, Michael and Schneider, Fred, "Hyperproperties," *Journal of Computer Security*, **18**, 6, 2010, pp. 1157–1210

[91] Goguen, Joseph and Meseguer, José, "Security Policies and Security Models," in *Proc. IEEE Symposium on Security and Privacy*, pp. 11–20

[92] Dillig, Isil, Dillig, Thomas, and Aiken, Alex, "Precise reasoning for programs using containers," *ACM SIGPLAN Notices*, **46**, 1, 2011, pp. 187–200

[93] Peirce, C.S., **Collected papers of Charles Sanders Peirce**, Belknap Press, 1932

[94] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H., "Compositional shape analysis by means of bi-abduction," *POPL*, **44**, 1, 2009, pp. 289–300

[95] Giacobazzi, R., "Abductive analysis of modular logic programs," in *Proceedings of the 1994 International Symposium on Logic programming*, Citeseer, pp. 377–391

[96] Dillig, I., Dillig, T., and Aiken, A., "Automated error diagnosis using abductive inference," in *PLDI*

[97] Gulwani, S., McCloskey, B., and Tiwari, A., "Lifting abstract interpreters to quantified logical domains," in *POPL*, ACM, pp. 235–246

[98] Zhu, H., Dillig, I., and Dillig, T., "Abduction-based inference of library specifications for source-sink property verification," in *Technical Report, College of William & Mary*

[99] Li, B., Dillig, I., Dillig, T., McMillan, K., and Sagiv, M., "Synthesis of Circular Compositional Program Proofs via Abduction," in *To appear in TACAS*

[100] Dillig, I., Dillig, T., McMillan, K., and Aiken, A., "Minimum satisfying assignments for SMT," CAV

[101] Dillig, I., Dillig, T., and Aiken, A., "Small formulas for large programs: On-line constraint simplification in scalable static analysis," *Static Analysis*, 2011, pp. 236–252

[102] Dillig, Isil, Dillig, Thomas, and Aiken, Alex, "Small formulas for large programs: On-line constraint simplification in scalable static analysis," in *Static Analysis*, Springer, 2010 pp. 236–252

[103] Dillig, Isil, Dillig, Thomas, and Aiken, Alex, "Fluid updates: Beyond strong vs. weak updates," in *Programming Languages and Systems*, Springer, 2010 pp. 246–266

11.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ADI Android device implementation. 9, 11, 16, 17, 18, 19, 20, 22, 23, 24, 26, 34, 35, 36, 37, 54, 55

AUT application under test. 54, 55, 57, 58

PTA points-to analysis. iv, 20, 21, 22, 23, 24, 25, 26, 34, 35