Networking and Security Research Center
Technical Report

NAS-TR-0144-2011

A Study of Android Application Security

William Enck and Damien Octeau and Patrick McDaniel and Swarat Chaudhuri

13 January 2011 (updated 9 May 2011)

The Pennsylvania State University
344 IST Building
University Park, PA 16802 USA
http://nsrc.cse.psu.edu

A Study of Android Application Security[*]

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
{enck, octeau, mcdaniel, swarat}@cse.psu.edu

Abstract

The fluidity of application markets complicate smartphone security. Although recent efforts have shed light on particular security issues, there remains little insight into broader security characteristics of smartphone applications. This paper seeks to better understand smartphone application security by studying 1,100 popular free Android applications. We introduce the `ded` decompiler, which recovers Android application source code directly from its installation image. We design and execute a horizontal study of smartphone applications based on static analysis of 21 million lines of recovered code. Our analysis uncovered pervasive use/misuse of personal/phone identifiers, and deep penetration of advertising and analytics networks. However, we did not find evidence of malware or exploitable vulnerabilities in the studied applications. We conclude by considering the implications of these preliminary findings and offer directions for future analysis.

1 Introduction

The rapid growth of smartphones has lead to a renaissance for mobile services. Go-anywhere applications support a wide array of social, financial, and enterprise services for any user with a cellular data plan. Application markets such as Apple's App Store and Google's Android Market provide point and click access to hundreds of thousands of paid and free applications. Markets streamline software marketing, installation, and update—therein creating low barriers to bring applications to market, and even lower barriers for users to obtain and use them.

The fluidity of the markets also presents enormous security challenges. Rapidly developed and deployed applications [40], coarse permission systems [16], privacy-invading behaviors [14, 12, 21], malware [20, 25, 38], and limited security models [36, 37, 27] have led to exploitable phones and applications. Although users seemingly desire it, markets are not in a position to provide security in more than a superficial way [30]. The lack of a common definition for security and the volume of applications ensures that some malicious, questionable, and vulnerable applications will find their way to market.

[*]This Technical Report contains security analysis specifications and additional results excluded from the conference version appearing in the proceedings of the 20th USENIX Security Symposium [15].

In this paper, we broadly characterize the security of applications in the Android Market. In contrast to past studies with narrower foci, e.g., [14, 12], we consider a breadth of concerns including both dangerous functionality and vulnerabilities, and apply a wide range of analysis techniques. In this, we make two primary contributions:

- We design and implement a Dalvik decompilier, `ded`. `ded` recovers an application's Java source solely from its installation image by inferring lost types, performing DVM-to-JVM bytecode retargeting, and translating class and method structures.
- We analyze 21 million LOC retrieved from the top 1,100 free applications in the Android Market using automated tests and manual inspection. Where possible, we identify root causes and posit the severity of discovered vulnerabilities.

Our popularity-focused security analysis provides insight into the most frequently used applications. Our findings inform the following broad observations.

1. Similar to past studies, we found wide misuse of privacy sensitive information—particularly phone identifiers and geographic location. Phone identifiers, e.g., IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to accounts numbers.
2. We found no evidence of telephony misuse, background recording of audio or video, abusive connections, or harvesting lists of installed applications.
3. Ad and analytic network libraries are integrated with 51% of the applications studied, with Ad Mob

(appearing in 29.09% of apps) and Google Ads (appearing in 18.72% of apps) dominating. Many applications include more than one ad library.

4. Many developers fail to securely use Android APIs. These failures generally fall into the classification of insufficient protection of privacy sensitive information. However, we found no exploitable vulnerabilities that can lead malicious control of the phone.

This paper is an initial but not final word on Android application security. Thus, one should be circumspect about any interpretation of the following results as a definitive statement about how secure applications are today. Rather, we believe these results are indicative of the current state, but there remain many aspects of the applications that warrant deeper analysis. We plan to continue with this analysis in the future and have made the decompiler freely available at `http://siis.cse.psu.edu/ded/` to aid the broader security community in understanding Android security.

The following sections reflect the two thrusts of this work: Sections 2 and 3 provide background and detail our decompilation process, and Sections 4, 5, and 6 detail the application study. The remaining sections discuss our limitations and interpret the results.

2 Background

Android: Android is an OS designed for smartphones. Depicted in Figure 1, Android provides a sandboxed application execution environment. A customized embedded Linux system interacts with the phone hardware and an off-processor cellular radio. The Binder middleware and application API runs on top of Linux. To simplify, an application's only interface to the phone is through these APIs. Each application is executed within a Dalvik Virtual Machine (DVM) running under a unique UNIX `uid`. The phone comes pre-installed with a selection of *system applications*, e.g., phone dialer, address book.

Applications interact with each other and the phone through different forms of IPC. *Intents* are typed interprocess messages that are directed to particular applications or systems services, or broadcast to applications subscribing to a particular intent type. Persistent *content provider* data stores are queried through SQL-like interfaces. Background *services* provide RPC and callback interfaces that applications use to trigger actions or access data. Finally user interface *activities* receive named *action* signals from the system and other applications.

Binder acts as a mediation point for all IPC. Access to system resources (e.g., GPS receivers, text messaging, phone services, and the Internet), data (e.g., address books, email) and IPC is governed by permissions assigned at install time. The permissions requested by the application and the permissions required to access the

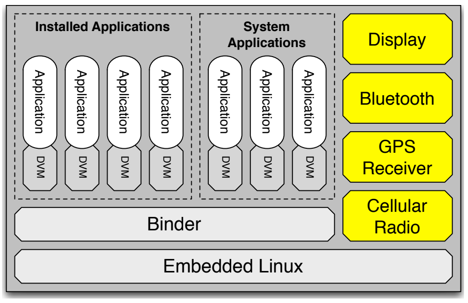

Figure 1: The Android system architecture

application's interfaces/data are defined in its *manifest* file. To simplify, an application is allowed to access a resource or interface if the required permission allows it. Permission assignment—and indirectly the security policy for the phone—is largely delegated to the phone's owner: the user is presented a screen listing the permissions an application requests at install time, which they can accept or reject.

Dalvik Virtual Machine: Android applications are written in Java, but run in the DVM. The DVM and Java bytecode run-time environments differ substantially:

Application Structure. Java applications are composed of one or more `.class` files, one file per class. The JVM loads the bytecode for a Java class from the associated `.class` file as it is referenced at run time. Conversely, a Dalvik application consists of a single `.dex` file containing all application classes.

Figure 2 provides a conceptual view of the compilation process for DVM applications. After the Java compiler creates JVM bytecode, the Dalvik `dx` compiler consumes the `.class` files, recompiles them to Dalvik bytecode, and writes the resulting application into a single `.dex` file. This process consists of the translation, reconstruction, and interpretation of three basic elements of the application: the constant pools, the class definitions, and the data segment. A constant pool describes, not surprisingly, the constants used by a class. This includes, among other items, references to other classes, method names, and numerical constants. The class definitions consist in the basic information such as access flags and class names. The data element contains the method code executed by the target VM, as well as other information related to methods (e.g., number of DVM registers used, local variable table, and operand stack sizes) and to class and instance variables.

Register architecture. The DVM is register-based, whereas existing JVMs are stack-based. Java bytecode can assign local variables to a local variable table before pushing them onto an operand stack for manipulation by opcodes, but it can also just work on the stack without explicitly storing variables in the table. Dalvik bytecode assigns local variables to any of the 2^{16} available regis-

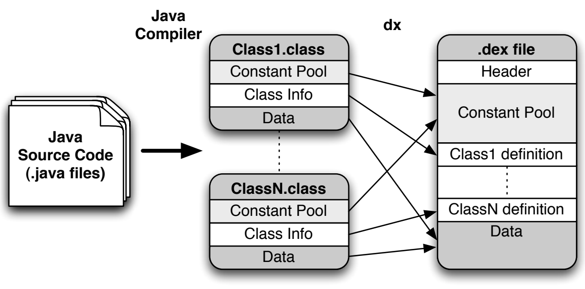

Figure 2: Compilation process for DVM applications

ters. The Dalvik opcodes directly manipulate registers, rather than accessing elements on a program stack.

Instruction set. The Dalvik bytecode instruction set is substantially different than that of Java. Dalvik has 218 opcodes while Java has 200; however, the nature of the opcodes is very different. For example, Java has tens of opcodes dedicated to moving elements between the stack and local variable table. Dalvik instructions tend to be longer than Java instructions; they often include the source and destination registers. As a result, Dalvik applications require fewer instructions. In Dalvik bytecode, applications have on average 30% fewer instructions than in Java, but have a 35% larger code size (bytes) [9].

Constant pool structure. Java applications replicate elements in constant pools within the multiple `.class` files, e.g., referrer and referent method names. The `dx` compiler eliminates much of this replication. Dalvik uses a single pool that all classes simultaneously reference. Additionally, `dx` eliminates some constants by inlining their values directly into the bytecode. In practice, integers, long integers, and single and double precision floating-point elements disappear during this process.

Control flow Structure. Control flow elements such as loops, switch statements and exception handlers are structured differently in Dalvik and Java bytecode. Java bytecode structure loosely mirrors the source code, whereas Dalvik bytecode does not.

Ambiguous primitive types. Java bytecode variable assignments distinguish between integer (`int`) and single-precision floating-point (`float`) constants and between long integer (`long`) and double-precision floating-point (`double`) constants. However, Dalvik assignments (`int/float` and `long/double`) use the same opcodes for integers and floats, e.g., the opcodes are untyped beyond specifying precision.

Null references. The Dalvik bytecode does not specify a `null` type, instead opting to use a zero value constant. Thus, constant zero values present in the Dalvik bytecode have ambiguous typing that must be recovered.

Comparison of object references. The Java bytecode uses typed opcodes for the comparison of object references (`if_acmpeq` and `if_acmpne`) and for null comparison of object references (`ifnull` and `ifnonnull`). The Dalvik bytecode uses a more simplistic integer comparison for these purposes: a comparison between two integers, and a comparison of an integer and zero, respectively. This requires the decompilation process to recover types for integer comparisons used in DVM bytecode.

Storage of primitive types in arrays. The Dalvik bytecode uses ambiguous opcodes to store and retrieve elements in arrays of primitive types (e.g., `aget` for int/float and `aget-wide` for long/double) whereas the corresponding Java bytecode is unambiguous. The array type must be recovered for correct translation.

3 The `ded` decompiler

Building a decompiler from DEX to Java for the study proved to be surprisingly challenging. On the one hand, Java decompilation has been studied since the 1990s—tools such as Mocha [5] date back over a decade, with many other techniques being developed [39, 32, 31, 4, 3, 1]. Unfortunately, prior to our work, there existed no functional tool for the Dalvik bytecode.[1] Because of the vast differences between JVM and DVM, simple modification of existing decompilers was not possible.

This choice to decompile the Java source rather than operate on the DEX opcodes directly was grounded in two reasons. First, we wanted to leverage existing tools for code analysis. Second, we required access to source code to identify false-positives resulting from automated code analysis, e.g., perform manual confirmation.

`ded` extraction occurs in three stages: *a*) retargeting, *b*) optimization, and *c*) decompilation. This section presents the challenges and process of `ded`, and concludes with a brief discussion of its validation. Interested readers are referred to [35] for a thorough treatment.

3.1 Application Retargeting

The initial stage of decompilation retargets the application `.dex` file to Java classes. Figure 3 overviews this process: (1) recovering typing information, (2) translating the constant pool, and (3) retargeting the bytecode.

Type Inference: The first step in retargeting is to identify class and method constants and variables. However, the Dalvik bytecode does not always provide enough information to determine the type of a variable or constant from its register declaration. There are two generalized cases where variable types are ambiguous: 1) constant and variable declaration only specifies the variable width (e.g., 32 or 64 bits), but not whether it is a float, integer, or null reference; and 2) comparison operators do not

[1]The undx and dex2jar tools attempt to decompile `.dex` files, but were non-functional at the time of this writing.

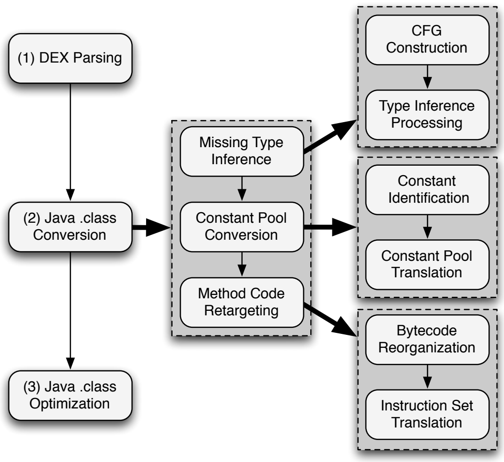

Figure 3: Dalvik bytecode retargeting

distinguish between integer and object reference comparison (i.e., null reference checks).

Type inference has been widely studied [45]. The seminal Hindley-Milner [33] algorithm provides the basis for type inference algorithms used by many languages such as Haskell and ML. These approaches determine unknown types by observing how variables are used in operations with known type operands. Similar techniques are used by languages with strong type inference, e.g., OCAML, as well weaker inference, e.g., Perl.

`ded` adopts the accepted approach: it infers register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. Because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be *path-sensitive*. Note further that `ded` type inference is also *method-local*. Because the types of passed parameters and return values are identified by method signatures, there is no need to search outside the method.

There are three ways `ded` infers a register's type. First, any comparison of a variable or constant with a known type exposes the type. Comparison of dissimilar types requires type coercion in Java, which is propagated to the Dalvik bytecode. Hence legal Dalvik comparisons always involve registers of the same type. Second, instructions such as `add-int` only operate on specific types, manifestly exposing typing information. Third, instructions that pass registers to methods or use a return value expose the type via the method signature.

The `ded` type inference algorithm proceeds as follows. After reconstructing the control flow graph, `ded` identifies any ambiguous register declaration. For each such register, `ded` walks the instructions in the control flow graph starting from its declaration. Each branch of the control flow encountered is pushed onto an inference stack, e.g., `ded` performs a depth-first search of the control flow graph looking for type-exposing instructions. If a type-exposing instruction is encountered, the variable is labeled and the process is complete for that variable.[2] There are three events that cause a branch search to terminate: a) when the register is reassigned to another variable (e.g., a new declaration is encountered), b) when a return function is encountered, and c) when an exception is thrown. After a branch is abandoned, the next branch is popped off the stack and the search continues. Lastly, type information is forward propagated, modulo register reassignment, through the control flow graph from each register declaration to all subsequent ambiguous uses. This algorithm resolves all ambiguous primitive types, except for one isolated case when all paths leading to a type ambiguous instruction originate with ambiguous constant instructions (e.g., all paths leading to an integer comparison originate with registers assigned a constant zero). In this case, the type does not impact decompilation, and a default type (e.g., integer) can be assigned.

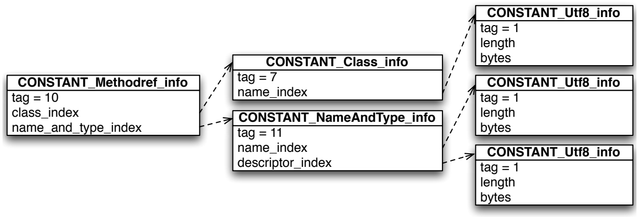

Figure 4: Java constant pool entry defining "class name," "method name," and "descriptor" for a method reference

Constant Pool Conversion: The `.dex` and `.class` file constant pools differ in that: *a*) Dalvik maintains a single constant pool for the application and Java maintains one for each class, and *b*) Dalvik bytecode places primitive type constants directly in the bytecode, whereas Java bytecode uses the constant pool for most references. We convert constant pool information in two steps.

The first step is to identify which constants are needed for a `.class` file. Constants include references to classes, methods, and instance variables. `ded` traverses the bytecode for each method in a class, noting such references. `ded` also identifies all constant primitives.

Once `ded` identifies the constants required by a class, it adds them to the target `.class` file. For primitive type constants, new entries are created. For class, method, and instance variable references, the created Java constant pool entries are based on the Dalvik constant pool

[2]Note that it is sufficient to find *any* type-exposing instruction for a register assignment. Any code that could result in different types for the same register would be illegal. If this were to occur, the primitive type would be dependent on the path taken at run time, a clear violation of Java's type system.

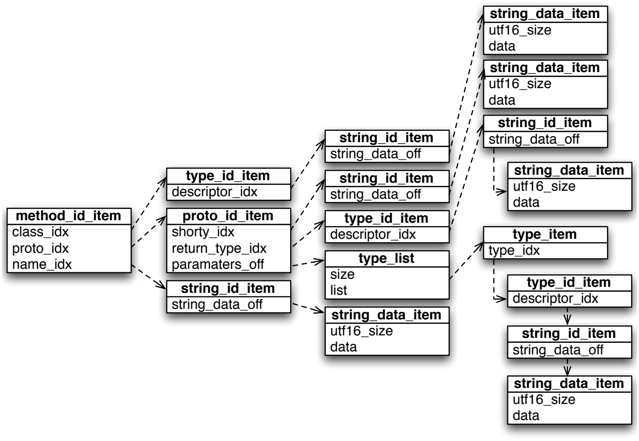

Figure 5: Dalvik constant pool entry defining "class name," "method name," and "descriptor" for a method reference

entries. The constant pool formats differ in complexity. Specifically, Dalvik constant pool entries use significantly more references to reduce memory overhead.

Figures 4 and 5 depict the method entry constant in both Java and Dalvik formats. Other constant pool entry types have similar structures. Each box is a data structure. Index entries (denoted as "idx" for the Dalvik format) are pointers to a data structure. The Java method constant pool entry, Both figures provide three strings: 1) the class name, 2) the method name, and 3) a descriptor string representing the argument and return types, but vary in complexity.

Method Code Retargeting: The final stage of the retargeting process is the translation of the method code. First, we preprocess the bytecode to reorganize structures that cannot be directly retargeted. Second, we linearly traverse the DVM bytecode and translate to the JVM.

The preprocessing phase addresses multidimensional arrays. Both Dalvik and Java use blocks of bytecode instructions to create multidimensional arrays; however, the instructions have different semantics and layout. `ded` reorders and annotates the bytecode with array size and type information for translation.

The bytecode translation linearly processes each Dalvik instruction. First, `ded` maps each referenced register to a Java local variable table index. Second, `ded` performs an instruction translation for each encountered Dalvik instruction. As Dalvik bytecode is more compact and takes more arguments, one Dalvik instruction frequently expands to multiple Java instructions. Third, `ded` patches the relative offsets used for branches based on preprocessing annotations. Finally, `ded` defines exception tables that describe `try/catch/finally` blocks. The resulting translated code is combined with the constant pool to creates a legal Java `.class` file.

The following is an example translation for `add-int`:

Dalvik	Java
`add-int` d_0, s_0, s_1	`iload` s'_0
	`iload` s'_1
	`iadd`
	`istore` d'_0

where `ded` creates a Java local variable for each register, i.e., $d_0 \rightarrow d'_0$, $s_0 \rightarrow s'_0$, etc. The translation creates four Java instructions: two to push the variables onto the stack, one to add, and one to pop the result.

3.2 Optimization and Decompilation

At this stage, the retargeted `.class` files can be decompiled using existing tools, e.g., Fernflower [1] or Soot [46]. However, `ded`'s bytecode translation process yields unoptimized Java code. For example, Java tools often optimize out unnecessary assignments to the local variable table, e.g., unneeded return values. Without optimization, decompiled code is complex and frustrates analysis. Furthermore, artifacts of the retargeting process can lead to decompilation errors in some decompilers. The need for bytecode optimization is easily demonstrated by considering decompiled loops. Most decompilers convert `for` loops into infinite loops with `break` instructions. While the resulting source code is functionally equivalent to the original, it is significantly more difficult to understand and analyze, especially for nested loops. Thus, we use Soot as a post-retargeting optimizer. While Soot is centrally an optimization tool with the ability to recover source code in most cases, it does not process certain legal program idioms (bytecode structures) generated by `ded`. In particular, we encountered two central problems involving, 1) interactions between synchronized blocks and exception handling, and 2) complex control flows caused by break statements. While the Java bytecode generated by `ded` is legal, the source code failure rate reported in the following section is almost entirely due to Soot's inability to extract source code from these two cases. We will consider other decompilers in future work, e.g., Jad [4], JD [3], and Fernflower [1].

3.3 Source Code Recovery Validation

We have performed extensive validation testing of `ded` [35]. The included tests recovered the source code for small, medium and large open source applications and found no errors in recovery. In most cases the recovered code was virtually indistinguishable from the original source (modulo comments and method local-variable names, which are not included in the bytecode).

We also used `ded` to recover the source code for the top 50 free applications (as listed by the Android Market) from each of the 22 application categories—1,100 in total. The application images were obtained from the mar-

Table 1: Studied Applications (from Android Market)

Category	Total Classes	Retargeted Classes	Decompiled Classes	LOC
Comics	5627	99.54%	94.72%	415625
Communication	23000	99.12%	92.32%	1832514
Demo	8012	99.90%	94.75%	830471
Entertainment	10300	99.64%	95.39%	709915
Finance	18375	99.34%	94.29%	1556392
Games (Arcade)	8508	99.27%	93.16%	766045
Games (Puzzle)	9809	99.38%	94.58%	727642
Games (Casino)	10754	99.39%	93.38%	985423
Games (Casual)	8047	99.33%	93.69%	681429
Health	11438	99.55%	94.69%	847511
Lifestyle	9548	99.69%	95.30%	778446
Multimedia	15539	99.20%	93.46%	1323805
News/Weather	14297	99.41%	94.52%	1123674
Productivity	14751	99.25%	94.87%	1443600
Reference	10596	99.69%	94.87%	887794
Shopping	15771	99.64%	96.25%	1371351
Social	23188	99.57%	95.23%	2048177
Libraries	2748	99.45%	94.18%	182655
Sports	8509	99.49%	94.44%	651881
Themes	4806	99.04%	93.30%	310203
Tools	9696	99.28%	95.29%	839866
Travel	18791	99.30%	94.47%	1419783
Total	**262110**	**99.41%**	**94.41%**	**21734202**

ket using a custom retrieval tool on September 1, 2010. Table 1 lists decompilation statistics. The decompilation of all 1,100 applications took 497.7 hours (about 20.7 days) of compute time. Soot dominated the processing time: 99.97% of the total time was devoted to Soot optimization and decompilation. The decompilation process was able to recover over 247 thousand classes spread over 21.7 million lines of code. This represents about 94% of the total classes in the applications. All decompilation errors are manifest during/after decompilation, and thus are ignored for the study reported in the latter sections. There are two categories of failures:

Retargeting Failures. 0.59% of classes were not retargeted. These errors fall into three classes: *a*) unresolved references which prevent optimization by Soot, *b*) type violations caused by Android's `dex` compiler and *c*) extremely rare cases in which `ded` produces illegal bytecode. Recent efforts have focused on improving optimization, as well as redesigning `ded` with a formally defined type inference apparatus. Parallel work on improving `ded` has been able to reduce these errors by a third, and we expect further improvements in the near future.

Decompilation Failures. 5% of the classes were successfully retargeted, but Soot failed to recover the source code. Here we are limited by the state of the art in decompilation. In order to understand the impact of decompiling `ded` retargeted classes verses ordinary Java `.class` files, we performed a parallel study to evaluate Soot on Java applications generated with traditional Java compilers. Of 31,553 classes from a variety of packages, Soot was able to decompile 94.59%, indicating we cannot do better while using Soot for decompilation.

A possible way to improve this is to use a different decompiler. Since our study, Fernflower [1] was available for a short period as part of a beta test. We decompiled the same 1,100 optimized applications using Fernflower and had a recovery rate of 98.04% of the 1.65 million retargeted methods–a significant improvement. Future studies will investigate the fidelity of Fernflower's output and its appropriateness as input for program analysis.

4 Evaluating Android Security

Our Android application study consisted of a broad range of tests focused on three kinds of analysis: *a*) exploring issues uncovered in previous studies and malware advisories, *b*) searching for general coding security failures, and *c*) exploring misuse/security failures in the use of Android framework. The following discusses the process of identifying and encoding the tests.

4.1 Analysis Specification

We used four approaches to evaluate recovered source code: *control flow analysis*, *data flow analysis*, *structural analysis*, and *semantic analysis*. Unless otherwise specified, all tests used the Fortify SCA [2] static analysis suite, which provides these four types of analysis. The following discusses the general application of these approaches.

Control flow analysis. Control flow analysis imposes constraints on the sequences of actions executed by an input program *P*, classifying some of them as errors. Essentially, a control flow rule is an automaton *A* whose input words are sequences of actions of *P*—i.e., the rule *monitors* executions of *P*. An erroneous action sequence is one that drives *A* into a predefined *error state*. To statically detect violations specified by *A*, the program analysis traces each control flow path in the tool's model of *P*, synchronously "executing" *A* on the actions executed along this path. Since not all control flow paths in the model are feasible in concrete executions of *P*, false positives are possible. False negatives are also possible in principle, though uncommon in practice. Figure 6 shows an example automaton for sending intents. Here, the error state is reached if the intent contains data and is sent unprotected without specifying the target component, resulting in a potential unintended information leakage.

Data flow analysis. Data flow analysis permits the declarative specification of problematic data flows in the input program. For example, an Android phone contains several pieces of private information that should never leave the phone: the user's phone number, IMEI (device

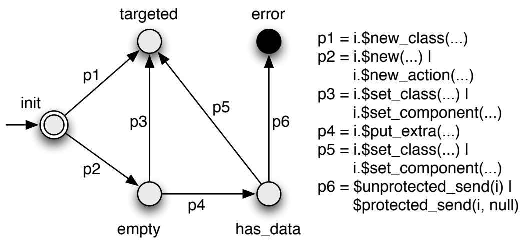

Figure 6: Example control flow specification

ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number). In our study, we wanted to check that this information is not leaked to the network. While this property can in principle be coded using automata, data flow specification allows for a much easier encoding. The specification declaratively labels program statements matching certain syntactic patterns as *data flow sources* and *sinks*. Data flows between the sources and sinks are violations.

Structural analysis. Structural analysis allows for declarative pattern matching on the abstract syntax of the input source code. Structural analysis specifications are not concerned with program executions or data flow, therefore, analysis is local and straightforward. For example, in our study, we wanted to specify a bug pattern where an Android application mines the device ID of the phone on which it runs. This pattern was defined using a structural rule that stated that the input program called a method *getDeviceId()* whose enclosing class was *android.telephony.TelephonyManager*.

Semantic analysis. Semantic analysis allows the specification of a limited set of constraints on the values used by the input program. For example, a property of interest in our study was that an Android application does not send SMS messages to hard-coded targets. To express this property, we defined a pattern matching calls to Android messaging methods such as *sendTextMessage()*. Semantic specifications permit us to directly specify that the first parameter in these calls (the phone number) is not a constant. The analyzer detects violations to this property using constant propagation techniques well known in program analysis literature.

4.2 Analysis Overview

Our analysis covers both dangerous functionality and vulnerabilities. Selecting the properties for study was a significant challenge. The following overviews our specifications for accessibility. Section 5 discusses the specifications in detail.

Misuse of Phone Identifiers (Section 6.1.1). Previous studies [14, 12] identified phone identifiers leaking to remote network servers. We seek to identify not only the existence of data flows, but understand why they occur.

Exposure of Physical Location (Section 6.1.2). Previous studies [14] identified location exposure to advertisement servers. Many applications provide valuable location-aware utility, which may be desired by the user. By manually inspecting code, we seek to identify the portion of the application responsible for the exposure.

Abuse of Telephony Services (Section 6.2.1). Smartphone malware has sent SMS messages to premium-rate numbers. We study the use of hard-coded phone numbers to identify SMS and voice call abuse.

Eavesdropping on Audio/Video (Section 6.2.2). Audio and video eavesdropping is a commonly discussed smartphone threat [41]. We examine cases where applications record audio or video without control flows to UI code.

Botnet Characteristics (Sockets) (Section 6.2.3). PC botnet clients historically use non-HTTP ports and protocols for command and control. Most applications use HTTP client wrappers for network connections, therefore, we examine *Socket* use for suspicious behavior.

Harvesting Installed Applications (Section 6.2.4). The list of installed applications is a valuable demographic for marketing. We survey the use of APIs to retrieve this list to identify harvesting of installed applications.

Use of Advertisement Libraries (Section 6.3.1). Previous studies [14, 12] identified information exposure to ad and analytics networks. We survey inclusion of ad and analytics libraries and the information they access.

Dangerous Developer Libraries (Section 6.3.2). During our manual source code inspection, we observed dangerous functionality replicated between applications. We report on this replication and the implications.

Android-specific Vulnerabilities (Section 6.4). We search for non-secure coding practices [17, 10], including: writing sensitive information to logs, unprotected broadcasts of information, IPC null checks, injection attacks on intent actions, and delegation.

Misuse of Passwords (Section 6.5.1). We look for vulnerabilities in password management, including: hard-coded passwords, null or empty passwords, storage of plaintext passwords, and leakage of passwords to external storage.

Misuse of Cryptography (Section 6.5.2). We look for cryptography misuse, including: insufficient key size, deprecated algorithms (e.g., DES, MD5), inappropriate padding, and weak pseudo-random number generators.

Injection Vulnerabilities (Section 6.5.3). We look for injection vulnerabilities, including: file paths, databases, and command execution.

5 Analysis Query Definitions

We now describe in detail the source code analysis specifications used to identify dangerous functionality and vulnerabilities. We define control flow analysis as FSMs, and data flow analysis as articulations of sources and sinks. The structural and semantic are presented in the Fortify SCA syntax used for the analysis. We begin with analysis definitions for dangerous functionality.

5.1 Dangerous Functionality

Our analysis considers several types of dangerous functionality. First, we discuss exfiltration of information using data flow analysis. Specifically, we consider location and phone identifiers as data flow sources. Second, we discuss the use of semantic analysis to identify misuse of telephony services by looking for static values passed to corresponding APIs. Third, we describe how to identify recording of audio and video in the background using both structural and control flow analysis. Fourth, we discuss how the *Socket* API can be used as an indicator for identifying botnet-like activity. Here, we use structural analysis as an more-intelligent *grep* utility. The analysis results in Section 6 demonstrate how even simple queries can shed enormous insight. Finally, we discuss specifications to help identify harvesting of the list of installed applications.

Note that all of the specifications are heuristics in and of themselves. They represent idioms that indicate potential dangerous functionality. The goal of study is to *practically* understand functionality by any means necessary. Hence, we use program analysis to minimize manual inspection effort. As will be shown in the results for several rules, a small number of false positives is worth the manual effort required to identify the non-existence of a dangerous behavior.

5.1.1 Exfiltration of Information

We use data flow analysis to identify exfiltration of phone identifiers and location. Our specification uses the following data flow sources in sinks. Note that there are several APIs through which information may be transmitted to the network. As discussed below, subtleties of these APIs require creative definitions.

Additionally, note that the Fortify SCA data flow specification syntax allows us to apply source and sink targets to all classes that implement, override, and extend the specified class. This syntax significantly simplifies definition specification.

Data Flow Sources. We define a data flow source for phone identifier APIs of interest: *getLine1Number()* (phone number); *getDeviceID()* (IMEI); *getSubscriberId()* (IMSI); and *getSimSerialNumber()* (ICC-ID). All of these APIs are defined within the *TelephonyManager* class in the *android.telephony* namespace.

Location information can be accessed directly, or obtained via a callback method registered with the phone's location manager. Instead of using these APIs as data sources, we observe that location information is always obtained as a *Location* object defined in the *android.location* namespace. The latitude and longitude values are obtained by invoking the *getLatitude()* and *getLongitude()* access methods. Therefore, we use these methods as data flow sources. While this choice may result in additional false positives, it simplifies source specification, and possibly more importantly, shortens the data flow path that must be tracked. This latter advantage is important, because it lessens the impact of Android IPC and non-recoverable source code.

Data Flow Sinks. The network is our primary data sink; however, there are multiple APIs through which information can leave an application. Specifically, Android provides HTTP-specific APIs to simplify communication with Web servers. We consider both the *URLConnection* class in the *java.net* namespace (and extended classes) as well as the *HttpClient* class in *org.apache.http.client* namespace.

Our data flow sinks consider HTTP GET and POST parameters, request properties, and data written to the output stream. Specification of parameters and properties to APIs is straightforward. First, HTTP GET parameters are commonly encoded directly in a URL. Therefore, we define a data flow sink at the constructor and *set()* method of the *URL* class in the *java.net* namespace. Second, the *HttpClient* API is commonly used to specify HTTP POST parameters. We define a data flow sink at the *setEntity()* method of the *HttpEntityEnclosingRequestBase* class of the *org.apache.http.client.methods* namespace. However, to ensure the data flow is properly tracked, defined several data flow pass-through rules for the constructors the *HttpEntity* and *AbstractHttpEntity* in the *org.apache.http.entity* namespace, and the *BasicNameValuePair* class in the *org.apache.http.message* namespace. HTTP POST parameters are passed to these classes, and we must track information through them to *setEntity()*. Since these classes are within the Android library (and not included in the source code analysis), data flow pass-through specifications are required. Finally, we define a data flow sink at the *setRequestProperty()* of the *URLConnection* class in the *java.net* namespace.

In addition to these APIs, sensitive information may be written to an output stream. However, we must distinguish between a file output stream and a network output stream. To distinguish between these two IO interfaces, we define a data flow source at the *getOutputStream()* method of the *URLConnection* in the *java.net* namespace. This source adds the flat `NETOUT` to the returned *OutputStream* object. Next, leveraging the implements/overrides/extends syntactic feature in SCA, we de-

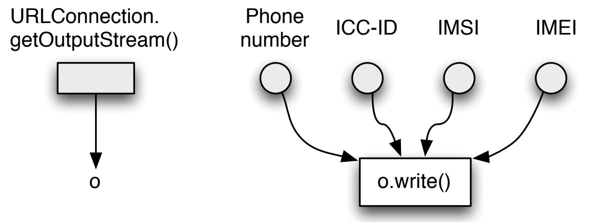

Figure 7: Data flow analysis to distinguish file and network IO.

fine a pass-through rule for all *OutputStream* and *Writer* classes in the *java.io* namespace. This effectively propagates the NETOUT flag to all corresponding output child classes. Finally, we define a data flow sink at the *write()* method of the *OutputStream* and *Writer* classes (again utilizing implements/overrides/extends), to identify the combination of the NETOUT flag *and* one of the data flow sources for location and phone identifiers, as discussed above. Figure 7 depicts this approach.

Identification of API Use. The data flow analysis may have false negatives resulting from Android IPC and code recovery. To gain a better understanding of the information that applications access, we also define simple queries for API use. For phone identifiers, our rules identify when the above defined methods are called. For location, we consider both the *getLastKnownLocation()* and *requestLocationUpdates()* methods of the *LocationManager* class in the *android.location* namespace. Additionally, we query for instances of the *onLocationChanged()* callback method defined within a *LocationListener* class in the *android.location* namespace.

5.1.2 Misuse of Telephony Services

We use semantic analysis to identify misuse of telephony service APIs. Specifically, we are interested in the destination phone number used for SMS messages and placing voice calls. Android defines significantly different APIs for SMS and voice calls. Use cases are also different. For example, calling statically defined customer service numbers from an application is sometimes desirable, whereas sending SMS messages are generally not used for customer service (as email provides a more suitable mechanism for asynchronous help).

SMS Services. We looked for hard-coded phone numbers in the SMS API using semantic analysis. Our specification identifies constant values passed to the destination phone number in the *sendTextMessage()*, *sendDataMessage()*, and *sendMultiPartTextMessage()* methods of the *SmsManager* class in the *android.telephony* namespace. Note that this specification is limited by Fortify SCA's analysis. The semantic analysis is limited to constant values, which are semantically different than hard-coded values. While the definition identifies hard-coded values passed directly to the API, it does not identify hard-coded values assigned to variables, unless the Java variable is made `final`.

Voice Call Services. Android does not use a simple method for making voice calls. Instead, an intent with either the CALL or DIAL framework defined action strings is used to start an activity. This intent specifies a phone number as its data URI field. If the CALL action string is used, the application must have the CALL_PHONE permission, and the voice call is initiated immediately. If the DIAL action string is used, Android starts the phone dialer with the specified number entered, but the user must select the "call" button to initiate the voice call. No permission is required to use the DIAL action string.

Lacking an obvious method argument for semantic analysis, we consider the *parse()* method of the *Uri* class in the *java.net* namespace. *Uri* objects are used for various purposes; however, Android uses the "`tel:`" prefix to identify telephone numbers. Therefore, our semantic analysis queries for constant strings beginning with "`tel:`" (note that again, this is an approximation of the desired analysis for hard-coded values). We also define a query that includes "900" in the semantic analysis to identify premium-rate numbers.

5.1.3 Background Audio and Video Recording

Android provides three APIs for recording audio and visual information. The *AudioRecord* class in the *android.media* namespace defines the *read()* method to access microphone input. The *Camera* class in the *android.hardware* namespace defines the *takePicture()* method to capture still images. Finally, the *MediaRecorder* class in the *android.media* namespace is used to both record audio and video.

Audio. Simply recording audio is not suspicious in and of itself. As an approximation of recording audio without the user's knowledge, we look for use of the *AudioRecord* API in code not accessible to an Android activity component. Figure 8 provides this structural analysis definition. Here, we use the special `reachedBy` direc-

```
FunctionCall c: c.function.name == "read" and
c.function.enclosingClass.name ==
                       "android.media.AudioRecord"
and not c.enclosingClass reachedBy
[ Class a: a.supers contains
[ Class super: super.name ==
                           "android.app.Activity"]]
```

Figure 8: Structural analysis definition to identify background audio recording.

tive that considers the call graph leading to the execution of *read()*.

A nearly identical specification is used for the *start()* method of *MediaRecord*. This rule also identifies video recording; however, as we discuss below, there are alternative methods of identifying background recording of video.

Still Images. Still images are obtained using the *takePicture()* method of the *Camera* class. The API requires that *startPreview()* is called before *takePicture()*. In order to call *takePicture()*, *setPreviewDisplay()* must be called to specify a user interface layout area to display a preview of the still image. Therefore, still images cannot be taken without the user's knowledge.

Video. The *MediaRecorder* class contains methods to set audio and video sources. Our specification ensures that if *setVideoSource()* is called, then so is *setPreviewDisplay()*, which will inform the user that the camera is being accessed. The FSM for our control flow specification is shown in Figure 9.

5.1.4 Socket API Use

We hypothesize that most network-based Android applications are HTTP clients. Android includes the *URLConnection* and *HttpClient* APIs to abstract network communication to Web servers. Therefore, we expect most applications will choose to use these APIs over direct access with *Socket* objects. To identify the use sockets, we use simple structural analysis that queries invocation of the *connect()* method and the *InetAddress* and *String* constructor variants of the *Socket* class in the *java.net* namespace.

Note that this definition is not looking for dangerous functionality directly. In fact, we expect some applications will use sockets directly, e.g., for streaming audio. Rather, we hope to characterize how many applications use sockets directly in order to evaluate our hypothesis and determine its usefulness as a security analysis heuristic.

5.1.5 Harvesting Installed Applications

Android provides the *PackageManager* API (in the *android.content.pm* namespace) to abstract access to information pertaining to installed applications, their components, and permissions. Using this API, an application harvest the list of installed applications.

There are two general approaches to acquiring the list of installed applications. First, one of the *getInstalledApplications()*, *getInstalledPackages()*, or *getPreferredPackages()* can be called. Second, generic queries can be made to one of the *queryIntentActivities()*, *queryIntentServices()*, *queryBroadcastReceivers()*, or

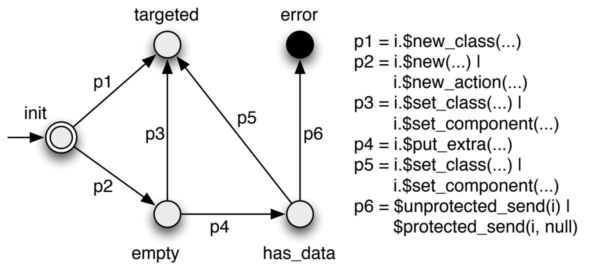

Figure 10: FSM for control flow analysis to identify leaks to IPC.

queryContentProviders() methods. We use a simple structural analysis definition for each of these two cases.

5.2 Vulnerabilities

Android applications are written Java. There are many vulnerability definitions for Java applications; however, the definitions are frequently designed to detect vulnerabilities in server applications. While our study includes analysis of standard Java vulnerabilities, we seek to define Android-specific vulnerability specifications. The remainder of this section discusses these specifications. First, we use data flow analysis to identify location and phone identifiers written to Android's insecure logging interface. Second, we use control flow analysis to identify unsafe intent broadcasts that may expose sensitive information. Third, we use control flow analysis to identify insufficiently protected dynamically created broadcast receiver components. Fourth, we use data flow analysis to identify injection attacks on intent messages. Fifth, we use control flow analysis to identify delegation vulnerabilities when using pending intents. Finally, we use control flow analysis to look specifically for failures to check for null values before dereferencing objects obtained via Android's IPC API.

5.2.1 Leaking Information to Insecure Locations

We look for two types of information leaks: leaks to log files, and leaks to IPC. We begin with log files.

Leaks to Log files. Java has several APIs for simplified application logging. These log files are often stored within a directory only accessible to the application writing to the log. However, Android includes the *logcat* API for centralized logging by all applications. Any application with the `READ_LOGS` permission can access the logs. For the data flow analysis, we use the same sources for location and phone identifiers indicated in Section 5.1.1 to identify exfiltration. We define a data flow sink at all methods of the *Log* class in the *android.util* namespace.

Leaks to IPC. Sending sensitive information over IPC

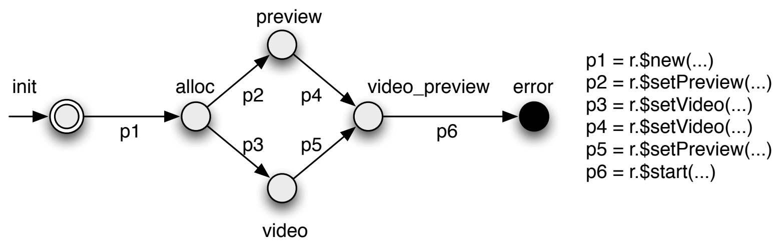

Figure 9: FSM for control flow analysis to identify background video recording.

is only a vulnerability if it can be accessed by an unintended application. Therefore, instead of using data flow analysis to identify leaks to IPC, we use control flow analysis to determine if an intent message is broadcast to an action string without specifying either a permission to protect the intent, or the target application and component name. If this condition is not met, a malicious application can eavesdrop to potentially access sensitive information. The FSM for this control flow analysis is shown in Figure 10.

In the figure, paths lead to the "targeted" state if the application and component is explicitly provided. Note that there are several ways in which this can occur. Next, we only want to identify intent messages that contain "extras" information, therefore, the FSM must transition to `has_data` before reaching the `error` state. We assume intent messages without extras do not contain sensitive information. Finally, eavesdropping can only occurs if the intent is broadcast without a permission protecting its access. Note that the developer may pass `null` to the API accepting a permission.

5.2.2 Unprotected Broadcast Receivers

Activity and service components occasionally dynamically register broadcast receivers to receive intent messages only during a specific time interval (e.g., while the activity is in focus). By default, if a component defines an intent filter, it is public, and any application can construct an intent message that can be sent directly to it. In contrast, if there is no intent filter, the component is private, and can only be accessed by components in the same application. Developers can prevent forging attacks by protecting public components with a permission. However, there are two APIs to dynamically register a broadcast receiver, and one does not include a permission to protect the component.

Figure 11 shows the FSM for control flow analysis to identify unprotected dynamic registration of a broadcast receiver that has an intent filter. Here, the FSM tracks an intent filter object `if`. The intent filter will only make the broadcast receiver public if it contains at least one action string. Finally, the FSM considers the case where

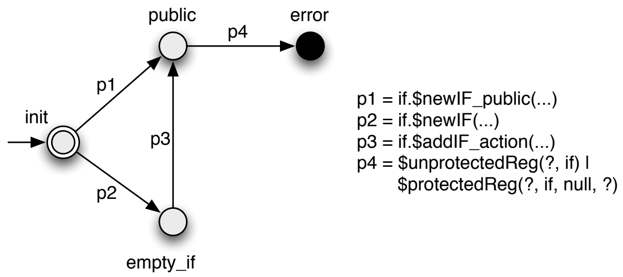
public
error
init
empty_if
p1
p2
p3
p4
p1 = if.$newIF_public(...)
p2 = if.$newIF(...)
p3 = if.$addIF_action(...)
p4 = $unprotectedReg(?, if) |
$protectedReg(?, if, null, ?)

Figure 11: FSM for control flow analysis to identify unprotected broadcast receivers.

the developer passes `null` as a permission.

5.2.3 Intent Injection Attacks

Android's intent messages perform actions and therefore subject to injection attacks from the network and IPC input from other applications. Specifically, we are interested in untrusted input that is used in an address field of an intent message, which includes both the action string and the destination application and component fields.

Network Data Flow Sources. To identify untrusted network input, we assign the data flow source flag `NETIN` at the return of the *getInputStream()* method of the *URLConnection* class in the *java.net* namespace, as well as the *getEntity()* method of the *HttpResponse* class in the *org.apache.http* namespace. To account for logic within Android libraries, we define pass-through specifications to propagate `NETIN` through *InputStream* and *Reader* class constructors (*java.io* namespace), and to the return of the *read()* methods of these classes. Note that we again leverage the implements/overrides/extends feature for method specification. Additionally, to propagate `NETIN` for input from *HttpResponse*, we define a pass-through specification for the *toString()*, *toByteArray()*, and *getContentCharSet()* methods of the *EntityUtils* class in the *org.apache.http.util* namespace.

IPC Data Flow Sources. To identify untrusted IPC input, we assign the data flow source flag `IPCIN` to the return of all methods with prefix "*get*" in the *Intent* class

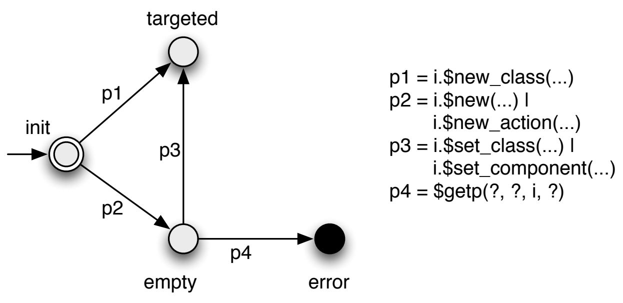

Figure 12: FSM for control flow analysis to identify unsafe pending intents.

of the *android.content* namespace. The prefix matching is achieved using wildcard symbols.

Intent Address Data Flow Sink. We define the data flow sink for NETIN and IPCIN flags at the inputs to the constructor, *setAction()*, *setClassName()*, and *setComponent()* methods of the *Intent* class in the *android.content* namespace. Note that we define an additional pass-through specification for the constructor of the *ComponentName* class of the *android.content* namespace. This class converts text strings specifying the application and component names into a *ComponentName* object passed to *setComponent()*.

5.2.4 Delegating Control

Pending intents are created from intent messages. To create a pending intent, an application defines an intent message and specifies whether it should be used for an activity, service, or broadcast. The pending intent itself is simply a reference to the Intent, and can be shared via RPC to another process within the application, or to another application altogether. When another application receives a pending intent, it can fill in any missing fields (e.g., "extras" values), and invoke the intent. When the intent is invoked, it targets the predefined component type (e.g., activity) and executes within the protection domain of the application that created the pending intent. Frequently, applications use pending intents as "long-term" callbacks, e.g., to be woken up by the system's alarm service, or Android's notification manager that allows the user to resume applications based on events.

A vulnerability can result if a pending intent is created from an intent message that does not explicitly define the target component. If the target component is not defined, the application receiving the pending intent can effectively redirect the intent to the application of its choosing. Figure 12 shows the FSM for control flow analysis to identify pending intents created from intent messages without a specified target component.

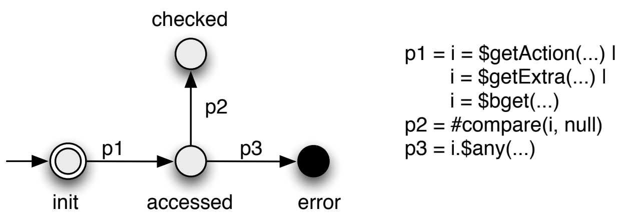

Figure 13: FSM for control flow analysis of null checks on IPC input.

In the figure, state variable `i` tracks an intent message from creation. Once a target component is specified, the FSM transitions to the `targeted` state, which can never transition to the `error` state. However, if a pending intent is created from an intent message with the `empty` state, an error indicating a vulnerability is reported. Here, `$getp()` is a macro that matches the *getActivity()*, *getService()*, and *getBroadcast()* methods of the *PendingIntent* class in the *android.app* namespace.

Note that this FSM does not identify how the pending intent is used. In a practical threat model, sending the pending intent to a system application is not a vulnerability. However, for our analysis, we seek to understand how applications use pending intents in potentially unsafe scenarios. Future work will consider how to reduce the possibility of false positive.

5.2.5 Null Checks on IPC Input

Android terminates an application if it dereferences null. Frequently, applications perform actions based on objects received via IPC. If the application does not perform null checks on received objects, remote applications can cause the application to crash. This denial of service vulnerability is particularly useful for an adversary attempting to stop background service functionality, as the user may not be aware the service has terminated.

Figure 13 shows the FSM for control flow analysis to identify missed null checks on IPC input. The FSM tracks any variable `i` and transitions to state `accessed` when `i` is input from IPC. Here, `$getAction()` is the similarly named method of the *Intent* class in the *android.content* namespace; `$getExtra()` matches any `get.*Extra()` method of the *Intent* class, and `$bget` matches any `get.*()` method of the *Bundle* class in the *android.os* namespace. The FSM uses the analysis directive `#compare()` to determine if the application compares `i` to `null`. However, if this does not occur, `$any()` matches any method of object `i`, which will result in a null dereference.

To gain additional insight as to where null dereferences on IPC input occur, we define multiple versions of the FSM that are only evaluated in *Activity*, *Service*, and

Table 2: Access of Phone Identifier APIs

Identifier	# Calls	# Apps	# w/ Permission*
Phone Number	167	129	105
IMEI	378	216	184†
IMSI	38	30	27
ICC-ID	33	21	21
Total Unique	-	246	210†

* Defined as having the READ_PHONE_STATE permission.
† Only 1 app did not also have the INTERNET permission.

BroadcastReceiver classes. Note that this limits analysis to methods defined within classes extending these classes, and does not include null dereferences in objects used by activities, services, and broadcast receivers. Therefore, our analysis considers both general and specific cases.

6 Application Analysis Results

In this section, we document the program analysis results and manual inspection of identified violations.

6.1 Information Misuse

In this section, we explore how sensitive information is being leaked [12, 14] through information sinks including *OutputStream* objects retrieved from *URLConnection*s, HTTP GET and POST parameters in *HttpClient* connections, and the string used for *URL* objects. Future work may also include SMS as a sink.

6.1.1 Phone Identifiers

We studied four phone identifiers: phone number, IMEI (device identifier), IMSI (subscriber identifier), and ICC-ID (SIM card serial number). We performed two types of analysis: *a*) we scanned for APIs that access identifiers, and *b*) we used data flow analysis to identify code capable of sending the identifiers to the network.

Table 2 summarizes APIs calls that receive phone identifiers. In total, 246 applications (22.4%) included code to obtain a phone identifier; however, only 210 of these applications have the READ_PHONE_STATE permission required to obtain access. Section 6.3 discusses code that probes for permissions. We observe from Table 2 that applications most frequently access the IMEI (216 applications, 19.6%). The phone number is used second most (129 applications, 11.7%). Finally, the IMSI and ICC-ID are very rarely used (less than 3%).

Table 3 indicates the data flows that exfiltrate phone identifiers. The 33 applications have the INTERNET permission, but 1 application does not have the READ_PHONE_STATE permission. We found data flows for all four identifier types: 25 applications have IMEI data flows; 10 applications have phone number data flows; 5 applications have IMSI data flows; and 4 applications have ICC-ID data flows.

Table 3: Detected Data Flows to Network Sinks

	Phone Identifiers		Location Info.	
Sink	**# Flows**	**# Apps**	**# Flows**	**# Apps**
OutputStream	10	9	0	0
HttpClient Param	24	9	12	4
URL Object	59	19	49	10
Total Unique	-	33	-	13

To gain a better understanding of how phone identifiers are used, we manually inspected all 33 identified applications, as well as several additional applications that contain calls to identifier APIs. We confirmed exfiltration for all but one application. In this case, code complexity hindered manual confirmation; however we identified a different data flow not found by program analysis. The analysis informs the following findings.

Finding 1 - *Phone identifiers are frequently leaked through plaintext requests.* Most sinks are HTTP GET or POST parameters. HTTP parameter names for the IMEI include: "uid," "user-id," "imei," "deviceId," "deviceSerialNumber," "devicePrint," "X-DSN," and "uniquely_code"; phone number names include "phone" and "mdn"; and IMSI names include "did" and "imsi." In one case we identified an HTTP parameter for the ICC-ID, but the developer mislabeled it "imei."

Finding 2 - *Phone identifiers are used as device fingerprints.* Several data flows directed us towards code that reports not only phone identifiers, but also other phone properties to a remote server. For example, a wallpaper application (com.eoeandroid.eWallpapers.cartoon) contains a class named *SyncDeviceInfosService* that collects the IMEI and attributes such as the OS version and device hardware. The method *sendDeviceInfos()* sends this information to a server. In another application (com.avantar.wny), the method *PhoneStats.toUrlFormatedString()* creates a URL parameter string containing the IMEI, device model, platform, and application name. While the intent is not clear, such fingerprinting indicates that phone identifiers are used for more than a unique identifier.

Finding 3 - *Phone identifiers, specifically the IMEI, are used to track individual users.* Several applications contain code that binds the IMEI as a unique identifier to network requests. For example, some applications (e.g. com.Qunar and com.nextmobileweb.craigsphone) appear to bundle the IMEI in search queries; in a travel application (com.visualit.tubeLondonCity), the method *refreshLiveInfo()* includes the IMEI in a URL; and a "keyring" application (com.froogloid.kring.google.zxing.client.android) appends the IMEI to a variable named *retailerLookupCmd*. We also found functionality that includes the IMEI when checking for updates (e.g., com.webascender.callerid, which also includes the phone number) and retrieving advertisements (see Finding 6). Furthermore, we found two applications (com.taobo.tao and raker.duobao.store) with network access wrapper methods that include the IMEI for all connections. These behaviors indicate that the IMEI is used as a form of "tracking cookie".

Finding 4 - *The IMEI is tied to personally identifiable information (PII).* The common belief that the IMEI to phone owner mapping is not visible outside the cellular network is no longer true. In several cases, we found code that bound the IMEI to account information and other PII. For example, applications (e.g. com.slacker.radio and com.statefarm.pocketagent) include the IMEI in account registration and login requests. In another application (com.amazon.mp3), the method *linkDevice()* includes the IMEI. Code inspection indicated that this method is called when the user chooses to "Enter a claim code" to redeem gift cards. We also found IMEI use in code for sending comments and reporting problems (e.g., com.morbe.guarder and com.fm207.discount). Finally, we found one application (com.andoop.highscore) that appears to bundle the IMEI when submitting high scores for games. Thus, it seems clear that databases containing mappings between physical users and IMEIs are being created.

Finding 5 - *Not all phone identifier use leads to exfiltration.* Several applications that access phone identifiers did not exfiltrate the values. For example, one application (com.amazon.kindle) creates a device fingerprint for a verification check. The fingerprint is kept in "secure storage" and does not appear to leave the phone. Another application (com.match.android.matchmobile) assigns the phone number to a text field used for account registration. While the value is sent to the network during registration, the user can easily change or remove it.

Finding 6 - *Phone identifiers are sent to advertisement and analytics servers.* Many applications have custom ad and analytics functionality. For example, in one application (com.accuweather.android), the class *ACCUWX_AdRequest* is an IMEI data flow sink. Another application (com.amazon.mp3) defines Android service component *AndroidMetricsManager*, which is an IMEI data flow sink. Phone identifier data flows also occur in ad libraries. For example, we found a phone number data flow sink in the `com/wooboo/adlib_android` library used by several applications (e.g., cn.ecook, com.superdroid.sqd, and com.superdroid.ewc). Section 6.3 discusses ad libraries in more detail.

6.1.2 Location Information

Location information is accessed in two ways: (1) calling *getLastKnownLocation()*, and (2) defining callbacks in a *LocationListener* object passed to *requestLocationUpdates()*. Due to code recovery failures, not all *LocationListener* objects have corresponding *requestLocationUpdates()* calls. We scanned for all three constructs.

Table 4: Access of Location APIs

Identifier	**# Uses**	**# Apps**	**# w/ Perm.***
getLastKnownLocation	428	204	148
LocationListener	652	469	282
requestLocationUpdates	316	146	128
Total Unique	-	505	304†

* Defined as having a `LOCATION` permission.

† In total, 5 apps did not also have the `INTERNET` permission.

Table 4 summarizes the access of location information. In total, 505 applications (45.9%) attempt to access location, only 304 (27.6%) have the permission to do so. This difference is likely due to libraries that probe for permissions, as discussed in Section 6.3. The separation between *LocationListener* and *requestLocationUpdates()* is primarily due to the AdMob library, which defined the former but has no calls to the latter.

Table 3 shows detected location data flows to the network. To overcome missing code challenges, the data flow source was defined as the *getLatitude()* and *getLongitude()* methods of the *Location* object retrieved from the location APIs. We manually inspected the 13 applications with location data flows. Many data flows appeared to reflect legitimate uses of location for weather, classifieds, points of interest, and social networking services. Inspection of the remaining applications informs the following findings:

Finding 7 - *The granularity of location reporting may not always be obvious to the user.* In one application (com.andoop.highscore) both the city/country *and* geographic coordinates are sent along with high scores. Users may be aware of regional geographic information associated with scores, but it was unclear if users are aware that precise coordinates are also used.

Finding 8 - *Location information is sent to advertisement servers.* Several location data flows appeared to terminate in network connections used to retrieve ads. For example, two applications (com.avantar.wny and com.avantar.yp) appended the location to the variable *webAdURLString*. Motivated by [14], we inspected the AdMob library to determine why no data flow was found and determined that source code recovery failures led to the false negatives. Section 6.3 expands on ad libraries.

6.2 Phone Misuse

This section explores misuse of the smartphone interfaces, including telephony services, background recording of audio and video, sockets, and accessing the list of installed applications.

6.2.1 Telephony Services

Smartphone malware can provide direct compensation using phone calls or SMS messages to premium-rate numbers [18, 25]. We defined three queries to identify such malicious behavior: (1) a constant used for the SMS destination number; (2) creation of *URI* objects with a "`tel:`" prefix (used for phone call intent messages) and the string "900" (a premium-rate number prefix in the US); and (3) any *URI* objects with a "`tel:`" prefix. The analysis informs the following findings.

Finding 9 - *Applications do not appear to be using fixed phone number services.* We found zero applications using a constant destination number for the SMS API. Note that our analysis specification is limited to constants passed directly to the API and *final* variables, and therefore may have false negatives. We found two applications creating *URI* objects with the "`tel:`" prefix and containing the string "900". One application included code to call "`tel://0900-9292`", which is a premium-rate number (€0.70 per minute) for travel advice in the Netherlands. However, this did not appear malicious, as the application (com.Planner9292) is designed to provide travel advice. The other application contained several hard-coded numbers with "900" in the last four digits of the number. The SMS and premium-rate analysis results are promising indicators for non-existence of malicious behavior. Future analysis should consider more premium-rate prefixes.

Finding 10 - *Applications do not appear to be misusing voice services.* We found 468 *URI* objects with the "`tel:`" prefix in 358 applications. We manually inspected a sample of applications to better understand phone number use. We found: (1) applications frequently include call functionality for customer service; (2) the "`CALL`" and "`DIAL`" intent actions were used equally for the same purpose (`CALL` calls immediately and requires the `CALL_PHONE` permission, whereas `DIAL` has user confirmation the dialer and requires no permission); and (3) not all hard-coded telephone numbers are used to make phone calls, e.g., the AdMob library had a apparently unused phone number hard coded.

6.2.2 Background Audio/Video

Microphone and camera eavesdropping on smartphones is a real concern [41]. We analyzed application eavesdropping behaviors, specifically: (1) recording video without calling *setPreviewDisplay()* (this API is always required for still image capture); (2) *AudioRecord.read()* in code not reachable from an Android activity component; and (3) *MediaRecorder.start()* in code not reachable from an activity component.

Finding 11 - *Applications do not appear to be misusing video recording.* We found no applications that record video without calling *setPreviewDisplay()*. The query reasonably did not consider the value passed to the preview display, and therefore may create false negatives. For example, the "preview display" might be one pixel in size. The *MediaRecorder.start()* query detects audio recording, but it also detects video recording. This query found two applications using video in code not reachable from an activity; however the classes extended *SurfaceView*, which is used by *setPreviewDisplay()*.

Finding 12 - *Applications do not appear to be misusing audio recording.* We found eight uses in seven applications of *AudioRecord.read()* without a control flow path to an activity component. Of these applications, three provide VoIP functionality, two are games that repeat what the user says, and one provides voice search. In these applications, audio recording is expected; the lack of reachability was likely due to code recovery failures. The remaining application did not have the required `RECORD_AUDIO` permission and the code most likely was part of a developer toolkit. The *MediaRecorder.start()* query identified an additional five applications recording audio without reachability to an activity. Three of these applications have legitimate reasons to record audio: voice search, game interaction, and VoIP. Finally, two games included audio recording in a developer toolkit, but no record permission, which explains the lack of reachability. Section 6.3.2 discusses developer toolkits.

6.2.3 Socket API Use

Java sockets represent an open interface to external services, and thus are a potential source of malicious behavior. For example, smartphone-based botnets have been found to exist on "jailbroken" iPhones [8]. We observe that most Internet-based smartphone applications are HTTP clients. Android includes useful classes (e.g., *HttpURLConnection* and *HttpClient*) for communicating with Web servers. Therefore, we queried for applications that make network connections using the *Socket* class.

Finding 13 - *A small number of applications include code that uses the* Socket *class directly.* We found 177 *Socket* connections in 75 applications (6.8%). Many applications are flagged for inclusion of well-known network libraries such as `org/apache/thrift`, `org/apache/commons`, and `org/eclipse/jetty`, which use sockets directly. Socket factories were also detected. Identified factory names such as *TrustAllSSLSocketFactory*, *AllTrustSSLSocketFactory*, and *NonValidatingSSLSocketFactory* are interesting as potential vulnerabilities, but we found no evidence of malicious use. Several applications also included their own HTTP wrapper methods that duplicate functionality in the Android libraries, but did not appear malicious. Among the applications including custom network connection wrappers is a group of applications in the "Finance" category implementing cryptographic network protocols (e.g., in the `com/lumensoft/ks` library). We note that these applications use Asian character sets for their market descriptions, and we could not determine their exact purpose.

Finding 14 - *We found no evidence of malicious behavior by applications using* Socket *directly.* We manually inspected all 75 applications to determine if *Socket* use seemed appropriate based on the application description. Our survey yielded a diverse array of *Socket* uses, including: file transfer protocols, chat protocols, audio and video streaming, and network connection tethering, among other uses excluded for brevity. A handful of applications have socket connections to hard-coded IP address and non-standard ports. For example, one application (com.eingrad.vintagecomicdroid) downloads comics from 208.94.242.218 on port 2009. Additionally, two of the aforementioned financial applications (com.miraeasset.mstock and kvp.jjy.MispAndroid320) include the `kr/co/shiftworks` library that connects to 221.143.48.118 on port 9001. Furthermore, one application (com.tf1.lci) connects to 209.85.227.147 on port 80 in a class named *AdService* and subsequently calls *getLocalAddress()* to retrieve the phone's IP address. Overall, we found no evidence of malicious behavior, but several applications warrant deeper investigation.

6.2.4 Installed Applications

The list of installed applications provides valuable marketing data. Android has two relevant APIs types: (1) a set of *get* APIs returning the list of installed applications or package names; and (2) a set of *query* APIs that mirrors Android's runtime intent resolution, but can be made generic. We found 54 uses of the get APIs in 45 applications, and 1015 uses of the query APIs in 361 applications. Sampling these applications, we observe:

Finding 15 - *Applications do not appear to be harvesting information about which applications are installed on the phone.* In all but two cases, the sampled applications using the get APIs search the results for a specific application. One application (com.davidgoemans.simpleClockWidget) defines a method that returns the list of all installed applications, but the results were only displayed to the user. The second application (raker.duobao.store) defines a similar method, but it only appears to be called by unused debugging code. Our survey of the query APIs identified three calls within the AdMob library duplicated in many applications. These uses queried specific functionality and thus are not likely to harvest application information. The one non-AdMob application we inspected queried for specific functionality, e.g., speech recognition, and thus did not appear to attempt harvesting.

Table 5: Identified Ad and Analytics Library Paths

Library Path	**# Apps**	**Format**	**Obtains**[*]
com/admob/android/ads	320	Obf.	L
com/google/ads	206	Plain	-
com/flurry/android	98	Obf.	-
com/qwapi/adclient/android	74	Plain	L, P, E
com/google/android/apps/analytics	67	Plain	-
com/adwhirl	60	Plain	L
com/mobclix/android/sdk	58	Plain	L, E[‡]
com/millennialmedia/android	52	Plain	-
com/zestadz/android	10	Plain	-
com/admarvel/android/ads	8	Plain	-
com/estsoft/adlocal	8	Plain	L
com/adfonic/android	5	Obf.	-
com/vdroid/ads	5	Obf.	L, E
com/greystripe/android/sdk	4	Obf.	E
com/medialets	4	Obf.	L
com/wooboo/adlib_android	4	Obf.	L, P, I[†]
com/adserver/adview	3	Obf.	L
com/tapjoy	3	Plain	-
com/inmobi/androidsdk	2	Plain	E[‡]
com/apegroup/ad	1	Plain	-
com/casee/adsdk	1	Plain	S
com/webtrends/mobile	1	Plain	L, E, S, I
Total Unique Apps	561	-	-

[*] L = Location; P = Phone number; E = IMEI; S = IMSI; I = ICC-ID
[†] In 1 app, the library included "L", while the other 3 included "P, I".
[‡] Direct API use not decompiled, but wrapper *.getDeviceId()* called.

6.3 Included Libraries

Libraries included by applications are often easy to identify due to namespace conventions: i.e., the source code for com.foo.appname typically exists in `com/foo/appname`. During our manual inspection, we documented advertisement and analytics library paths. We also found applications sharing what we term "developer toolkits," i.e., a common set of developer utilities.

6.3.1 Advertisement and Analytics Libraries

We identified 22 library paths containing ad or analytics functionality. Sampled applications frequently contained multiple of these libraries. Using the paths listed in Table 5, we found: 1 app has 8 libraries; 10 apps have 7 libraries; 8 apps have 6 libraries; 15 apps have 5 libraries; 37 apps have 4 libraries; 32 apps have 3 libraries; 91 apps have 2 libraries; and 367 apps have 1 library.

Table 5 shows advertisement and analytics library use. In total, at least 561 applications (51%) include these libraries; however, additional libraries may exist, and some applications include custom ad and analytics functionality. The AdMob library is used most pervasively, existing in 320 applications (29.1%). Google Ads is used by 206 applications (18.7%). We observe from Table 5 that only a handful of libraries are used pervasively.

Several libraries access phone identifier and location APIs. Given the library purpose, it is easy to speculate data flows to network APIs. However, many of these flows were not detected by program analysis. This is (likely) a result of code recovery failures and flows through Android IPC. For example, AdMob has known location to network data flows [14], and we identified a code recovery failure for the class implementing that functionality. Several libraries are also obfuscated, as mentioned in Section 7. Interesting, 6 of the 13 libraries accessing sensitive information are obfuscated. The analysis informs the following additional findings.

Finding 16 - *Ad and analytics library use of phone identifiers and location is sometimes configurable.* The `com/webtrends/mobile` analytics library (used by com.statefarm.pocketagent), defines the *WebtrendsIdMethod* class specifying four identifier types. Only one type, "*system_id_extended*" uses phone identifiers (IMEI, IMSI, and ICC-ID). It is unclear which identifier type was used by the application. Other libraries provide similar configuration. For example, the AdMob SDK documentation [6] indicates that location information is only included if a package manifest configuration enables it.

Finding 17 - *Analytics library reporting frequency is often configurable.* During manual inspection, we encountered one application (com.handmark.mpp.news.reuters) in which the phone number is passed to *FlurryAgent.onEvent()* as generic data. This method is called throughout the application, specifying event labels such as "GetMoreStories," "StoryClickedFromList," and "ImageZoom." Here, we observe the main application code not only specifies the phone number to be reported, but also report frequency.

Finding 18 - *Ad and analytics libraries probe for permissions.* The `com/webtrends/mobile` library accesses the IMEI, IMSI, ICC-ID, and location. The (*WebtrendsAndroidValueFetcher*) class uses try/catch blocks that catch the *SecurityException* that is thrown when an application does not have the proper permission. Similar functionality exists in the `com/casee/adsdk` library (used by com.fish.luny). In *AdFetcher.getDeviceId()*, Android's *checkCallingOrSelfPermission()* method is evaluated before accessing the IMSI.

6.3.2 Developer Toolkits

Several inspected applications use developer toolkits containing common sets of utilities identifiable by class name or library path. We observe the following.

Finding 19 - *Some developer toolkits replicate dangerous functionality.* We found three wallpaper applications by developer "callmejack" that include utilities in the library path `com/jackeeywu/apps/eWallpaper` (com.eoeandroid.eWallpapers.cartoon, com.jackeey.wallpapers.all1.orange, and com.jackeey.eWallpapers.gundam). This library has data flow sinks for the phone number, IMEI, IMSI, and ICC-ID. In July 2010, Lookout, Inc. reported a wallpaper application by developer "jackeey,wallpaper" as sending these identifiers to `imnet.us` [29]. This report also indicated that the developer changed his name to "callmejack". While the original "jackeey,wallpaper" application was removed from the Android Market, the applications by "callmejack" remained as of September 2010.[3]

Finding 20 - *Some developer toolkits probe for permissions.* In one application (com.july.cbssports.activity), we found code in the `com/julysystems` library that evaluates Android's *checkPermission()* method for the `READ_PHONE_STATE` and `ACCESS_FINE_LOCATION` permissions before accessing the IMEI, phone number, and last known location, respectively. A second application (v00032.com.wordplayer) defines the *CustomExceptionHander* class to send an exception event to an HTTP URL. The class attempts to retrieve the phone number within a try/catch block, catching a generic *Exception*. However, the application does not have the `READ_PHONE_STATE` permission, indicating the class is likely used in multiple applications.

Finding 21 - *Well-known brands sometimes commission developers that include dangerous functionality.* The `com/julysystems` developer toolkit identified as probing for permissions exists in two applications with reputable application providers. "CBS Sports Pro Football" (com.july.cbssports.activity) is provided by "CBS Interactive, Inc.", and "Univision Fütbol" (com.july.univision) is provided by "Univision Interactive Media, Inc.". Both have location and phone state permissions, and hence potentially misuse information.

Similarly, "USA TODAY" (com.usatoday.android.news) provided by "USA TODAY" and "FOX News" (com.foxnews.android) provided by "FOX News Network, LLC" contain the `com/mercuryintermedia` toolkit. Both applications contain an Android activity component named *MainActivity*. In the initialization phase, the IMEI is retrieved and passed to *ProductConfiguration.initialize()* (part of the `com/mecuryintermedia` toolkit). Both applications have IMEI to network data flows through this method.

6.4 Android-specific Vulnerabilities

This section explores Android-specific vulnerabilities.

6.4.1 Leaking Information to Logs

Android provides centralized logging via the *Log* API, which can displayed with the "`logcat`" command. While `logcat` is a debugging tool, applications with the `READ_LOGS` permission can read these log messages. The Android documentation for this permission indicates that "[the logs] can contain slightly private information about

[3]Fortunately, these dangerous applications are now nonfunctional, as the `imnet.us` NS entry is `NS1.SUSPENDED-FOR.SPAM-AND-ABUSE.COM`.

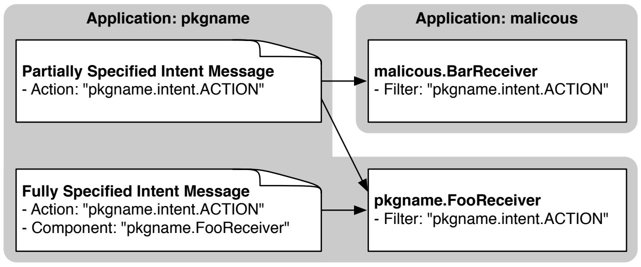
Application: pkgname
Application: malicous
Partially Specified Intent Message
- Action: "pkgname.intent.ACTION"
malicous.BarReceiver
- Filter: "pkgname.intent.ACTION"
Fully Specified Intent Message
- Action: "pkgname.intent.ACTION"
- Component: "pkgname.FooReceiver"
pkgname.FooReceiver
- Filter: "pkgname.intent.ACTION"

Figure 14: Eavesdropping on unprotected intents

what is happening on the device, but should never contain the user's private information." We looked for data flows from phone identifier and location APIs to the Android logging interface and found the following.

Finding 22 - *Private information is written to Android's general logging interface.* We found 253 data flows in 96 applications for location information, and 123 flows in 90 applications for phone identifiers. Frequently, URLs containing this private information are logged just before a network connection is made. Thus, the `READ_LOGS` permission allows access to private information.

6.4.2 Leaking Information via IPC

Shown in Figure 14, any application can receive intent broadcasts that do not specify the target component or protect the broadcast with a permission (permission variant not shown). This is unsafe if the intent contains sensitive information. We found 271 such unsafe intent broadcasts with "extras" data in 92 applications (8.4%). Sampling these applications, we found several such intents used to install shortcuts to the home screen.

Finding 23 - *Applications broadcast private information in IPC accessible to all applications.* We found many cases of applications sending unsafe intents to action strings containing the application's namespace (e.g., "pkgname.intent.ACTION" for application pkgname). The contents of the bundled information varied. In some instances, the data was not sensitive, e.g., widget and task identifiers. However, we also found sensitive information. For example one application (com.ulocate) broadcasts the user's location to the "com.ulocate.service.LOCATION" intent action string without protection. Another application (com.himsn) broadcasts the instant messaging client's status to the "cm.mz.stS" action string. These vulnerabilities allow malicious applications to eavesdrop on sensitive information in IPC, and in some cases, gain access to information that requires a permission (e.g., location).

6.4.3 Unprotected Broadcast Receivers

Applications use broadcast receiver components to receive intent messages. Broadcast receivers define "intent filters" to subscribe to specific event types are public. If the receiver is not protected by a permission, a malicious application can forge messages.

Finding 24 - *Few applications are vulnerable to forging attacks to dynamic broadcast receivers.* We found 406 unprotected broadcast receivers in 154 applications (14%). We found an large number of receivers subscribed to system defined intent types. These receivers are indirectly protected by Android's "protected broadcasts" introduced to eliminate forging. We found one application with an unprotected broadcast receiver for a custom intent type; however it appears to have limited impact. Additional sampling may uncover more cases.

6.4.4 Intent Injection Attacks

Intent messages are also used to start activity and service components. An intent injection attack occurs if the intent address is derived from untrusted input.

We found 10 data flows from the network to an intent address in 1 application. We could not confirm the data flow and classify it a false positive. The data flow sink exists in a class named *ProgressBroadcasting-FileInputStream*. No decompiled code references this class, and all data flow sources are calls to *URLConnection.getInputStream()*, which is used to create *InputStreamReader* objects. We believe the false positives results from the program analysis modeling of classes extending *InputStream*.

We found 80 data flows from IPC to an intent address in 37 applications. We classified the data flows by the sink: the *Intent* constructor is the sink for 13 applications; *setAction()* is the sink for 16 applications; and *setComponent()* is the sink for 8 applications. These sets are disjoint. Of the 37 applications, we found that 17 applications set the target component class explicitly (all except 3 use the *setAction()* data flow sink), e.g., to relay the action string from a broadcast receiver to a service. We also found four false positives due to our assumption that all *Intent* objects come from IPC (a few exceptions exist). For the remaining 16 cases, we observe:

Finding 25 - *Some applications define intent addresses based on IPC input.* Three applications use IPC input strings to specify the package and component names for the *setComponent()* data flow sink. Similarly, one application uses the IPC "extras" input to specify an action to an *Intent* constructor. Two additional applications start an activity based on the action string returned as a result from a previously started activity. However, to exploit this vulnerability, the applications must first start a malicious activity. In the remaining cases, the action string used to start a component is copied directly into a new intent object. A malicious application can exploit this vulnerability by specifying the vulnerable component's name directly and controlling the action string.

6.4.5 Delegating Control

Applications can delegate actions to other applications using a "pending intent." An application first creates an intent message as if it was performing the action. It then creates a reference to the intent based on the target component type (restricting how it can be used). The pending intent recipient cannot change values, but it can fill in missing fields. Therefore, if the intent address is unspecified, the remote application can redirect an action that is performed with the original application's permissions.

Finding 26 - *Few applications unsafely delegate actions.* We found 300 unsafe pending intent objects in 116 applications (10.5%). Sampling these applications, we found an overwhelming number of pending intents used for either: (1) Android's UI notification service; (2) Android's alarm service; or (3) communicating between a UI widget and the main application. None of these cases allow manipulation by a malicious application. We found two applications that send unsafe pending intents via IPC. However, exploiting these vulnerabilities appears to provides negligible adversarial advantage. We also note that more a more sophisticated analysis framework could be used to eliminate the aforementioned false positives.

6.4.6 Null Checks on IPC Input

Android applications frequently process information from intent messages received from other applications. Null dereferences cause an application to crash, and can thus be used to as a denial of service.

Finding 27 - *Applications frequently do not perform null checks on IPC input.* We found 3,925 potential null dereferences on IPC input in 591 applications (53.7%). Most occur in classes for activity components (2,484 dereferences in 481 applications). Null dereferences in activity components have minimal impact, as the application crash is obvious to the user. We found 746 potential null dereferences in 230 applications within classes defining broadcast receiver components. Applications commonly use broadcast receivers to start background services, therefore it is unclear what effect a null dereference in a broadcast receiver will have. Finally, we found 72 potential null dereferences in 36 applications within classes defining service components. Applications crashes corresponding to these null dereferences have a higher probability of going unnoticed. The remaining potential null dereferences are not easily associated with a component type.

6.4.7 SDcard Use

Any application that has access to read or write data on the SDcard can read or write any other application's data on the SDcard. We found 657 references to the SDcard in 251 applications (22.8%). Sampling these applications, we found a few unexpected uses. For example, the `com/tapjoy` ad library (used by com.jnj.mocospace.android) determines the free space available on the SDcard. Another application (com.rent) obtains a URL from a file named `connRentInfo.dat` at the root of the SDcard.

6.4.8 JNI Use

Applications can include functionality in native libraries using the Java Native Interface (JNI). As these methods are not written in Java, they have inherent dangers. We found 2,762 calls to native methods in 69 applications (6.3%). Investigating the application package files, we found that 71 applications contain `.so` files. This indicates two applications with an `.so` file either do not call any native methods, or the code calling the native methods was not decompiled. Across these 71 applications, we found 95 `.so` files, 82 of which have unique names.

6.5 General Application Vulnerabilities

We analyzed the application source code for general Java application vulnerabilities based on industry-standard criteria [2]. Many of the criteria are irrelevant due to either artifacts of the decompilation process (e.g., variables including the "$" character) or clearly irrelevant to Android (e.g., J2EE vulnerabilities). For the applicable criteria, we identified 5,325 issues. A breakdown of the number of instances and affected applications is shown in Table 6. We observe that analysis results are unevenly distributed. Only 564 out of 1,100 application had potential vulnerabilities detected by the general criteria. Further, less than a third of the 564 applications—16.91% of the 1,100—account for over 82% of the detected code locations.

6.5.1 Password Misuse

Finding 28 - *Few applications have hard-coded or empty passwords.* We only found eight applications with hard-coded passwords. Two of these applications rely on a Twitter and an ICQ login. In a library common to two "Finance" applications, unique username and password pairs are specified to authenticate to the same hard-coded IP. The purpose of this code was unclear. The fifth application uses a hard-coded password to authenticate custom error reporting. The sixth application contained a hard-coded password for a demo account. Finally, in the last two applications, the same GMail username and password is used in a class named *SendTest*. All of the other sampled code locations for hard-coded or empty passwords are initializers (e.g., "changeit") in library code.

Finding 29 - *We found no evidence of plaintext passwords written to file.* All of the sampled code lo-

Table 6: Source code analysis results for general application vulnerabilities.

Application	Lines	Password Mgmt		Cryptography		Injection		All Flag Types	
Category	**of Code**	**# Flags**	**# Apps**	**# Flags**	**# Apps**	**# Flags**	**# Apps**	**# Flags**	**# Apps**
Comics	415,625	0	0	32	10	37	5	69	13
Communication	1,832,514	96	16	119	26	608	26	823	34
Demo	830,471	2	2	21	7	10	2	33	7
Entertainment	709,915	6	3	55	16	90	11	151	22
Finance	709,915	105	8	96	21	91	12	292	26
Games (Arcade)	766,045	2	1	390	27	1	1	393	27
Games (Puzzle)	727,642	1	1	124	31	17	7	142	35
Games (Casino)	985,423	15	3	111	36	6	3	132	38
Games (Casual)	681,429	3	2	129	32	6	4	138	32
Health	847,511	37	11	196	21	85	6	318	26
Lifestyle	778,446	4	2	70	15	85	12	159	22
Multimedia	1,323,805	77	4	186	26	335	23	598	32
News/Weather	1,123,674	4	2	55	15	62	11	121	20
Productivity	1,443,600	26	7	108	18	140	21	274	31
Reference	887,794	4	1	93	25	42	12	139	30
Shopping	1,371,351	14	4	85	23	111	13	210	27
Social	2,048,177	59	12	155	37	478	29	692	41
Libraries	182,655	3	2	9	5	110	11	122	13
Sports	651,881	1	1	17	7	31	11	49	18
Themes	310,203	0	0	95	23	9	4	104	24
Tools	839,866	3	2	56	17	86	20	145	26
Travel	1,419,783	65	8	76	14	80	12	221	20
TOTAL	**21,734,202**	**527**	**92**	**2,278**	**452**	**2,520**	**259**	**5,325**	**564**

cations identified as writing plaintext passwords to file are false positives. Most studied code locations exist in unused library code (e.g., obtaining a password from *stdin.readline()*, which cannot be used in Android). Note that the Wells Fargo application recently identified as writing a plaintext password to file [43] did not appear to have this functionality in the recovered source code.

6.5.2 Cryptography Misuse

Finding 30 - *Some applications use unsafe cryptographic keys and algorithms.* We found 7 applications including the toolkit class *SecurityUtil* that uses a variable containing the "device ID" (most likely the IMEI) as a DES encryption key. If the device ID is not available, a hard-coded constant is used. We found several other uses of DES; however these are in the NTLM implementation of an Apache library. It is unclear if this functionality is used.

Finding 31 - *Few applications use insufficient key size.* We found two applications (com.scfirstbank and edaily.daishin) using RSA with a 1024-bit modulus. A third application (com.slacker.radio) uses RSA with a 512-bit modulus. Several additional instances exist in libraries.

Finding 32 - *We found no evidence of PRNG misuse.* Insecure randomness accounts for 1,681 of the 2,278 flags in the cryptography category in Table 6. Our sampling found a library for one application (com.scfirstbank) where a non-cryptographic PRNG is selected only if the requested cryptographic one is not available; however the method did not appear to be used. Finally, the analysis looks for all PRNG uses, most of which are not cryptographically related (e.g., selecting keywords for ad targeting).

6.5.3 Injection Vulnerabilities

Finding 33 - *Few applications have path manipulation vulnerabilities.* Path manipulation consisted of 1,228 of the 2,520 flagged code locations in the injection category. Most of the sampled cases existed within a *main()* method of a library (e.g., for Base64 encoding/decoding); however, Android does not execute *main()* methods. Android programming conventions also led to false positives. For example, content provider components that share files commonly store the filename in an SQLite database. This filename was detected as untrusted. We only found one application with a path manipulation vulnerability. In this case, a filename is based on values received from an intent message; however, the conditions under which the vulnerability can be exploited are unclear.

Finding 34 - *We found no evidence of database or com-*

mand injection vulnerabilities. All sampled flags for database injection are false positives. We speculate the non-existence of this vulnerability is the widespread use of parameterized SQL queries in Android.

7 Study Limitations

Our study section was limited in three ways: *a*) the studied applications were selected with a bias towards popularity; *b*) the program analysis tool cannot compute data and control flows for IPC between components; and *c*) source code recovery failures interrupt data and control flows. Missing data and control flows may lead to false negatives. In addition to the recovery failures, the program analysis tool could not parse 8,042 classes, reducing coverage to 91.34% of the classes.

Additionally, a portion of the recovered source code was obfuscated before distribution. Code obfuscation significantly impedes manual inspection. It likely exists to protect intellectual property; Google suggests obfuscation using ProGuard (`proguard.sf.net`) for applications using its licensing service [23]. ProGuard protects against readability and does not obfuscate control flow. Therefore it has limited impact on program analysis.

Many forms of obfuscated code are easily recognizable: e.g., class, method, and field names are converted to single letters, producing single letter Java filenames (e.g., `a.java`). For a rough estimate on the use of obfuscation, we searched applications containing `a.java`. In total, 396 of the 1,100 applications contain this file. As discussed in Section 6.3, several advertisement and analytics libraries are obfuscated. To obtain a closer estimate of the number of applications whose main code is obfuscated, we searched for `a.java` within a file path equivalent to the package name (e.g., `com/foo/appname` for com.foo.appname). Only 20 applications (1.8%) have this obfuscation property, which is expected for free applications (as opposed to paid applications). However, we stress that the `a.java` heuristic is not intended to be a firm characterization of the percentage of obfuscated code, but rather a means of acquiring insight.

8 What This All Means

Identifying a singular take-away from a broad study such as this is non-obvious. We come away from the study with two central thoughts; one having to do with the study apparatus, and the other regarding the applications.

`ded` and the program analysis specifications are enabling technologies that open a new door for application certification. We found the approach rather effective despite existing limitations. In addition to further studies of this kind, we see the potential to integrate these tools into an application certification process. We leave such discussions for future work, noting that such integration is challenging for both logistical and technical reasons [30].

On a technical level, we found the security characteristics of the top 1,100 free popular applications to be consistent with smaller studies (e.g., Enck et al. [14]). Our findings indicate an overwhelming concern for misuse of privacy sensitive information such as phone identifiers and location information. One might speculate this occur due to the difficulty in assigning malicious intent.

Arguably more important than identifying the existence the information misuse, our manual source code inspection sheds more light on *how* information is misused. We found phone identifiers, e.g., phone number, IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to account numbers. Our findings also support the existence of databases external to cellular providers that link identifiers such as the IMEI to personally identifiable information.

Our analysis also identified significant penetration of ad and analytic libraries, occurring in 51% of the studied applications. While this might not be surprising for free applications, the number of ad and analytics libraries included per application was unexpected. One application included as many as eight different libraries. It is unclear why an application needs more than one advertisement and one analytics library.

From a vulnerability perspective, we found that many developers fail to take necessary security precautions. For example, sensitive information is frequently written to Android's centralized logs, as well as occasionally broadcast to unprotected IPC. We also identified the potential for IPC injection attacks; however, no cases were readily exploitable.

Finally, our study only characterized one edge of the application space. While we found no evidence of telephony misuse, background recording of audio or video, or abusive network connections, one might argue that such malicious functionality is less likely to occur in popular applications. We focused our study on popular applications to characterize those most frequently used. Future studies should take samples that span application popularity. However, even these samples may miss the existence of truly malicious applications. Future studies should also consider several additional attacks, including installing new applications [44], JNI execution [34], address book exfiltration, destruction of SDcard contents, and phishing [20].

9 Related Work

Many tools and techniques have been designed to identify security concerns in software. Software written in C is particularly susceptible to programming errors that result in vulnerabilities. Ashcraft and Engler [7] use compiler extensions to identify errors in range checks.

MOPS [11] uses model checking to scale to large amounts of source code [42]. Java applications are inherently safer than C applications and avoid simple vulnerabilities such as buffer overflows. Ware and Fox [47] compare eight different open source and commercially available Java source code analysis tools, finding that no one tool detects all vulnerabilities. Hovemeyer and Pugh [22] study six popular Java applications and libraries using FindBugs extended with additional checks. While analysis included non-security bugs, the results motivate a strong need for automated analysis by all developers. Livshits and Lam [28] focus on Java-based Web applications. In the Web server environment, inputs are easily controlled by an adversary, and left unchecked can lead to SQL injection, cross-site scripting, HTTP response splitting, path traversal, and command injection. Felmetsger et al. [19] also study Java-based web applications; they advance vulnerability analysis by providing automatic detection of application-specific logic errors.

Spyware and privacy breaching software have also been studied. Kirda et al. [26] consider behavioral properties of BHOs and toolbars. Egele et al. [13] target information leaks by browser-based spyware explicitly using dynamic taint analysis. Panaorama [48] considers privacy-breaching malware in general using whole-system, fine-grained taint tracking. Privacy Oracle [24] uses differential black box fuzz testing to find privacy leaks in applications.

On smartphones, TaintDroid [14] uses system-wide dynamic taint tracking to identify privacy leaks in Android applications. By using static analysis, we were able to study a far greater number of applications (1,100 vs. 30). However, TaintDroid's analysis confirms the exfiltration of information, while our static analysis only confirms the potential for it. Kirin [16] also uses static analysis, but focuses on permissions and other application configuration data, whereas our study analyzes source code. Finally, PiOS [12] performs static analysis on iOS applications for the iPhone. The PiOS study found the majority of analyzed applications to leak the device ID and over half of the applications include advertisement and analytics libraries.

10 Conclusions

Smartphones are rapidly becoming a dominant computing platform. Low barriers of entry for application developers increases the security risk for end users. In this paper, we described the ded decompiler for Android applications and used decompiled source code to perform a breadth study of both dangerous functionality and vulnerabilities. While our findings of exposure of phone identifiers and location are consistent with previous studies, our analysis framework allows us to observe not only the existence of dangerous functionality, but also how it occurs within the context of the application.

Moving forward, we foresee ded and our analysis specifications as enabling technologies that will open new doors for application certification. However, the integration of these technologies into an application certification process requires overcoming logistical and technical challenges. Our future work will consider these challenges, and broaden our analysis to new areas, including application installation, malicious JNI, and phishing.

Acknowledgments

We would like to thank Fortify Software Inc. for providing us with a complementary copy of Fortify SCA to perform the study. We also thank Suneel Sundar and Joy Marie Forsythe at Fortify for helping us debug custom rules. Finally, we thank Kevin Butler, Stephen McLaughlin, Patrick Traynor, and the SIIS lab for their editorial comments during the writing of this paper. This material is based upon work supported by the National Science Foundation Grant No. CNS-0905447, CNS-0721579, and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Fernflower - java decompiler. `http://www.reversed-java.com/fernflower/`.

[2] Fortify 360 Source Code Analyzer (SCA). `https://www.fortify.com/products/fortify360/source-code-analyzer.html`.

[3] Jad. `http://www.kpdus.com/jad.html`.

[4] Jd java decompiler. `http://java.decompiler.free.fr/`.

[5] Mocha, the java decompiler. `http://www.brouhaha.com/~eric/software/mocha/`.

[6] ADMOB. AdMob Android SDK: Installation Instructions. `http://www.admob.com/docs/AdMob_Android_SDK_Instructions.pdf`. Accessed November 2010.

[7] ASHCRAFT, K., AND ENGLER, D. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy* (2002).

[8] BBC NEWS. New iPhone worm can act like botnet say experts. `http://news.bbc.co.uk/2/hi/technology/8373739.stm`, November 23, 2009.

[9] BORNSTEIN, D. Google i/o 2008 - dalvik virtual machine internals. `http://www.youtube.com/watch?v=ptjedOZEXPM`.

[10] BURNS, J. Developing Secure Mobile Applications for Android. iSEC Partners, October 2008. `http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf`.

[11] CHEN, H., DEAN, D., AND WAGNER, D. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium* (Feb. 2004).

[12] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[13] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. Dynamic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference* (June 2007), pp. 233–246.

[14] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2010).

[15] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA, August 2011).

[16] Enck, W., Ongtang, M., and McDaniel, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).

[17] Enck, W., Ongtang, M., and McDaniel, P. Understanding Android Security. *IEEE Security & Privacy Magazine 7*, 1 (January/February 2009), 50–57.

[18] F-Secure Corporation. Virus Description: Viver.A. `http://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml`.

[19] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (2010).

[20] First Tech Credit Union. Security Fraud: Rogue Android Smartphone app created. `http://www.firsttechcu.com/home/security/fraud/security_fraud.html`, Dec. 2009.

[21] Goodin, D. Backdoor in top iphone games stole user data, suit claims. The Register, November 2009. `http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/`.

[22] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages, and Applications* (2004).

[23] Johns, T. Securing Android LVL Applications. `http://android-developers.blogspot.com/2010/09/securing-android-lvl-applications.html`, 2010.

[24] Jung, J., Sheth, A., Greenstein, B., Wetherall, D., Maganis, G., and Kohno, T. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the ACM conference on Computer and Communications Security* (2008).

[25] Kasperskey Lab. First SMS Trojan detected for smartphones running Android. `http://www.kaspersky.com/news?id=207576158`, August 2010.

[26] Kirda, E., Kruegel, C., Banks, G., Vigna, G., and Kemmerer, R. A. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium* (Aug. 2006).

[27] Kralevich, N. Best Practices for Handling Android User Data. `http://android-developers.blogspot.com/2010/08/best-practices-for-handling-android.html`, 2010.

[28] Livshits, V. B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[29] Lookout. Update and Clarification of Analysis of Mobile Applications at Blackhat 2010. `http://blog.mylookout.com/2010/07/mobile-application-analysis-blackhat/`, July 2010.

[30] McDaniel, P., and Enck, W. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine 8*, 5 (September/October 2010), 76–78.

[31] Miecznikowski, J., and Hendren, L. Decompiling java using staged encapsulation. In *Proceedings of the Eighth Working Conference on Reverse Engineering* (2001).

[32] Miecznikowski, J., and Hendren, L. J. Decompiling java bytecode: Problems, traps and pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction* (2002).

[33] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (August 1978).

[34] Oberheide, J. Android Hax. In *Proceedings of SummerCon* (June 2010).

[35] Octeau, D., Enck, W., and McDaniel, P. The ded Decompiler. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sept. 2010.

[36] Ongtang, M., Butler, K., and McDaniel, P. Porscha: Policy Oriented Secure Content Handling in Android. In *Proc. of the Annual Computer Security Applications Conference* (2010).

[37] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the Annual Computer Security Applications Conference* (2009).

[38] Porras, P., Saidi, H., and Yegneswaran, V. An Analysis of the Ikee.B (Duh) iPhone Botnet. Tech. rep., SRI International, Dec. 2009. `http://mtc.sri.com/iPhone/`.

[39] Proebsting, T. A., and Watterson, S. A. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems* (1997).

[40] Raphel, J. Google: Android wallpaper apps were not security threats. *Computerworld* (August 2010).

[41] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., and Wang, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[42] Schwarz, B., Chen, H., Wagner, D., Morrison, G., West, J., Lin, J., and Tu, W. Model Checking an Entire Linux Distribution for Security Violations. In *Proceedings of the Annual Computer Security Applications Conference* (2005).

[43] Spencer E. Ante. Banks Rush to Fix Security Flaws in Wireless Apps. `http://online.wsj.com/article/SB10001424052748703805704575594581203248658.html`, November 2010.

[44] Storm, D. Zombies and Angry Birds attack: mobile phone malware. *Computerworld* (November 2010).

[45] Tiuryn, J. Type inference problems: A survey. In *Proceedings of the Mathematical Foundations of Computer Science* (1990).

[46] Vallee-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing java bytecode using the soot framework: Is it feasible? In *International Conference on Compiler Construction, LNCS 1781* (2000), pp. 18–34.

[47] Ware, M. S., and Fox, C. J. Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools. In *Proceedings of the Workshop on Static Analysis (SAW)* (2008).

[48] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the ACM conference on Computer and Communications Security* (2007).