Path-Sensitive Analysis of Message-Controlled Communication for Android Apps

Joshua Garcia
University of California, Irvine
joshua.garcia@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

September 2016
ISR Technical Report # UCI-ISR-16-4

Path-Sensitive Analysis of Message-Controlled Communication for Android Apps

Joshua Garcia and Sam Malek
Institute for Software Research, University of California, Irvine
Department of Informatics, University of California, Irvine
Irvine, California, USA
{joshug4, malek}@uci.edu

***Abstract*—To support quality development of Android apps, a variety of techniques have been produced for analyzing the exchange of messages, i.e., Intents, among Android components. Intents and their payloads can cause a variety of operations to be performed, and can be filled with malicious data, demonstrating that Intents can serve as attack vectors of an insecure app. Intents may further guard or control execution of different program paths, which may contain vulnerable, faulty, or energy-inefficient code. While different techniques have focused on determining possible Intents in an app, *none have focused on analyzing Intents per program path*, i.e., path-sensitive Intent analysis. Analyzing a program per path allows the *determination of the attributes of Intents needed to control execution of a program* from its message-based inter-component interface. Unfortunately, *analyzing apps in a path-sensitive manner faces scalability issues*. To address these challenges, we introduce a novel, scalable framework called *PHENOMENON* (PatH-sEnsitive aNalysis Of MEssage-coNtrOlled communication for aNdroid apps). We evaluate the accuracy of *PHENOMENON*'s path-sensitive analysis on five apps with 4KSLOC–460KSLOC, over a total of 4,100 program paths, achieving an accuracy of over 96% for each app. To evaluate *PHENOMENON*'s efficiency, we assess it on 100 randomly selected apps, demonstrating an average runtime of 30 seconds, with no app taking more than 180 seconds to analyze.**[1]

***Keywords*-Android; Intent; path-sensitive; event-based; message-control;**

I. Introduction

Mobile devices are ubiquitous, with hundred of millions of smartphones and tablets used worldwide. Among these popular mobile devices, Android has emerged as the dominant platform. Fueling the popularity of such devices are the abundance of applications (apps) available on a variety of app markets (e.g., Google Play). This abundance of apps arises, in large part, due to the Android platform's low barrier to entry for amateur and professional developers alike, where a re-usable infrastructure enables relatively quick production of apps. However, this low barrier to entry is associated with an increased risk of apps with errors, security vulnerabilities, poor energy efficiency, etc. Consequently, developers and designers of such apps need to utilize appropriate approaches, tools, and frameworks that aid them in satisfying an app's functional and non-functional requirements.

To support quality development of Android apps, a variety of techniques have been produced for analyzing inter-component communication (ICC) of Android apps [1], [2], [3], [4]. The main form of ICC in Android apps are *Intents*, which are asynchronous messages that enable transfer of data and control between Android components and apps [5]. Intents can serve as interfaces to background services of apps (e.g., app-specific services or repeating alarms), communication between GUI components, performing broadcasts between apps, and other forms of ICC. Intents and their payloads can cause a variety of operations to be performed, and can be filled with malicious data, demonstrating that Intents can serve as attack vectors of an insecure app. Intents may further guard or control execution of different program paths, which may contain vulnerable, faulty, or energy-inefficient code.

While different techniques have focused on determining possible Intents in an app, *none have focused on analyzing Intents per program path*, i.e., path-sensitive Intent analysis. Analyzing Intents per program path allows engineers to analyze different use-case scenarios involving different parts of an app. Furthermore, analyzing programs per path enables determination of the effects of Intent payloads on the execution of particular program paths. For example, a specific combination of attributes of an Intent may cause a particular vulnerability to be exploited along a program path. As a result, analyzing a program per path allows the *determination of the attributes of Intents needed to control execution of a program* from its ICC interface.

Unfortunately, *analyzing apps in a path-sensitive manner faces scalability issues*. Specifically, the number of paths in a program grows exponentially. Due to this challenge, existing analysis techniques have focused on utilizing more scalable analyses (i.e., data-flow analyses). Although such analyses provide useful information as to the possible Intents in an app, these analyses significantly over-approximate possible Intent-driven behaviors in an app and cannot determine the specific payload of an Intent needed to execute a particular program path.

To provide scalable path-sensitive analyses of Intents and the manner in which they control program execution, we introduce a novel framework called *PHENOMENON* (PatH-sEnsitive aNalysis Of MEssage-coNtrOlled communication for aNdroid apps). Our framework performs its analyses in a scalable and path-sensitive manner. To achieve scalability, *PHENOMENON* performs a pre-analysis to first identify *tar-*

[1]ISR Technical Report UCI-ISR-16-4, September 2016

get program statements at which an Intent or its payload may be used, and the attributes that may control execution of a particular path in a program. Using that information, *PHENOMENON* performs a backward symbolic execution per identified target statement, pruning the potential analysis space. This symbolic execution is conducted in a parallel manner to improve scalability. To achieve this parallelism, our algorithm carefully handles locking critical sections of our algorithm where data may be shared. We further carefully select the number of paths analyzed per target statement, to avoid cases where the underlying analysis may not terminate, due to path explosion. Our experiments demonstrate that it is possible to set an upper bound on the number of paths analyzed, while still maintaining completeness or accuracy of our analysis.

The main contributions of *PHENOMENON* are as follows:

- We present a scalable, accurate framework, *PHENOMENON*, for the determination of message-controlled communication per program path in an Android app.
- We construct an implementation of *PHENOMENON* and make it, and other evaluation artifacts, publicly available for inspection, replication, and support of future research [6].
- We evaluate the accuracy of *PHENOMENON*'s path-sensitive analysis on five apps with 4KSLOC–460KSLOC, over a total of 4,100 program paths, achieving an accuracy of over 96% for each app. More specifically, we assessed the extent to which *PHENOMENON* is capable of accurately identifying the Intents along particular program paths and the accuracy of expressions generated by *PHENOMENON* describing message control along those paths.
- To assess the efficiency of *PHENOMENON*, we ran *PHENOMENON*'s implementation on 100 randomly selected apps, demonstrating an average runtime of 30 seconds, with no app taking more than 180 seconds to analyze.

II. Android Background and Running Example

The Android Development Framework (*ADF*) supplies developers with a set of customizable components and communication mechanisms that allow construction of mobile apps. In particular, Android includes four pre-defined components: Activities, Services, Broadcast Receivers, and Content Providers. An *Activity* represents a GUI screen that an app displays to a user and allows her to interact with the app. A *Service* runs operations in the background for an app. *Content Providers* represent persistent data storage for an app. *Broadcast Receivers* receive Intents that are, as its name implies, broadcasted by other apps or the Android framework itself (e.g., indicating that the battery is low or that the device has finished booting). Of these four different data types, Activities, Services, and Broadcast Receivers can exchange Intents.

As an event-based system [7], [8], certain Android components may have multiple entry points corresponding to the lifecycle of a component's type. For example, an Activity has separate entry points for initial creation, being sent to the background in order to pause the Activity, and resuming the Activity after pausing. As another example, Services have separate entry points depending on whether another component connects to it for a long period of time (referred to as starting the service) as opposed to temporarily binding to it. Although Broadcast Receivers have a single entry point, they may be registered dynamically, and ideally an analysis of Intents should take that dynamic registration into account to improve the accuracy of analysis.

From our studies of Android apps, there are mainly three types of attributes of an Intent that an app uses to determine the manner in which it will utilize the Intent and perform operations based on it: the *action* of an Intent, its *categories*, and its *extra data*. The action of an Intent is an attribute that indicates the general operation to be performed in response to an intent (e.g., display data to the user or deliver data to some person or agent). Categories of an Intent provide additional information as to the manner in which the Intent's action should be performed (e.g., whether the Intent will allow launching of an application as referenced by a link in a browser). Extra data, also called *Bundles*, are a collection of key-value pairs in an Intent, allowing even more flexible attributes to be stored in it. Intents require an action but categories and extra data are optional.

To better understand the challenges faced by *PHENOMENON*, we present an example of an Android encryption Service based on a popular Android encryption/decryption app, **ApgIntentService**, depicted in Figure 1. This Service performs different kinds of encryption-related operations based on the payload of an Intent. For example, to perform encryption using this Service, the **action** of the Intent (extracted on line 3) must be "ENCRYPT". The **target** extra datum of the Intent (extracted on line 11) determines whether the encryption is stored as a byte array or a URI stream. The **useAsciiArmor** boolean extra of the Intent (extracted on line 12) determines if the resulting data is encrypted as a string or byte array. Furthermore, a passphrase may be cached on line 8, if the "PASSPHRASE_CACHE" action is received by **ApgIntentService** with an extra datum containing the key "ttl" > 0 and the Intent does not have the default category, indicating the Intent was sent explicitly to the **ApgIntentService**. Each of these different values, all obtained from an Intent, control the execution paths of the Service, and whether or not particular target statements execute.

III. Related Work

Researchers have produced a variety of analyses for Android and Intent-based communication in Android. EPICC [1] utilizes a data-flow analysis based on IFDS/IDE to identify Intents and certain aspects of their payloads. IC3 expands upon EPICC with a greater focus on URIs, which are particularly used with Content Providers, and composite constant propagation. However, neither IC3 [2] nor EPICC identify possible Intents in each program path or the potential values, as opposed to just keys, of Intent extra data. Both of these static analyses for Intents do not describe the specific values

```java
public class ApgIntentService extends IntentService {
protected void onHandleIntent(Intent intent) {
  String action = intent.getAction();
  if ("ENCRYPT".equals(action))
      doEncrypt(intent);
  else if ("PASSPHRASE_CACHE".equals(action)) {
    if (intent.getIntExtra("ttl",0) > 0 && !intent.
          hasCategory("android.intent.category.DEFAULT"))
         cachePassphrase(intent); }}
private void doEncrypt(Intent intent) {
  Bundle data = intent.getExtras();
  int target = data.getInt("target");
  boolean useAsciiArmor = data.getBoolean("
       ENCRYPT_USE_ASCII_ARMOR");
  switch (target)
  {case 0:
     if (useAsciiArmor)
       // encrypt as a string
     else
       // encrypt as byte array
     break;
   case 1:
     // encrypt as URI stream
     break;}}}
```

Fig. 1. An encryption Service example based on a popular encryption Android app.

of Intent payloads needed to execute particular program paths in an app.

A series of techniques have been constructed that focus on data or permission leakage for Android apps. IccTA [9], AmanDroid [10], and SEPAR [11] focus on detecting inter-component data leakage for Android apps. COVERT [12] identifies permission leakages among groups of Android apps. FlowDroid [13] focuses on data leakage within components. None of these approaches focus on constructing Intents, their payloads, and the manner in which Intents execute different program paths. TASMAN [14] uses symbolic execution to filter out infeasible paths as a post analysis for a data-leakage analysis.

Certain techniques utilize symbolic execution for various purposes in Android. AppIntent [15] and SIG-Droid [16] generate tests for GUI screens by using symbolic execution that analyzes GUI events. IntelliDroid [17] focuses on executing Android APIs in order to find data leakages in malicious apps. IntentDroid [18] utilizes dynamic analyses to generate Intents in order to identify security issues in Android apps. IntentDroid is particularly limited to only analyzing extra data of boolean types, and thus path conditions analyzed only involve boolean variables. None of these techniques attempt to construct a path-sensitive approach for determining Intent payloads, especially both keys and values of extra data for a variety of data types, in order to determine the payload needed across Intent-controlled paths.

Various static analyses have been designed specifically for Android in order to enhance or enable other analyses. Two techniques [19], [20] have been developed to improve string analysis for Android apps. Yang et al. [21] produced a technique to determine a control-flow graph focusing on user-driven callbacks for Android. EdgeMiner [22] identifies pairs of registrations and callbacks by analyzing the Android framework itself. CLAPP [23] extracts information about loops in Android apps, particularly the extent to which a loop is bounded.

Analysis of event-based systems, including the messages they exchange, has been studied outside of the domain of Android as well. Jayaram and Eugster [24] present static and dynamic analyses for event-based systems, including determining dependencies between messages. Helios [25] extracts dependencies between messages where messages are represented as nominal types, i.e., a message type is represented by the event-based frameworks underlying programming language. Eos [26] improves upon Helios by extracting messages that include attributes in the form of key-value pairs, instead of simply its nominal type. Unlike *PHENOMENON*, none of these techniques extract messages needed to execute a particular program path. Furthermore, while Eos can extract values of attribute that are strings, *PHENOMENON* can extract values of extra data of primitive types (e.g., booleans, floats, integers, etc. and constraints among them) and the nullness of objects as values of an extra datum. *PHENOMENON* also models collections of attributes, particularly in the case of Intent categories.

Table I depicts key features of *PHENOMENON* and the techniques most similar to it along five dimensions: the technique's path-sensitivity, if applicable; if the technique determines the payload necessary to execute a particular program path controlled by the contents of a message; whether the technique leverages static analysis, dynamic analysis, or both; the degree to which the analysis can extract attributes of a message, i.e., the contents of the message; and the targeted event-based framework of the analysis. *PHENOMENON* is currently the only technique designed to determine a rich variety of possible Intent contents necessary to execute any reachable program path of an Android app. *PHENOMENON* achieves this by utilizing a path-sensitive analysis that models Intent actions and categories, and also strings, primitive types, and object nullness of Intent extra data. Furthermore, one of the key issues reducing accuracy for Eos was its lack of path sensitivity [26].

IV. Approach

PHENOMENON is a flow-sensitive, object-sensitive, context-sensitive, and path-sensitive analysis that operates primarily through backwards symbolic execution [27] and a backwards data-flow analysis over the app's use-def chains [22]. To obtain a call graph suitable for analysis of Android apps, the call graph must take into account the multiple entry points of an Android app and its lifecycle. To achieve this, *PHENOMENON* incorporates incremental callback analysis to construct a call graph as described in previous work [13], where the call graph is continuously updated with identified callback registrations until a fixed point is reached. *PHENOMENON* further includes entry points for call graph construction to dynamically registered Broadcast Receivers. Specifically, *PHENOMENON* scans the call graph built using incremental callback analysis. If within that callback, a programmatic registration of a Broadcast Receiver is found, our

TABLE I
FEATURE COMPARISON OF *PHENOMENON* WITH MOST RELATED MESSAGE-BASED ANALYSES

	PHENOMENON	**EPICC** [1]	**IC3** [2]	**EventJava** [24]	**Helios** [25]	**Eos** [26]	**IntelliDroid** [17]	**IntentDroid** [18]
Path-Sensitive	Yes	No	No	No	No	No	Yes	N/A
Message-Control	Yes	No	No	No	No	No	No	Partial
Static/Dynamic	Static	Static	Static	Static/Dynamic	Static	Static	Static	Dynamic
Attribute-Value Extraction	Intent actions, categories, extra data of primitive types, extra data of object nullness	Intent actions and categories	Intent actions and categories	Unsupported	Unsupported	Non-string attribute values	Intent actions and categories	Intent actions, categories, and boolean extra data
Framework Target	Android	Android	Android	EventJava Framework	Any	Any	Android	Android

Algorithm 1: *intentControlAnalysis*

```
Input: set of methods $M$ in reverse topological order from the app
Output: a map $\Sigma_\omega : M \rightarrow targetExprs$, which describe the Intents and path
        conditions of methods $M$
1  $targetStmts \leftarrow identifyTargetStmts(M)$;
2  $\Sigma_\alpha \leftarrow \Sigma_\beta \leftarrow \Sigma_\omega \leftarrow \emptyset$;
3  foreach method $m \in M$ do
4      $useDefs_m \leftarrow constructUseDefChains(m)$;
5      foreach statement $s_t \in m.statements$ do
6          if $s_t \in targetStmts$ then
7              $reachPaths \leftarrow constructBackReachPaths(m.cfg, s_t)$;
8              $intraPathExprs \leftarrow \emptyset$;
9              foreach path $p \in reachPaths$ do
10                 foreach $s_p \in reachPaths$ do
11                     $intraPathExprs \leftarrow$
                         $generateExprsForStmt(s_p, p, useDefs_m) \cup$
                         $intraPathExprs$;
12                 $\Sigma_\alpha \leftarrow \Delta_\alpha(\Sigma_\alpha, s_t, intraPathExprs, p)$;
13             foreach path $p \in reachPaths$ do
14                 foreach statement $s_p \in p$ do
15                     if $s_p$ is an invocation of the form $r_b.m_\alpha(A)$ and
                         $\Sigma_\alpha(m_\alpha) \neq \emptyset$, and $A = (a_n)_{n \in \mathbb{N}}$ then
16                         if argument $a$ is an Intent and $a \in A$ then
17                             if $\Sigma_\alpha(m_\alpha)$ has an Intent referencing the
                                 parameter matching argument $a$ then
18                                 $\Sigma_\beta \leftarrow \Delta_\beta(\Sigma_\beta, s_t, newExprs, p)$;
19             foreach path $p \in reachPaths$ do
20                 foreach statement $s_p \in p$ do
21                     $\Sigma_\omega \leftarrow$
                         $buildCtxtSensSummaries(\Sigma_\alpha, \Sigma_\beta, p, s_p, m)$;
```

analysis rebuilds the call graph by including the Broadcast Receiver's entry point. In addition to the callbacks of the four canonical Android component types described in Section II, we further include entry points for our analysis that utilize the Android **Loader** class and its subclasses (e.g., **AsyncTaskLoader**). **Loader**s provide means to perform asynchronous tasks on a separate thread in an event-driven manner and often have an observer (e.g., a dynamically registered Broadcast Receiver) that monitors data source changes of the **Loader**.

The main algorithm driving *PHENOMENON*'s analysis, *intentControlAnalysis*, is depicted in Algorithm 1. Similar to previous analyses [28], [26], *intentControlAnalysis* is a summary-based analysis that processes methods in the app's call graph in reverse topological order. By analyzing methods in that order, *intentControlAnalysis* ensures that a callee method's summary is constructed and available before a caller method is analyzed, preventing the need to analyze a method more than once and improving *intentControlAnalysis*'s efficiency. *intentControlAnalysis* returns a map $\Sigma_\alpha : M \rightarrow targetExprs$ summarizing the analysis results for each method $m \in M$. Each $e \in targetExprs$ is a pair $(s_\tau, exprs_p)$. $exprs_p$ is a sequence of expressions describing Intents and path conditions in a program path p; and s_τ is a *target statement*, where backward symbolic execution initiates from, which is futher elaborated in the next section.

intentControlAnalysis constructs intermediate analysis results in two phases, and combines those results in a third and final phase. The first phase occurs on (lines 9-11 of Algorithm 1) and stores Intent data in Σ_α for intra-method paths. In the next phase (lines 13-18 of Algorithm 1), method summaries are utilized to determine context-sensitive Intent information at call sites by enumerating path expressions from Σ_α inter-procedurally, and storing that information in Σ_β. The final phase (lines 19-21 of Algorithm 1) involves combining the results of Σ_α, intra-method results, and Σ_β, inter-procedural results, to obtain the final results Σ_ω, context-sensitive results for the entire app. At this point, the computed path conditions and expressions describing Intents may be sent to a solver to check for feasibility, generate data along the path, or other purposes.

A. Bootstrapping the Analysis

Before determining the attributes (i.e., actions, categories, and extra data) within an Intent along a particular program path and how these attributes may control execution of that path, Algorithm 1 begins by identifying program points that serve as target statements for *PHENOMENON*'s backwards symbolic execution by invoking $identifyTargetStmts$ on line 1. These target statements are any points in the app where an Intent's attributes are used. For example, the **getAction** invocation that extracts an Intent's action (line 3 in Figure 1) or the **hasCategory** invocation that checks for existence of a category in the Intent (line 7 in Figure 1). To further ensure that the blocks controllable by these Intents are also analyzed, we add successor blocks of these initial target statements as additional target statements to perform backwards symbolic execution from. For instance, once line 7 of Figure 1 is marked as a target statement, the line containing the invocation to **cachePassphrase()** is also marked as an additional starting

Algorithm 2: *constructBackReachPaths*

```
Input: A control-flow graph $cfg$ of a method, A target statement $t$ that is a node of $cfg$
Output: A set of program paths $reachPaths$ that reach $t$
1  $workStmts \leftarrow t$;
2  $workPaths \leftarrow ((t))$;
3  $reachPaths \leftarrow \{\}$;
4  while $workStmts \neq \emptyset$ do
5      $currStmt \leftarrow workStmts.head$;
6      $currPath \leftarrow workPaths.head$;
7      if $cfg.predecessorsOf(currStmt) = \emptyset$ then
8          $reachPaths \leftarrow reachPaths \cup currPath$
9  foreach $predStmt \in cfg.predecessorsOf(currStmt)$ do
10     if $predStmt \notin currPath$ then
11         $newPath \leftarrow currPath$;
12         $newPath \leftarrow currPath \frown predStmt$;
13         $workPaths \leftarrow newPath \frown workPaths$;
14         $workStmts \leftarrow predStmt \frown workStmts$;
```

Algorithm 3: *generateExprsForStmt*

```
Input: A statement $s_p$ of method $m$, path $p$ in $m$ containing $s_p$, use-def chain $useDefs_m$ of method $m$
Output: expressions $newExprs$ describing Intent and path-condition information at statement $s_p$
1  $newExprs \leftarrow \emptyset$;
2  if $s_p$ extracts extra data from an Intent $i$ of the form $r_e$ = $i$.get[$\Psi$]Extra($r_k$) then
3      $newExprs \leftarrow genExtraDataExprs(s_p, p, useDefs_m) \cup newExprs$;
4  else if $s_p$ extracts an action from an Intent $i$ of the form $r_a$ = $i$.getAction() then
5      $newExprs \leftarrow genGetActionExprs(s_p, p, useDefs_m) \cup newExprs$;
6  else if $s_p$ is of the form if ($i$.hasCategory($r_c$)), where $i$ is an Intent then
7      $newExprs \leftarrow genCategoryExprs(s_p, p, useDefs_m) \cup newExprs$;
8  else if $s_p$ is a conditional statement of the form if (l op r) then
9      $newExprs \leftarrow genConditionalExprs(s_p, p) \cup newExprs$;
10 else if $s_p$ is a conditional statement of the form if ($r_1$.equals($r_2$)) then
11     if $r_1$ is a String obtained from an Intent's extra data then
12         $newExprs \leftarrow genStringAttrExprs(s_p, p, useDefs_m) \cup newExprs$;
13     else if $r_1$ is an arbitrary object obtained from an Intent's extra data then
14         $newExprs \leftarrow genObjEqualityExprs(s_p, p, useDefs_m) \cup newExprs$;
```

point. By first selecting relevant program points to start our analysis from, *PHENOMENON* prunes the space of the analysis, reducing issues of path explosion often faced by symbolic execution.

Once the points from which to start backward symbolic execution are identified, Algorithm 1 analyzes each method m by first constructing m's use-def chains (line 4 of Algorithm 1). For each target statement s_t of a method m, line 7 of Algorithm 1 constructs all the *relevant* program paths from the entry point of the program to the target statements by invoking *constructBackReachPaths*, shown in Algorithm 2. To avoid analyzing paths that may not actually reach these target statements, and thus improving analysis efficiency. Algorithm 2 builds these relevant program paths through a backward traversal algorithm over a method's control-flow graph. Algorithm 2 takes as input a control-flow graph *cfg* and a target statement t belonging to *cfg*, and outputs the set of program paths that reach t starting from the method's entry points.

B. Generating Expressions Describing Intents

The production of expressions describing message-controlling Intents occurs during the first phase, on lines 9-11 of Algorithm 1. For each statement s_p in a path p that reaches target statement s_t, Algorithm 1 at line 11 generates a set of expressions describing the Intent or the path conditions at s_p by invoking *generateExprsForStmt*, shown in Algorithm 3. *generateExprsForStmt* generates each expression in a language suitable for supplying to an SMT solver, i.e., the SMT-LIB language [29], allowing our analysis to use the SMT solver to determine feasibility of paths, and also the validity of the expressions describing Intents, their attributes (i.e., actions, categories, and extra data), and their relations to programming language-level constructs (e.g., object references, definition sites, etc.).

generateExprsForStmt takes as input a statement s_p of method m, path p in m containing s_p, and use-def chain $useDefs_m$ of method m. As output, *generateExprsForStmt* constructs path-sensitive expressions describing a message-controlling Intent and path conditions for the Intent. By considering the path p of s_p, *generateExprsForStmt* ensures that expressions generated for s_p are relevant to p, thus maintaining path sensitivity. Each conditional block in Algorithm 3 handles a different type of program statement and generates expressions based on that statement type: extra data on lines 2-3, the action of an Intent on lines 4-5, categories of the Intent on lines 6-7, operators between numeric and boolean symbols on lines 8-9, and string and object equivalence comparison on lines 13-14.

Extra Data. To handle extra data, *genExtraDataExprs* (at line 3 of Algorithm 3) produces symbolic variables for the following references: r_k, key of the extra datum extracted from the Intent i; r_e, containing the value of the extra datum; and i for the reference of the Intent housing the extra datum. *genExtraDataExprs* further records the type of the extra datum at the programming language-level when declaring a new symbol by taking the API method's type Ψ into account. For example, in the case of the API method **getIntExtra**, Ψ is an integer. To represent the new generated information, *genExtraDataExprs* creates expressions of the following form, with declarations removed for brevity:

e1) (**assert** (= (**containsKey** r_e r_k) **true**))
e2) (**assert** (= (**fromIntent** r_e) i))

The first expression indicates that extra datum r_e contains key r_k. The second expression asserts that r_e is from intent i. We further define a generic **Object** datatype for the solver that can be either null or not null. In the expressions above, i is declared as an **Object**, r_k is a built-in String type, and r_e varies in type depending on Ψ. As an example, consider line 7 of Figure 1. On that line, the **getIntExtra** invocation results in the generation of the following expressions:

e3) (**assert** (= (**containsKey** r_{ttl} "*ttl*") **true**))
e4) (**assert** (= (**fromIntent** r_{ttl}) r_{intent}))

PHENOMENON and its underlying algorithms ensure that, whenever a symbol or expression is generated, the following criteria are met: (1) any definitions of references are along the current path under analysis and (2) the closest definition for the reference at the statement under analysis is used. These two criteria ensure that data along other paths is not generated and that values of dead variables are not used, thus maintaining path sensitivity. Additionally, since we compute our analysis in a backwards fashion along a path, we create a different symbol every time a variable is redefined, as done in previous work [14], in order to simulate static single assignment.

PHENOMENON also tracks extra data not directly extracted from Intents. For example, consider the extra datum "target" from an Intent extracted at line 11 of Figure 1. This datum is obtained from a **Bundle** object, which represents the extra data within an Intent, but also has a different API than if extra data is extracted directly from an Intent. *PHENOMENON* tracks this information as well as part of extracting extra data from Intents.

Actions. For actions of an Intent, $genGetActionExprs$ (at line 5 of Algorithm 3) produces symbolic variables for the following references: r_a, the reference storing the action of Intent i; and the reference to i. Using those variables, the function creates the following expressions:

a1) (**assert** (= (**getAction** i) r_a))
a2) (**assert** (= (**fromIntent** r_a) i))

The first expression indicates that i has the action r_a. By defining the function **getAction** in the solver, it can verify that along the path, Intent i should only have a single action. As in the case for extra data, i is declared as an **Object**.

Before conditional statements are processed by *generateExprsForStmt*, expressions generated by $genGetActionExprs$ only indicate the existence of an action r_a for an Intent i along a path.

Categories. $genCategoryExprs$ (at line 7 of Algorithm 3) handles categories by analyzing a conditional statement that checks if an Intent has a category. In particular, the function creates the following symbols: r_h representing the boolean reference indicating if the Intent i has a category r_c; the Intent i; and r_c, which is a string reference representing the name of the category. To determine, if along the path under analysis, i has category r_c, $genCategoryExprs$ must determine if the path containing the conditional check on **hasCategory** and its successor statement is along a true branch or false branch. A true branch indicates that r_c is in i; a false branch implies the opposite.

Although a set would be an ideal representation of categories in an SMT solver, it is not always the case that sets are built-in to the solver. Furthermore, specifying them properly is a research challenge in its own right [30]. Consequently, we simply represent a set as an array and use expressions involving quantifiers to check existence or absence of a category. Therefore, for existence of a category, $genCategoryExprs$ generates the following expression:

$$(\textbf{assert}\ (\textbf{exists}((idx\ \textbf{Int}))(=(\textbf{select}\ cats_{r_h}\ idx)\ r_c)))$$

The above expression simply asserts existence of an element at index idx in the array $cats_{r_h}$ that contains the value r_c, using the existential quantifier. For absence, $genCategoryExprs$ generates the following expression:

$$(\textbf{assert}\ (\textbf{forall}((idx\ Int))(\textbf{not}\ (=(\textbf{select}\ cats_{r_h}\ idx)\ r_c))))$$

The expression asserts that for all elements in the array $cats_{r_h}$ there is no element with the value r_c. To relate the categories $cats_{r_h}$ to an Intent i, $genCategoryExprs$ produces an expression using the **fromIntent** function, similar to the cases involving extra data and actions.

As an example, consider a partial path from lines 7-8 in Figure 1. For the statement at line 7, $genCategoryExprs$ generates the following expression, where we elide the **fromIntent** expression due to space limitations:

$$(\textbf{assert}\ (\textbf{forall}\ ((idx\ Int))\ (\textbf{not}\ (=(\textbf{select}\ cats_{r_h}\ idx)\ \text{“}android.intent.category.DEFAULT\text{”}))))$$

Primitive Comparisons. To properly characterize message control along program paths, it is of key importance that we model as many comparison statements as possible involving primitive types, especially those extracted from Intents. For example, in Figure 1, comparisons involving the boolean variable **useAsciiArmor**, and the integers **target** and **ttl** control the execution of different parts of **ApgIntentService**. To that end, $genConditionalExprs$ (invoked on line 9 of Algorithm 3) transforms conditional statements involving a variety of comparisons among primitive data types. To that end, given a conditional statement of the form **if** (l op r), *PHENOMENON* models the following comparison operators ==, !=, <=, >=, <,>. The left and right references (l and r) can be any numeric or boolean types.

For the l and r references of the conditional comparison statements, we track whether the reference refers to either (1) an extraction of extra data or (2) a check of its existence. In the former case, the generation of expressions closely resembles how extra data is extracted on lines 2-3 of Algorithm 3. In the second case, where a check of the existence of an extra datum of an Intent, the form of the statement is **if** (i.**hasExtra**(r_k)). In this case, we further generate the appropriate **containsKey** expression, as in the extra data extraction case; however, we further consider whether the path the reference obtained from is along a true or false branch. Based on that branch, we determine whether or not the **containsKey** expression is asserted to be **true** or **false**.

For example, consider lines 7-8, and the statement `if (intent.getIntExtra("ttl",0)> 0`. Besides the expressions e3 and e4 generated by $genExtraDataExprs$, $genConditionalExprs$ also generates an expression (**assert** ($>$ r_{ttl} 0)). Thus, conditions necessary to execute a program path, especially those obtained from Intents are characterized by $genConditionalExprs$.

Besides managing any primitive comparisons, including ones involving Intent extra data, $genConditionalExprs$ further generates expressions for checks that determine if an object is null. To handle this special case, $genConditionalExprs$ determines if r is the **null** constant. For these comparisons, $genConditionalExprs$ produces an expression of the following form to assert for nullness: $(\textbf{assert}(= \ (\textbf{isNull}\, l)\, \beta))$. β is a boolean value, i.e., **true** or **false**. As in other cases, by checking if the null check statement is along a true or false branch, for the path under analysis, and sets β to the appropriate boolean value accordingly.

String and Object Comparisons. Equality comparisons among strings, and to a lesser extent objects in general, are critical for determining the contents of Intents that control execution of different program paths. Although other forms of string manipulation may potentially affect execution, they are extremely rare, as found both in this study and previous work [14], [12], [26], [25]. Consequently, our analysis focuses on representing and handling string equality.

Specifically for Intents, determining string extra data and values of actions for an Intent are dependent on extracting equality comparisons. As an example, for any path that reaches line 8 of Figure 1, the **equals** comparison of strings at line 6 in that figure must evaluate to true. Furthermore, along that path, the action of the Intent must be “PASSPHRASE_CACHE”.

To extract Intent information from string comparisons, $genStringAttrExprs$ (invoked on line 12 of Algorithm 3) analyzes string equality statements. For statements of the form shown on line 13, $genStringAttrExprs$ creates symbols for references $r1$ and $r2$ declared as built-in strings and generates an expression of the form $asrt_{eq} = (\textbf{assert}\ (= r_1\, r_2))$ if the comparison is true along the path under analysis, and generates the expression $\neg asrt_{eq} = (\textbf{assert}\ (\textbf{not}\ (= r_1\, r_2)))$ otherwise. As in the case of non-conditional extra data extraction, expressions of the form e1 and e2 are generated as well, describing the key of the string extra datum and the Intent it belongs to.

To obtain potential values for actions of an Intent along a path, $genStringAttrExprs$ need only generate the assertion expressions of the form $asrt_{eq}$ or $\neg asrt_{eq}$. These expressions combined with the expressions of the form a1 and a2 extracted by $genGetActionExprs$ describe potential string values for actions of a particular intent.

For the example of line 6 in Figure 1 with a path ending at line 8 of Figure 1, the following expressions are generated by $genStringAttrExprs$:

a3) (**assert** $(= r_a$ “PASSPHRASE_CACHE”))

a4) (**assert** (**not** $(= r_a$ “ENCRYPT”)))

The other relevant information along the path for the action, generated by $genGetActionExprs$, are as follows:

a5) (**assert** (= (**getAction** i_2) r_a))

a6) (**assert** (= (**fromIntent** r_a) i_2))

In expression a5 and a6, the intent symbol’s subscript represents the line number where the Intent is created. In that case, the Intent is created at the entry point of **onHandleIntent** at line 2. Therefore, *PHENOMENON* tracks the attributes of an Intent along a path that must exist in it and that cannot exist in it.

When comparing arbitrary objects, $genObjEqualityExprs$ (invoked on line 14 of Algorithm 3) operates in a manner highly similar to that of $genStringAttrExprs$. $genObjEqualityExprs$ still creates expressions of the form $asrt_{eq}$ or $\neg asrt_{eq}$. These objects are declared as our custom **Object** type and may also be assigned to an Intent using the **fromIntent** function as part of a generated expression. A special case occurs when an string r_{so} is checked against the **null** constant along a path and is then checked to contain a specific string. An example of such case is as follows:

```
if (action != null)
  if ("ENCRYPT".equals(action))
```

To avoid type conflicts in such a case, we generate a symbol for a reference r_{so} as an **Object** and another symbol for r_{so} as a string. We then create a custom function **objEquals** that allows comparison of strings with objects for the SMT solver.

C. Constructing Context-Sensitive Results

After the first phase completes, computed results stored in Σ_α are intra-procedural. The second phase (lines 13-18 of Algorithm 1) enumerates paths inter-procedurally by identifying call sites of summarized methods stored in Σ_α. Specifically, *intentControlAnalysis* checks three criteria to determine where to enumerate paths for a statement s_p in path p under analysis: (1) s_p is a call site to a summarized method m_α in Σ_α, (2) an argument a passed to m_α is an Intent, and (3) $\Sigma_\alpha(m_\alpha)$ contains expressions indicating that information is generated from a parameter of m_α that matches a. For example, line 5 of Figure 1 is a call site where the method **doEncrypt** is invoked and is also summarized in Σ_α after the first phase. Consequently, the second phase of *intentControlAnalysis* will enumerate paths and paths expressions inter-procedurally and store that information in Σ_β.

To clarify, consider the following intra-method path $p_{ohi} = (2, 3, 4, 5)$ of **onHandleIntent** in Figure 1; and the two intra-method paths of **doEncrypt** from that figure $p_{de_1} = (9, 10, 11, 12, 13, 14, 15, 16)$ and $p_{de_1} = (9, 10, 11, 12, 13, 14, 20, 21)$. In these three paths, the numbers in the path represent the line numbers in the figure. In the second phase, for path p_{ohi}, the paths and path expressions for p_{de_1} and p_{de_2} are stored in Σ_β, while Σ_α has the paths and path expressions for p_{ohi}.

Once the third phase executes, these three paths will be combined to two paths with all the relevant path expressions stored in Σ_ω. Specifically, $p_{ohi} \frown p_{de_1}$ forms a final context-sensitive path and $p_{ohi} \frown p_{de_2}$ forms a second final context-sensitive path summarized by Σ_ω.

D. Improving Analysis Efficiency

To deal with scalability and efficiency issues during path explorations, besides pruning paths through target statements and focusing on Intent data, *PHENOMENON* can utilize either parallelism or setting an upper bound on the number of paths analyzed. In the former case, *PHENOMENON* can distribute the workload for each target statement t_s to a different worker thread. Synchronization must occur during construction of use-def chains (line 4 of Algorithm 1). Our experience has not found a need to synchronize other code regions. Given that our analysis makes significant use of map data strutures, we have found the use of concurrent map data structures is effective for enabling parallelism, while avoiding issues of concurrent modification. Specifically, they are most effective for use when implementing Σ_α, Σ_β, and Σ_ω.

In the second case, *PHENOMENON* only calculates paths per target statement up to a bound θ. Bounding path-sensitive analyses has shown to be effective in previous cases [28], where the bound was on the analysis time per path, rather than on the number of paths per targeted statement. We have found values of θ up to about 100 to be effective, which we will elaborate on further in our evaluation experiments.

V. Evaluation

To assess *PHENOMENON*, we study the following research questions:

RQ1: What is *PHENOMENON*'s accuracy in terms of its ability to produce information about Intents, the manner in which they control execution of different program paths, and the associated path conditions?

RQ2: What is *PHENOMENON*'s runtime efficiency? To what extent does *PHENOMENON* handle path explosion?

To answer these research questions, we implemented *PHENOMENON* in Java. For static analysis, we leveraged the Soot framework [31]. To transform the Android Dalvik format to an intermediate representation suitable for analysis in Soot, we utilized Dexpler [32]. To solve path expressions and processing of SMT-LIB expressions, we used the Z3 theorem prover [33]. Note that recent versions of Z3 have a built-in String type, which we leveraged for *PHENOMENON*.

A. RQ1: Accuracy of Generated Message-Control Information

To answer **RQ1**, we selected a set of apps listed in Table II from the open-source F-Droid repository [34]. For each app, we show the package name that uniquely identifies the app; a brief description of the app and its functionalities; the *version code* given to it in F-Droid; the app's size in terms of its source lines of code (*SLOC*); and the number of program paths containing Intent usage, or program paths executed based on Intents (*message control-based paths*). The F-Droid repository assigns a unique version code to every version of an app that it archives.

The set of apps shown in Table II meet several criteria that significantly aid in answering **RQ1**. First, each app belongs to a different application domain (e.g., security, file management, regional train tracking, etc.). They vary in their sizes in terms of SLOC—from 4KSLOC to over 460KSLOC. Most importantly, these apps exhibit sophisticated message usage by performing different operations along different program paths based on the Intents they receive, and the contents of those Intents. They include apps with over 2,600 program paths that involve message usage or message control.

For each app, we manually checked every program path—totalling 4,188 program paths—to determine if the expressions generated correctly describe the Intents and path conditions, particularly message-controlling path conditions. We checked the correctness of Intent information generated along a program path in the following conservative manner: If our analysis generated any extra information not valid for the path, we considered all of its information incorrect. For example, if any extra datum was missing along a program path, we considered the entire path incorrect. Consequently, we deem any partially correct expressions describing Intents or path conditions as completely incorrect. Furthermore, if extra information about an Intent was generated by *PHENOMENON*, we also consider all the Intent information generated for a program path as incorrect. For instance, a spurious extra datum that is described as belonging to an Intent is considered extra information and, for evaluation purposes, renders all information along the path as incorrect. To that end, we use the following correctness metric to assess the accuracy of *PHENOMENON* per app:

$$\text{Correctness Rate} = \frac{P_{cor}}{P_{tot}} \times 100$$

P_{cor} is the number of correct message control-based paths; P_{tot} is the total of number of message control-based paths.

The accuracy results that answer **RQ1** are shown in Table III. For each app, the table lists the number of paths with correct Intent information (P_{cor}), the number of paths with incorrect Intent information (P_{inc}), the total number of message-controlling paths (P_{tot}), and the correctness rate (*% Correct*).

PHENOMENON's correctness rate is very high with no app having a rate lower than 96%. Overall, this indicates that for the overwhelming majority of cases, *PHENOMENON* generates correct Intent information.

For **cri.sanity**, the following code snippet displays an interesting case for which *PHENOMENON* produces incorrect paths:

```
1 final boolean conn = BluetoothDevice.ACTION_ACL_CONNECTED
      .equals(act);
2 final int oldCount = pl==null? A.geti(K.BT_COUNT) : pl.
      btCount;
3 final int newCount = conn? Math.max(oldCount+1,1) : (Dev.
      isBtOn()? Math.max(oldCount-1,0) : 0);
4 if(oldCount == newCount) return;
5 A.putc(K.BT_COUNT, newCount);
```

This code snippet is from a Broadcast Receiver called **BtReceiver** in **cri.sanity**. On line 1, **act** is an action extracted from an Intent and is checked for equality against the String value **BluetoothDevice.ACTION_ACL_CONNECTED**. The result of that comparison is stored in **conn**. Line 3 depicts a ternary expression where the executed path is based on the value of **conn**. The particular manner in which message-controlling paths fork based on **conn** is not modeled by *PHENOMENON*, resulting in inaccuracies.

Another issue affecting correctness for *PHENOMENON* in **cri.sanity** involves string manipulation. In particular, some

TABLE II
APPS USED FOR ACCURACY EXPERIMENTS

App Package Name	App Description	F-Droid Version Code	SLOC	Message Control-Based Paths
com.samsung.srpol	List a device's app categories and permissions	9	4,649	47
com.naholyr.android.horairessncf	Search and track regional train in France	301	4,054	90
cri.sanity	Phone call, SMS, audio recording, and bluetooth management	21100	9,604	458
com.ghostsq.commander	Multi-protocol local and remote file manager	270	24,883	1,263
org.thialfihar.android.apg	Android port of OpenPGP for data encryption and decryption	11199	461,338	2,650

message-controlling paths occur based on partial values of an Intent's action. Specifically, in another Broadcast Receiver of **cri.sanity**, **Alarmer**, the **String** method **endsWith** is used to check if an Intent's action ends with the characters "Async". Given that *PHENOMENON* does not handle such string manipulations, Intent information along the resulting paths is incorrect.

com.ghostsq.commander's majority of issues arise due to the manner in which it uses **Parcelable** objects from the Android framework. The class of this object is designed to aid in marshalling and unmarshalling data sent between Intents. Given that certain classes of **com.ghostsq.commander** utilize **Parcelable**s to control execution of different paths, and **Parcelable**s are not explicitly modeled in *PHENOMENON*, we count these missing expressions as incorrect. Nevertheless, use of these objects to control execution of Android apps along different program paths is very rare; hence, the *PHENOMENON*'s accuracy results remain very high.

Another recurring pattern in **com.ghostsq.commander** that reduces accuracy for *PHENOMENON* involves **URI** objects. In particular, Intents may store **URI** information, which may point to files on a device, Content Providers storing information in specific apps, resources on the Internet, etc. We do not model API methods in *PHENOMENON* involving **URI**s since, as our results show, they rarely are used in conditional statements and, thus, only occasionally affect message-controlling paths. Nevertheless, *PHENOMENON* may model these cases by taking the Intent's **getData** method into account, which would be a relatively trivial modification to *PHENOMENON*.

PHENOMENON obtains incorrect results in certain cases for **android.apg**—which we will use here as a shortened form of **org.thialfihar.android.apg**—for two reasons: (1) *PHENOMENON* not modeling **URI**s of Intents; and (2) *PHENOMENON* not modeling MIME types of Intents. The MIME types of an Intent indicate the type of data that may have been supplied with an Intent or the type of data expected to be returned as a result. Although our results indicate that these objects have little effect on our overall results, we intend to extend *PHENOMENON* in the future to handle MIME types and **URI**s.

B. RQ2: Efficiency Results

To evaluate *PHENOMENON*'s efficiency, we ran *PHENOMENON*'s implementation on 100 apps randomly chosen from the F-Droid repository. Each app was run on a machine with four AMD Opteron 6376 2.3GHz 16MB Cache Sixteen-Core Processors and 256GB RAM. For this experiment, parallelization and upper bounds on the number of paths per target statement, which we refer to as *path upper-bounding*, as described in Section IV-D, was enabled to determine the overall speed of our analysis. Path upper-bounding was set to 100 paths per target statement, which covers a potentially large number of possible Intents that reach a target statement.

TABLE III
ACCURACY OF INTENT INFORMATION GENERATED FOR MESSAGE-CONTROL PATHS OF SUBJECT APPS

App Package Name	P_{cor}	P_{inc}	P_{tot}	% Correct
com.samsung.srpol	47	0	47	100.00%
com.naholyr.android.horairessncf	90	0	90	100.00%
cri.sanity	454	4	458	99.13%
com.ghostsq.commander	906	37	943	96.08%
org.thialfihar.android.apg	2,565	85	2,650	97.79%

PHENOMENON took on average 30.99 seconds to run per app on our evaluation machine. The minimum runtime was 11 milliseconds and the maximum runtime was 180.20 seconds. Besides parallelization and path upper-bounding, the low runtime is also due to the fact that selecting target statements, at the beginning of *PHENOMENON*'s analysis, significantly reduces the number of paths that need to be analyzed. Apps that use a fewer number of Intents have a lower runtime, compared to apps that use a greater number of Intents, since *PHENOMENON* prunes out paths that do not involve Intents.

Additionally, although path upper-bounding prevents certain paths that reach a target statement from being analyzed, the number of paths lost is relatively small (i.e., 320 paths across all apps). Consequently, path upper-bounding of 100 paths per target statement seems to be a reasonable tradeoff between efficiency and completeness for path selection.

VI. DISCUSSION AND LIMITATIONS

In terms of accuracy, the main threat to external validity is the number of apps we utilized for answering **RQ1**. To mitigate this threat, we selected apps varying across several dimensions, allowing us to draw more general conclusions. The apps come from different application domains, and they vary in size to as much as 460KSLOC and over 2,600 program paths involving Intent usage. In particular, **android.apg** is much larger than many apps on F-Droid, due to the fact that it intends to be a port of GnuPGP, which is an implementation of the OpenPGP standard.

Another threat to external validity is the fact that we selected apps that are all open-source and all from the F-Droid repository. Note that a key reason we selected open-source apps for evaluation is to allow us to carefully and manually inspect the correctness of information generated for over 4,100 program paths. Conducting such a manual analysis on disassembled code or intermediate representations (such as those produced for Soot) is intractable. Furthermore, by selecting apps that utilize a significant amount of Intent information, and particularly Intent information that affects execution of different program paths in an app, we select apps with sophisticated Intent usage. Note also that **android.apg** is a widely-used port [35] of a popular piece of encryption/decryption software, i.e., OpenPGP [36]. In fact, **android.apg** has up to 500,000 installs on devices, more than 3,300 reviews, and a 4.4 user rating, as of Aug. 26, 2016 [35].

A potential threat to construct validity is our selection of a simple correctness rate that assesses Intent information per program path. Note that for each path, a different set of Intents and their constituent attributes may exist. Furthermore, our metric takes both missing and spurious Intent information, i.e., expressions describing path conditions and Intent contents, into account—rendering the entire path as incorrect if any piece of information is incorrect.

A major challenge we faced was manually analyzing over 4,100 program paths, a large potentially error-prone effort. We mitigated the potential issues that may arise from conducting such a large-scale analysis in the following ways. Our experiments have been conducted over a year, where we learned to manually and correctly analyze program paths for Intent information and path conditions, and checked and re-checked results produced by *PHENOMENON*. We further leveraged experience building the ground truth for messages and their payloads obtained from previous work [26]. Given that program paths often tend to be variations of each other, for example many paths in an app are only slight variations of each other, we could leverage this information to simplify our correctness checks. At the same time, some paths can be empty in terms of path conditions, since some Intents may be constructed without having to use message control, further simplifying manual checking. We also make the logs of the apps we created accuracy experiments for, and information about correctness issues we discovered, available as artifacts for inspection, replication, or future research [6].

To ensure *PHENOMENON* itself remains accurate as it evolves, we constructed benchmark apps representing the kinds of paths and Intent usages we discovered in existing apps. This allowed us to run regression tests verifying correctness of *PHENOMENON* as it evolves. Building these benchmarks and running regression tests as *PHENOMENON* evolves has further aided us in effectively inspecting program paths for Intent information.

PHENOMENON currently focuses on Intent information and message control based on Intents received by an app. *PHENOMENON* does not model Intents and their contents as they are sent out of an Android component. However, *PHENOMENON* enables the analysis of the contents of Intents per program path that are needed to send any Intent. In fact, *PHENOMENON*'s target statements include every statement in an app that sends an Intent to another component. Consequently, *PHENOMENON* determines path conditions, especially those controlled by incoming Intents, that are needed to produce outgoing Intents.

One of the goals we aim to achieve by creating an approach like *PHENOMENON* is to enable and improve a variety of downstream analyses. This includes providing information about the message-controlling program paths of Android apps for purposes such as the following: automatic generation of exploits [37], [38], analysis of energy for Android apps [39], [40], improved testing along the message-based interface of Android [41], [24], field failure replication [42], etc. For security and automatic exploit generation, path-sensitive analysis has been critical for identifying exploits, as opposed to mere vulnerabilities [37]. Energy consumption of Android apps can vary significantly depending on the paths executed in an app [43], [44]. Furthermore, Intents are the most widely-used form of inter-component communication in Android [5].

VII. Conclusion

In this paper, we introduced *PHENOMENON*, an approach for obtaining path-sensitive Intent information, their contents, and the manner in which they control execution of different program paths in Android apps. *PHENOMENON*'s implementation obtains above 96% accuracy in five apps totalling over 4,100 program paths and ranging from 4KSLOC to over 460KSLOC. Additionally, it takes 31 seconds on average to run *PHENOMENON* on an app, using a machine with 32 cores and 256GB of RAM. These results indicate that *PHENOMENON* is effective for determining information about messages and message control in an Android app, which is useful for a variety of downstream analyses (e.g., automatic exploit generation, energy analysis, app testing, etc.).

In the future, we intend to utilize *PHENOMENON* to enable other analyses. For example, we aim to enable automatic exploit generation for Android apps, rather than just Linux binaries, as they have been used in previous work [37]. We further intend to extend our prior work on energy consumption analysis [43], [44] and test input generation for Android [45], [46] with path-sensitive Intent information.

References

[1] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 543–558. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534813

[2] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-component Communication Analysis," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 77–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818767

[3] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 576–587. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635869

[4] X. Cui, D. Yu, P. Chan, L. C. K. Hui, S. M. Yiu, and S. Qing, *CoChecker: Detecting Capability and Sensitive Data Leaks from Component Chains in Android*. Cham: Springer International Publishing, 2014, pp. 446–453. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08344-5_31

[5] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to practice," *Journal of Systems and Software*, vol. 119, pp. 31 – 44, 2016.

[6] "PHENOMENON - Website," http://tiny.cc/phenomenon.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[8] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer Science & Business Media, 2006.

[9] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 280–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818791

[10] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1329–1341. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660357

[11] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, formal synthesis and automatic enforcement of security policies for android," in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 514–525.

[12] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of android inter-app security vulnerabilities using covert," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 725–728. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819149

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[14] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2015. New York, NY, USA: ACM, 2015, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2771284.2771285

[15] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1043–1054. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516676

[16] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for Android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2015, pp. 461–471.

[17] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[18] R. Hay, O. Tripp, and M. Pistoia, "Dynamic Detection of Inter-application Communication Vulnerabilities in Android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 118–128. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771800

[19] J. D. Vecchio, F. Shen, K. M. Yee, B. Wang, S. Y. Ko, and L. Ziarek, "String Analysis of Android Applications (N)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 680–685.

[20] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String Analysis for Java and Android Applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 661–672. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786879

[21] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering (ICSE)*, 2015. [Online]. Available: http://web.cse.ohio-state.edu/~rountev/presto/pubs/icse15.pdf

[22] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[23] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Clapp: Characterizing loops in android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 687–697. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786873

[24] P. Eugster and K. R. Jayaram, *EventJava: An Extension of Java for Event Correlation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 570–594. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_26

[25] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic, "Impact analysis for distributed event-based systems," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12. New York, NY, USA: ACM, 2012, pp. 241–251. [Online]. Available: http://doi.acm.org/10.1145/2335484.2335511

[26] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic, "Identifying message flow in distributed event-based systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 367–377. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491462

[27] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[28] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 133–143.

[29] C. Barrett, P. Fontaine, and C. Tinelli, "The smt-lib standard version 2.6," 2010.

[30] L. de Moura and N. Bjrner, "Generalized, efficient array decision procedures," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, Nov 2009, pp. 45–52.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[32] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259056

[33] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[34] "F-droid," https://f-droid.org.

[35] "APG on Google Play," https://play.google.com/store/apps/details?id=org.thialfihar.android.apg\&hl=en.

[36] "Openpgp," http://openpgp.org/.

[37] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560217.2560219

[38] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[39] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 78–89. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483780

[40] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 92–101.

[41] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Nov 2015, pp. 429–440.

[42] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 474–484. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337279

[43] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of android apps," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, ser. GREENS '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 8–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820158.2820161

[44] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 425–436.

[45] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[46] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 559–570. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884853