2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Composite Constant Propagation: Application to Android Inter-Component Communication Analysis

Damien Octeau[1,2], Daniel Luchaup[1,3], Matthew Dering[2], Somesh Jha[1], and Patrick McDaniel[2]
[1]*Department of Computer Sciences, University of Wisconsin*
[2]*Department of Computer Science and Engineering, Pennsylvania State University*
[3]*CyLab, Carnegie Mellon University*
octeau@cs.wisc.edu, luchaup@andrew.cmu.edu, dering@cse.psu.edu, jha@cs.wisc.edu, mcdaniel@cse.psu.edu

***Abstract*—Many program analyses require statically inferring the possible values of composite types. However, current approaches either do not account for correlations between object fields or do so in an *ad hoc* manner. In this paper, we introduce the problem of composite constant propagation. We develop the first generic solver that infers all possible values of complex objects in an interprocedural, flow and context-sensitive manner, taking field correlations into account. Composite constant propagation problems are specified using COAL, a declarative language. We apply our COAL solver to the problem of inferring Android Inter-Component Communication (ICC) values, which is required to understand how the components of Android applications interact. Using COAL, we model ICC objects in Android more thoroughly than the state-of-the-art. We compute ICC values for 460 applications from the Play store. The ICC values we infer are substantially more precise than previous work. The analysis is efficient, taking slightly over two minutes per application on average. While this work can be used as the basis for many whole-program analyses of Android applications, the COAL solver can also be used to infer the values of composite objects in many other contexts.**

I. Introduction

Program analyses sometimes need to statically infer the possible values of object fields. Such a program analysis that has recently received interest [11], [15], [34] is the inference of messages communicated between Android applications. The components of Android applications can interact with one another using platform-specific constructs. This Inter-Component Communication (ICC) facilitates the reuse of functionality, both within and between applications. For example, an application may need to render a map centered on specific geographic coordinates. In Android, this application simply needs to send an ICC message, which will be relayed to an appropriate target by the operating system. The target will then render the map based on passed values.

This development model potentially presents concerns. First, exposed application components may be activated in unexpected ways, leading, for example, to privilege escalation attacks [14], [25]. Second, ICC messages can be intercepted by malicious recipients, with consequences ranging from data leaks [8] to piracy [29]. Finally, since information may flow between components, secure information flow analysis must account for inter-component flows. Without ICC analysis, in order to remain conservative, static taint analyses in Android have to assume that any data coming from another component is tainted [1]. With ICC analysis, such a taint analysis can precisely determine if inter-component links carry tainted data. Thus, ICC analysis has proven very valuable in many contexts such as information flow analysis [22], [24], [38], [41], patch generation for privilege escalation vulnerabilities [42] and detection of stealthy behavior [18].

In order to infer facts about interactions between components, we need to find all possible values of the fields of ICC objects at program points where message passing occurs. Unfortunately, existing studies of application interfaces are limited. The Epicc tool [34] tries to determine the specifications of ICC interfaces. Unfortunately, it only addresses *Intent* messages and a small subset of *URI* messages for which all fields are constant values. Adding complete support for *URIs* using the same approach as for Intents would result in a significant increase in the complexity of the formulation and implementation of the data flow functions. While this is possible in theory, it is not feasible in practice. Apposcopy [15] also infers Intent values but does not compute all fields of an Intent. In particular, similarly to Epicc URI data is not inferred.

In this paper, we define the problem of Multi-Valued Composite (MVC) constant propagation. Unlike most constant propagation analyses, we attempt to find all possible values of objects of interest at important program points, making our analysis *multi-valued*. Our analysis targets *composite* constants, i.e., we determine the values of complex objects that may have multiple fields, taking the correlations between fields into account. Problems are specified using the COAL declarative language. We design a COAL solver, which takes COAL specifications and programs as input and outputs composite constant values at program points of interest. In order to automatically generate data flow functions, it utilizes the concept of *field transformers*, which express how fields are changed by program statements.

While MVC constant propagation was motivated by Android ICC analysis, this work applies to a wide variety of static program analyses where the range of values of objects needs to be determined. It is valuable in various areas such as software security, maintenance and modeling. It can apply to many object oriented programming languages.

As an application of our composite constant propagation solver, we implemented and evaluated IC3, a new tool for Android ICC analysis. In the COAL language, we modeled all ICC messages with about 750 lines of COAL specification. Since Android ICC messages heavily rely on strings of characters, we devised and implemented a string analysis that is both efficient and more precise than the one in Epicc. We computed ICC values in 460 applications from the official Play store. We

```
1|void map(float latitude, float longitude) {
2|  Intent intent = new Intent();
3|  intent.setAction("VIEW");
4|  Uri geoUri = Uri.parse("geo:" + latitude + ","
 |      + longitude);
5|  intent.setData(geoUri);
6|  startActivity(intent); }
```

(a) Intent targeted at components that can render a map.

```
1|<activity android:name="MapRenderingActivity">
2|  <intent-filter>
3|    <action android:name="VIEW"/>
4|    <data android:scheme="geo"/>
5|    <category android:name="DEFAULT"/>
6|  </intent-filter>
7|</activity>
```

(b) Example Intent Filter declaration to receive the Intent in (a).

```
1|<activity android:name="DialerActivity">
2|  <intent-filter>
3|    <action android:name="VIEW"/>
4|    <action android:name="DIAL"/>
5|    <data android:scheme="tel"/>
6|    <category android:name="DEFAULT"/>
7|  </intent-filter>
8|</activity>
```

(c) Example Intent Filter declaration to dial phone numbers.

Fig. 1. Example Intent and Intent Filter used for rendering a map and for displaying a dialer. The real string values have been abbreviated for clarity.

inferred precise ICC values in 85% of cases. Epicc, on the other hand, could only infer 66% precisely. The remaining 15% of values could not be determined because of constructs not yet handled by our string analysis and other pathological cases. Computing ICC values is efficient, taking on average slightly over two minutes per application. The extra precision in inferring ICC values directly translated to a significant increase in precision when matching messages with potential receivers. Since the matching process is an overapproximation of actual runtime communication, having fewer links is desirable. In our experiments with 460 applications, such a matching yielded 120,817 links with ICC values computed by Epicc, whereas values computed with IC3 produced only 26,872 potential links. We make the following contributions:

- We introduce the MVC constant propagation problem.
- We define COAL, a declarative language to specify MVC constant propagation problems and query their solution.
- We formally define an approach to solve MVC constant propagation problems in an interprocedural, flow and context-sensitive manner. We implement a COAL solver based on this formalism and open source it at [32].
- We build IC3, an ICC analysis tool that relies on the COAL solver. As a part of IC3, we develop a string analysis that is finely tuned for the most typical cases found in Android applications. We make its source code available at [33].

II. A Motivating Example: Android ICC

Android applications are composed of four different types of components. *Activities* represent the user interface. *Services* are used for background processing. *Content Providers* allow for sharing of structured data between components. *Broadcast Receivers* receive messages sent to the entire system.

Components can communicate with one another using two kinds of objects. *Uniform Resource Identifiers* (URIs) are used to address data in Content Providers. *Intent* object are used in all other cases. The target component of an Intent can be specified by explicitly naming it, or it can be determined automatically by the operating system based on the fields of the Intent. An *Intent resolution* procedure maps a given Intent to possible targets. Several fields of an implicit Intent are used to match it with potential targets. The *action* field represents the operation that the receiving component should perform. The *categories* field adds information about the component that the system can use. For instance, the system places components with the `LAUNCHER` category in the main application launcher. The *data* field includes data that the receiving component should act on. It has the form of a URI.

Components can subscribe to receive implicit Intents by specifying Intent Filters, which describe the actions, categories and data of the Intents that should be addressed to them. Most Intent Filters are specified in the manifest file that is included with every application.

Figure 1 shows a representative example of Android ICC. In this figure and in the remainder of this paper, we abbreviate string values for ease of exposition. Figure 1(a) shows a method that sends an Intent in order to render a map centered at given coordinates. An Intent $intent$ is created. Its action is set to `VIEW`, which is a generic action used to display many kinds of data. The data of the Intent is defined to be a URI with the `geo` scheme followed by coordinates. When the $startActivity()$ framework method is called, the operating system (OS) resolves potential target components, prompting the user to choose a recipient if several components match.

Figure 1(b) is a component declaration as it can be found in an application manifest. The `activity` element (Line 1) declares that the application contains an Activity component with name `MapRenderingActivity` that includes a single Intent Filter. The `action` line specifies that the action field of Intents received by the component should have value `VIEW`. The `data` declaration at Line 4 specifies that any received Intent should carry data in the form of a URI with a `geo` scheme. Finally, the `category` line declares that received Intents can carry the `DEFAULT` category. This category is added by the OS to implicit Intents targeting Activities, such as the one on Line 6 of Figure 1(a). Therefore, `MapRenderingActivity` could receive the Intent created in Figure 1(a).

In order to statically know how application components communicate with one another, we need to determine the values of ICC objects at message-passing program points. For example, in Figure 1(a), we want to know all the possible values of $intent$ at statement `startActivity(intent)`. Objects of interest are Intents, Intent Filters and URIs. It is very challenging to write data flow models separately for all of these. That is why previous work [34] has not properly handled URIs, which has two negative consequences. First, interactions with Content Providers cannot be determined. Second, the data field of Intents cannot be known, which significantly limits the Intent resolution process. Any field that cannot be known results in a loss of precision. For example, mapping the Intent from Figure 1(a) with the component from Figure 1(b) requires knowing the action and the URI data of the Intent. When the data field is not known, any attempt to resolve the possible targets of $intent$ from Figure 1(a) has to conservatively assume that the data field can take any value. Figure 1(c) declares a dialer Activity `DialerActivity`. It is similar to `MapRenderingActivity`, except that it adds support for a `DIAL` action and it handles `tel` URI schemes. Because of

```java
public class Intent {
  private String action;
  private Set<String> categories = new
      HashSet<>();
  private String data;
  private String mimeType;

  public void setAction(String act) {
    this.action = act; }
  public void addCategory(String cat) {
    this.category = cat; }
  public void setDataAndType(String d,
                             String t) {
    this.data = d;
    this.mimeType = t; }
  public void setData(Uri u) {
    this.data = u.getData();
    this.mimeType = null; } }

public class Uri {
  private String data;

  public void setData(String d) {
    this.data = d; }
  public void getData() {
    return this.data; } }
```

(a) Simplified Intent and Uri classes. The real ones comprise 2,000 and 1,200 SLOC, respectively.

```java
 1 void sendMessage(Context c, boolean b,
                    String mimeType) {
 2   Intent intent = new Intent();
 3   intent.setAction("VIEW");
 4   Uri uri = new Uri();
 5   if (b) {
 6     intent.addCategory("BROWSABLE");
 7     uri.setData("http://icse-conferences.org");
 8     intent.setData(uri);
 9   } else {
10     uri.setData("file:///florence.jpg");
11     intent.setDataAndType(uri.getData(),
                            mimeType); }
12   c.startActivity(intent); }
```

(b) Message-passing code. We assume that the $mimeType$ argument may have value either image/jpg or image/*.

```
class Intent {
 String action; String categories; String data;
     String mimeType;

 mod <Intent: void setAction(String)> {
  0: replace action; }
 mod <Intent: void addCategory(String)> {
  0: add categories; }
 mod <Intent: void setDataAndType(String,String)> {
  0: replace data;
  1: replace mimeType; }
 mod <Intent: void setData(Uri)> {
  0: replace data, type Uri:data;
  clear mimeType; }
 query <Context: void startActivity(Intent)> {
  0: type Intent; } }

class Uri {
 String data;

 mod <Uri: void setData(String)> {
   0: replace data; }
 source <Uri: String getData(String)> {
   data; } }
```

(c) COAL specification for the constant propagation problem. Each modifier specification (mod) describes the influence of a method call on the fields of an Intent. A query indicates that all Intent values at calls to $startActivity()$ should be computed. A source indicates how the value of a field flows out of an object.

	Value 1	**Value 2**	**Value 3**
action	VIEW	VIEW	VIEW
categories	{BROWSABLE}	∅	∅
data	http://...	file:///...	file:///...
mimeType	∅	image/jpg	image/*

(d) Possible values of the fields of $intent$ at the $startActivity()$ call. We have abbreviated the URI strings for space reasons, however our system would infer complete strings. Value 1 is for the first branch after the if statement (Lines 6-8 in (b)). Values 2 and 3 account for the fall-through branch of the if statement, where argument $mimeType$ may have two different values.

Fig. 2. Running example.

its inability to infer Intent URI data, the current state-of-the-art [34] would infer that both MapRenderingActivity and DialerActivity can receive the Intent. In reality, only the former is able to do so. Thus, more precision is needed to avoid such false positives.

We address this issue in this paper. In Sections III through VI, we introduce a novel approach to statically infer the set of values that objects can take. In Section VII, we apply this approach to the problem of inferring Android ICC values.

III. Overview

A. The MVC Constant Propagation Problem

Consider OBJ an object of type class Pair{int X; int Y;}. Assume that at some program location OBJ can be either (X, Y) = (1, 10) or (2, 20). We would like an analysis that can determine this fact. Classical constant analysis applied for each field fails at determining a useful value because none of the fields is the same constant across all paths. Multi-valued constant analysis could determine that OBJ.X $\in \{1, 2\}$ and OBJ.Y $\in \{10, 20\}$. These constraints accurately describe the individual fields, but they allow for imprecision in the object, because they allow the possibility that OBJ = (1, 20). We define the *Multi-Valued Composite (MVC)* constant propagation problem to be the problem of determining the set of values that an object *viewed as a tuple* (such as (X, Y)) can have. Note that the above multi-valued constant analysis applied to individual fields is a possible solution, it may just not be precise enough for certain analyses. We will show how to efficiently find more precise solutions.

We now introduce a running example that will be used throughout. Figure 2(a) shows code for a hypothetical Intent class that contains data used for passing messages between application components. It uses a data field which is copied from a Uri object, for which code is also shown in Figure 2(a). Figure 2(b) defines method $sendMessage()$, which we assume to be called as part of an Android application. This method creates an Intent object and sets its *action* field. Then, depending on the value of a Boolean, one of two things can happen. In the first branch after the if statement, a value is added to the *categories* field of $intent$. Then the *data* field of a Uri object is copied to the *data* field of $intent$ at Line 8. In the fall-through branch, the *data* and *type* fields of the $intent$ variable are set using a call to $setDataAndType()$ (Line 11). Finally, the Intent object is sent to another component using the $startActivity()$ method.

The data flow problem we are solving is to determine all the possible values of the fields of $intent$ at the call to $startActivity()$. In our propagation framework defined below, the problem can be specified using COAL, a declarative language we designed for this purpose. *The function of COAL (COnstant propAgation Language) is to specify Multi-Valued Composite (MVC) constant propagation problems*. It specifies the types of variables for which values should be inferred

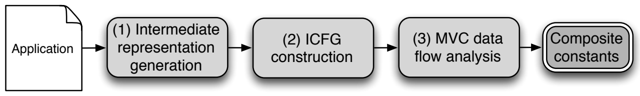
Application
(1) Intermediate representation generation
(2) ICFG construction
(3) MVC data flow analysis
Composite constants

Fig. 3. General overview of the MVC constant analysis process.

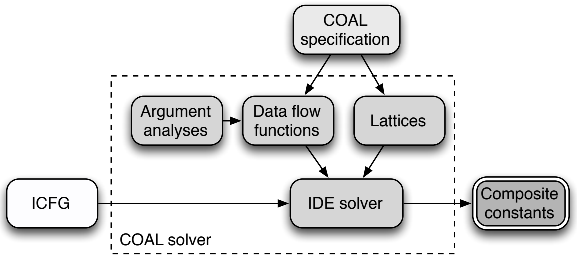

Fig. 4. The MVC data flow analysis process (Step 3 from Figure 3).

and how these values are modified by program statements. It enables abstract reasoning on the semantics of API methods. The COAL language is recognized by our COAL solver, which outputs solutions for many propagation problems solely from their COAL specification and an input program.

Figure 2(c) shows how to specify the problem with our framework using COAL. The COAL specification is manually written once and it can subsequently be used to solve the same problem for an arbitrary number of applications. It is composed of field declarations, modifiers and a query. The field declarations specify the fields that are being tracked and their type. Note that, for each field, we keep track of sets of values, even though the field declaration only specifies the type of each individual field value. The first modifier indicates how the $setAction()$ method influences the modeled value of an Intent object. A modifier specification starts with the signature of the modeled method. Each line in a modifier declaration is an argument whose value is used to modify the Intent value. Each argument declaration is composed of several attributes. An integer declares the position of the argument in the array of arguments to the method, with indices starting at 0. After the argument index, an operation and a field are declared. They describe both the field that is modified by the method and how it is modified. For example, in the $setAction()$ modifier, `0: replace action` means that the *action* field is replaced with the value of the first argument to $setAction()$. Other modifiers are declared in a similar manner, except when the type of an argument is a class that is modeled with COAL. In that case, a `type` attribute is used in order to specify which field of the argument object is used. For example, in the $setData()$ modifier, the `0: replace data, type Uri:data` argument means that the *data* field of the Uri argument is used to replace the *data* field of the Intent being modified.

The query statement indicates that we are querying the solution at all calls to $startActivity()$. Similarly to the modifier declaration, we specify a list of arguments. They describe the arguments whose value we would like to query. In this case, it is the first argument (as described by the `0` attribute), which is an Intent object. The source at Line 22 indicates how a field value flows out of an object. This is useful when the value subsequently flows into a COAL modifier, since the COAL solver can then infer the correct value.

Figure 2(d) shows the expected result of our analysis. We want our analysis to recover the three possible values of Intent $intent$. These values correspond to all possible execution paths of the program from Figure 2(b). We wish to recover exactly these possible values, and we do not want all the possible combinations of fields. For example, it is not possible in our problem to have an Intent value with category `BROWSABLE` and MIME type `image/jpg`. As a result, our analysis does not simply track fields individually as separate variables, but rather propagates composite constants.

B. MVC Constant Propagation Analysis

Figure 3 shows a general overview of the analysis process that takes an application as input and outputs the values of composite objects. It starts by converting the program to an intermediate representation that is suitable for further analysis (Step 1). It then generates an *Interprocedural Control Flow Graph (ICFG)* (Step 2). An ICFG is a collection of CFGs of all the procedures in the program connected with each other at procedure call sites. This includes building a call graph for the entire program. Finally, the actual MVC data flow analysis takes place in Step 3 and outputs the MVC constant values.

Existing work [1], [30] can perform Steps 1 and 2. Therefore, the scope of this paper is limited to the MVC data flow analysis (Step 3), which is performed using our COAL solver. Figure 4 depicts a more detailed overview of the COAL solver, which takes two inputs. First, it uses the ICFG of the program being analyzed. Second, it takes a COAL specification for the problem being solved. This COAL specification describes the structure of the composite objects for which constant propagation should be performed. It also describes the methods that can modify these objects and the program locations where the constants should be computed. The specification is written using the COAL language, which allows MVC constant propagation problems to be specified easily. It should be noted that, for a given problem, the COAL specification need only be written once, after which an arbitrary number of programs can be analyzed.

For each program, the COAL specification is parsed to build a model of the problem by creating problem-specific lattices of values and data flow functions. These are input with the ICFG into a solver for Interprocedural Distributive Environment (IDE) problems [36]. We present the generic IDE model for constant propagation in Section V. Finally, since the values of arguments to functions have to be known, we use argument value analyses (e.g., integer and string analyses) to generate the data flow functions.

The IDE solver outputs the analysis results. The COAL language allows specification of program points of interest (queries) where the MVC constant values should be computed. This is useful in cases where the program points of interest are known in advance. In other cases, we also allow lower-level queries to the IDE solver as part of the COAL solver API. The results can then be output in a simple text format or accessed using a programmatic interface (API).

Note that it is possible to write MVC data flow models for individual classes in an *ad hoc* manner without using an MVC constant solver. However, given the complexity of a typical MVC model, doing so is time consuming and prone to errors. In contrast, the approach we present relies on a

```
⟨model⟩    ::= 'class' ⟨type⟩ '{' { ⟨field⟩ | ⟨modifier⟩ | ⟨query⟩ | ⟨constant⟩
               | ⟨source⟩ } '}'
⟨field⟩    ::= ⟨type⟩ ⟨field name⟩ ';'
⟨modifier⟩ ::= 'mod' ⟨method sig⟩ '{' { ⟨modifier arg⟩ } '}'
⟨query⟩    ::= 'query' ⟨method sig⟩ '{' { ⟨query arg⟩ } '}'
⟨constant⟩ ::= 'constant' ⟨field sig⟩ '{' { ⟨field name⟩ '=' ⟨inline value⟩
               ';' } '}'
⟨source⟩   ::= 'source' ⟨method sig⟩ '{' ⟨field name⟩ ';' '}'
⟨modifier arg⟩ ::= [⟨arg number⟩ ':'] ⟨operation⟩ ⟨field⟩ [',' ⟨arg type⟩ ':'
               ⟨field name⟩]
⟨query arg⟩ ::= ⟨arg number⟩ ':' ⟨arg type⟩
⟨arg number⟩ ::= ⟨integer⟩ | '(' ⟨integer⟩ {',' ⟨integer⟩ } ')'
⟨arg type⟩ ::= 'type' ⟨type⟩
⟨field sig⟩ ::= '<' ⟨type⟩ ':' ⟨type⟩ ⟨field⟩ '>'
```

Fig. 5. COAL language for specifying MVC constant propagation problems.

single, generic and formally defined MVC data flow model. Instead of specifying data flow models for each class being analyzed, MVC constant propagation problems are specified by writing COAL specifications. This approach has several advantages over the *ad hoc* one. First, it is easier to implement and maintain, since COAL specifications are written using a simple declarative language. Second, it is easier to ensure that the models and their implementation are correct. Correctness of the generic model is discussed in Section VI-A and in an accompanying technical report [31]. Since COAL specifications only describe the relationships between fields and methods without specifying semilattices or flow functions, their correctness is relatively simpler to verify. Third, the generic data flow model used by the COAL solver can be changed without having to rewrite all the COAL specifications that have been written so far. In particular, we are currently using an IDE model [36], but it is possible to use reductions to other types of problems [35]. Finally, when modeled types interact with one another the data flow functions in the *ad hoc* approach become much more complex. On the other hand, the fixed point iteration described in Section VI-C supported by the COAL constructs demonstrated at Line 12 of Figure 2(c) enables seamless support for inter-object data flows.

IV. The COAL Language

The goal of the COAL language is to specify and query a wide variety of MVC constant propagation problems. COAL specifications are used by our solver to automatically generate semilattices of values and data flow functions.

A simplified grammar for this language is presented on Figure 5. The {} characters symbolize repetition, while [] characters surround optional parts of a production.

The model for a given object is composed of field declarations, modifiers, constants and sources. Queries can also be specified using the COAL language to specify program points where MVC constants should be inferred.

Field declarations - A field declaration specifies a field that is part of the modeled class. It describes a data type and a name for the field. In Figure 5, we use non-terminals ⟨*type*⟩ and ⟨*field name*⟩ to represent valid types and field names.

Modifiers - Modifiers represent method calls where constant values flow to the modeled object. The specification of the modifiers comprises a method signature ⟨*method sig*⟩ that identifies the method of interest. It also includes a set of arguments that describe how the method arguments are used to modify the fields of the modeled object. A modifier argument has several attributes. An argument index identifies the method argument of interest. In some cases, several arguments contribute to the value of a single field. That is why the language supports sets of argument indices. A field operation to be performed is also specified. This allows the solver to create appropriate data flow functions. Natively supported field operations are `add` (add argument value to the field), `remove` (remove argument value from field), `replace` (replace field with argument value) and `clear` (clear field value). A modifier specification also includes a field name that identifies the field being modified. In the case where an argument is a class modeled with COAL, an argument type and additional field name are specified. This indicates to the solver that the value of a field of a modeled class flows to the object being modified.

Constants - Many languages allow the specification of constants (e.g., static final fields in Java). The constants of a class are initialized in the class initializer the first time the class is referenced. A naïve way to deal with constants would consist in tracking the constant creation and initialization as it is done for all modeled objects. We would then propagate them throughout the entire program at the cost of performance. As a performance optimization, we allow constant modeled objects to be specified in COAL. Where these values are used, the COAL solver uses the specified value.

Sources - Sources model the case where a modeled field value flows to an argument value. For example the source at Line 22 of Figure 2(c) allows the COAL solver to infer that the *data* field of the Uri object flows to the $intent$ variable. Using this information allows the solver to infer the correct value for the *data* field of $intent$.

Queries - Queries specify statements of interest where modeled values should be determined.

The MVC constant propagation problem from Figure 2(b) can be solved by inputting the program and the specification from Figure 2(c) into our COAL solver. Alternative methods such as code annotations could be used to specify these problems. However, our approach specifies all analysis parameters in a single location and does not require the source code of the modeled objects. Annotations, on the other hand, would require source code and they would have to be spread over the modeled code. In our motivating example of Android, this implies spreading specifications over thousands of lines code.

V. A Generic Model for MVC Constant Propagation

The purpose of the COAL language and the associated constant propagation solver is to determine the possible values of composite objects by only defining a COAL specification. The COAL solver automatically converts the COAL specification to an instance of an Interprocedural Distributive Environment (IDE) problem, using the model defined in Section V. Given an IDE problem, existing algorithms can compute a solution [36]. This section presents the analysis domain and a space F of functions that model the influence of COAL modifiers. They will subsequently be used in Section VI to automatically build reductions to IDE problems. Interested readers are referred to the technical report [31] for a more detailed presentation.

A. The L Semilattice of Values

For any set X, we denote the power set of X by 2^X and the set of functions from X to X by X^X. We are trying to determine the value of an object with n fields, taking values in finite sets $V_1, \ldots, V_n$. If field i has a container type (e.g., set of integers), V_i is the contained type (e.g., integer). For $i \in \{1, \ldots, n\}$, let $P_i = 2^{V_i} \cup \{\omega\}$, where ω represents an undefined value. Let $B = P_1 \times \cdots \times P_n$. We define $L = (2^B, \subseteq)$ a join-semilattice with a bottom element $\bot = \varnothing$. The join operation on L is the set union $\cup$. The top element of L is the set of all elements in B.

Sets $V_1, \ldots, V_n$ are the domains of the field values we are trying to determine. For example, V_1 could be the set of constant strings of characters in the program. A value in B represents a value as it is seen on a single path. Finally, values in L represent values of objects, taking into account several paths of a program. Note from the definition of $P_1, \ldots, P_n$ that we keep track of sets of values, even for scalar fields. A scalar is simply represented by a set with a single value.

Let us consider the example from Figure 2(a). We are interested in four fields: *action*, *categories*, *data* and *mimeType*. Let $\mathcal{S}$ be the set of string constants in the program. In this case, we consider $P_1 = P_2 = P_3 = P_4 = 2^{\mathcal{S}} \cup \{\omega\}$. In other words, we consider all four fields to take values in the power set of $\mathcal{S}$. We have $B = P_1 \times P_2 \times P_3 \times P_4$ and $L = (2^B, \subseteq)$.

In method $sendMessage()$, the value associated with the $intent$ variable is initially $\bot$ before Line 2. Line 2 transforms this value to $\{(\varnothing, \varnothing, \varnothing, \varnothing)\}$. Right after Line 3, the value is

$$\{(\{\texttt{VIEW}\}, \varnothing, \varnothing, \varnothing)\}. \tag{1}$$

After the `if` statement, at Line 12, the value is

$$\begin{aligned}\{&(\{\texttt{VIEW}\}, \{\texttt{BROWSABLE}\}, \{\texttt{http://icse-conferences.org}\}, \varnothing),\\ &(\{\texttt{VIEW}\}, \varnothing, \{\texttt{file:///florence.jpg}\}, \{\texttt{image/jpg}\}),\\ &(\{\texttt{VIEW}\}, \varnothing, \{\texttt{file:///florence.jpg}\}, \{\texttt{image/*}\})\}.\end{aligned} \tag{2}$$

B. Transformers on L

The intuition behind the COAL language is that each argument in a COAL modifier represents the influence of the method call on a field. Accordingly, we introduce transformers that are defined at the granularity of fields. In this section, we assume that the value of uri is available where necessary. We revisit this assumption in Section VI-C.

Definition 1 - *For $i \in \{1, \ldots, n\}$, we define F_i a non-empty subset of $P_i^{P_i}$ closed under composition. Each $\phi \in F_i$ is called a* **field transformer**.

In this paper, we consider field transformers ϕ such that:

- Type (1): $\phi(\omega) = \omega$ and for all $X \in P_i$ such that $X \neq \omega, \phi(X) = (X - KILL) \cup GEN$, for some constant sets GEN and $KILL$ in P_i. Such a function will also be denoted as $\phi = \phi_{GEN}^{KILL}$. This is used for the `add` and `remove` field operations in COAL.
- Type (2): For all $X \in P_i$, $\phi(X) = GEN$, for some GEN in P_i. This case is also denoted by $\phi = \phi_{GEN}$. This is used for the `replace` and `clear` field operations in COAL.

It is easy to verify that the set of such field transformers is closed under composition.

Let us denote the identity field transformer by id. We have $id = \phi_{\varnothing}^{\varnothing}$. The important idea is that each modifier argument in COAL is mapped to a single field transformer. For example, let us consider the statement at Line 3 of Figure 2(b). Using the definition above and the fact that this method replaces the existing action value, we can model it using type (2) field transformer $\phi_{\{VIEW\}}$.

Field transformers are used as basic building blocks for data flow functions. We define the set $\mathcal{L}$ of functions from B to B such that for any $l \in \mathcal{L}$, there exists $(\phi_1, \ldots, \phi_n) \in F_1 \times \cdots \times F_n$ such that, for any $b = (\beta_1, \ldots, \beta_n) \in B$, $l(b) = (\phi_1(\beta_1), \ldots, \phi_n(\beta_n))$. We note $l = \phi_1 \times \cdots \times \phi_n$. Recall that the influence of the statement at Line 3 of Figure 2(b) on field *action* is modeled by field transformer $\phi_{\{VIEW\}}$. The function in $\mathcal{L}$ that models the influence of the $setAction()$ call on the *action* field is quite naturally $\phi_{\{\texttt{VIEW}\}} \times id \times id \times id \in \mathcal{L}$. This function solely modifies the *action* field.

Functions in $\mathcal{L}$ model the influence of a single execution path. We can define their composition as follows. For any $l_1 = \phi_1^1 \times \cdots \times \phi_n^1$ and $l_2 = \phi_1^2 \times \cdots \times \phi_n^2$ in $\mathcal{L}$, we have:

$$l_1 \circ l_2 = \phi_1^1 \circ \phi_1^2 \times \cdots \times \phi_n^1 \circ \phi_n^2.$$

Using Definition 1, $\mathcal{L}$ is closed under composition.

We now define a set F of functions from L to L using functions in $\mathcal{L}$. Functions in F can model the influence of several execution paths on all fields of an object. More specifically, any $f \in F$ is written $f = \{l_1, \ldots, l_m\}$, with $l_1, \ldots, l_m \in \mathcal{L}$, such that:

- for any $b \in B$, $f(\{b\}) = \bigcup_{i=1}^m l_i(b)$,
- for any $v = \{b_1, \ldots, b_k\} \in L$, $f(v) = \bigcup_{i=1}^k f(\{b_i\})$.

The identity over L is denoted by id_L. Additionally, F contains Ω, which is such that for all $v \in L$, $\Omega(v) = \bot$. Informally, the Ω function is used to "kill" data flow facts. This only happens when a variable is assigned a new value. Finally, F contains $init_v$ functions, which are such that $init_v(\bot) = v$, with $v \in L$. Informally, $init$ functions generate data flow facts and associate them with an initial value.

Let us now consider the `if` statement in Figure 2(b). The influence of the two branches is summarized by

$$\begin{aligned}\{&id \times \phi_{\{\texttt{BROWSABLE}\}}^{\varnothing} \times \phi_{\{\texttt{http://icse-conferences.org}\}} \times \phi_{\varnothing},\\ &id \times id \times \phi_{\{\texttt{file:///florence.jpg}\}} \times \phi_{\{\texttt{image/jpg}\}},\\ &id \times id \times \phi_{\{\texttt{file:///florence.jpg}\}} \times \phi_{\{\texttt{image/*}\}}\},\end{aligned} \tag{3}$$

where $\phi_{\varnothing}$ clears the value of the *mimeType* field. We can verify that applying this function to the value given by Equation (1) yields the value given by Equation (2).

By defining the composition of elements of F in a standard way, it is possible to prove the following proposition [31], which will be used in the next section.

Proposition 1 - *F is closed under composition.*

Finally, we define the $\cup$ operator such that, for $f_1 = \{l_1^1, \ldots, l_m^1\}$ and $f_2 = \{l_1^2, \ldots, l_k^2\}$, $f_1 \cup f_2 = \{l_1^1, \ldots, l_m^1, l_1^2, \ldots, l_k^2\}$.

VI. From COAL Specifications to IDE Problems

This section presents how COAL specifications are used to automatically generate instances of IDE problems by generating data flow functions in F. Recall that IDE problems can then be solved using existing algorithms [36]. We first outline the requirements of IDE problems.

A. Environment Transformers

Let D be the set that comprises all variables in the program and a special Λ symbol, which represents the absence of a data

Algorithm 1 Generate functions in F from COAL modifiers.

```
procedure GenerateFunction($modifier$, $statement$)
  $result := id_L$
  for all arguments $arg$ in $modifier.args$ do
    $values :=$ null
    if $arg.number$ != null then
      $values$ := GetArgumentValues($statement$, $arg.number$, $arg.type$)
    $arg\_function :=$ null
    if $values \neq$ null then
      for all argument values $value$ in $values$ do
        $current :=$ BuildFuncInF($arg.op$, $value$, $arg.field$)
        if $arg\_function =$ null then
          $arg\_function = current$
        else
          $arg\_function = arg\_function \cup current$
    else
      $arg\_function :=$ BuildFuncInF($arg.op$, null, $arg.field$)
    $result := result \circ arg\_function$
  return $result$
```

flow fact. An *environment* is a function from D to L, where L was introduced in Section V-A. The set of environments is E. A join operation $\sqcup$ is defined on E such that, for any $e_1, e_2 \in E$ and $d \in D$, $(e_1 \sqcup e_2)(d) = e_1(d) \cup e_2(d)$. *Environment transformers* are used to model the influence of program statements on the values of variables. They are functions from E to E. For example, before a program statement s, the values associated with each variable of interest are given by environment $e_1 \in E$. Statement s transforms e_1 to a new environment $e_2 \in E$, which is modeled by an environment transformer t such that $e_2 = t(e_1)$. The IDE framework requires that environment transformers be *distributive*. An environment transformer t is said to be distributive if, for every $e_1, e_2, \cdots \in E$ and for any $d \in D$, $(t(\sqcup_i e_i))(d) = \cup_i (t(e_i))(d)$.

The main function of the COAL solver is to turn COAL specifications into distributive environment transformers. We can show that the functions in F are distributive and that they can be used to easily build distributive environment transformers [31], which ensures the correctness of our approach.

B. Generating Functions in F

Since producing environment transformers from functions in F is trivial, this section addresses how the COAL solver builds elements of F from COAL specifications. Algorithm 1 is used by the COAL solver to generate a function in F from a statement and a modifier specification for the statement. It computes functions in F for each argument and composes them (recall from Proposition 1 that F is closed under composition). A modifier argument arg has several attributes: (i) an operation op, which is performed by the modifier method, (ii) an argument number $number$, which indicates the position of the arguments of interest in the modifier method, (iii) an argument type $type$, which can be declared as part of the field declaration (see Line 2 of Figure 2(c)) and (iv) $field$, the index (or the name) of the modified field.

We assume the existence of a procedure GetArgumentValues, which computes the possible values of a method argument, given an invoke statement, an argument number and an argument type. For most value types, this procedure simply traverses the interprocedural control flow graph starting at the method call looking for assignments to the variable that is used as an invocation argument. For string arguments, we use the analysis described in Section VII-A. Note that the argument type is needed by the COAL solver to select the argument analysis that should be used. We also assume that there is a procedure BuildFuncInF that generates a function in F given an operation, an argument value and a field. In the interest of space, we only summarize its main steps. It starts by generating a field transformer ϕ using the operation and the argument value. The field index (or name) allows the creation of a function $l \in \mathcal{L}$ of the form: $l = id \times \cdots \times id \times \phi \times id \times \cdots \times id$. The corresponding function in F is simply $\{l\}$. When a modifier method argument may have several values resulting in possible functions $f_1, \ldots, f_n$, we compute $f_1 \cup \cdots \cup f_n$ (Line 14).

To illustrate this procedure, let us consider Line 11 of Figure 2(b). The COAL solver determines that this is a modifier with two arguments (see Figure 2(c) Lines 8-10). Considering the first argument `0: replace data` and given the fact that *data* is a string field, the GetArgumentValues procedure finds that the method argument has value `file:///florence.jpg`. Since a `replace` operation is requested, the BuildFuncInF procedure generates field transformer $\phi_{\{\texttt{file:///florence.jpg}\}}$. Using the fact that *data* is the third field (Line 2 of Figure 2(c)), it generates function

$$\{id \times id \times \phi_{\{\texttt{file:///florence.jpg}\}} \times id\}. \tag{4}$$

Considering argument `1: replace mimeType`, the solver finds that there are two possible values for the $mimeType$ variable. Thus, Lines 9-14 of the algorithm yield function

$$\{id \times id \times id \times \phi_{\{\texttt{image/jpg}\}}, id \times id \times id \times \phi_{\{\texttt{image/*}\}}\}, \tag{5}$$

where Line 14 utilizes the definition of the $\cup$ operator on F from Section V. Finally, Line 17 of Algorithm 1 composes the two functions given by Equations (4) and (5), which yields:

$$\{id \times id \times \phi_{\{\texttt{file:///florence.jpg}\}} \times \phi_{\{\texttt{image/jpg}\}}, \\ id \times id \times \phi_{\{\texttt{file:///florence.jpg}\}} \times \phi_{\{\texttt{image/*}\}}\}.$$

C. Fixed Point Iteration

Let us consider method $sendMessage()$ from Figure 2(b). So far, we have assumed that the value of the Uri uri at Line 8 of Figure 2(c) is available when we generate field transformers for $intent$. In reality, it is not initially available, because when we solve the problem for the first time, values for $intent$ and uri are computed in the same iteration. Thus, in order to fully resolve all values, we run several iterations of the COAL solver. For example, in the first iteration, the transformer that is generated for statement `intent.setData(uri)` is

$$\{\phi_{intent,1} \times \phi_{intent,2} \times \phi_{intent,3} \times \phi_{intent,4}\} = \\ \{id \times id \times id \times \phi_{uri,1}\},$$

where $\phi_{uri,1}$ is a transformer that indicates that the value of the *data* field of $intent$ refers to the first field of Uri uri. We initially start with $\phi_{intent,i}$ and $\phi_{uri,1}$ mapping to ω, for $1 \leq i \leq 4$. We then iterate until a fixed point is reached for $\phi_{intent,i}$ and $\phi_{uri,1}$. The same process allows the solver to resolve the value of $intent$ at Line 11 of Figure 2(b).

VII. Application to Android ICC

As an application of the COAL language and solver, we built IC3 (Inter-Component Communication analysis with COAL), an ICC inference tool that is based on COAL specifications. The main ICC classes are Intents, Intent Filters and URIs. For completeness we also model the Component Name, Bundle, Pending Intent and Uri Builder classes since they are referenced by the main class types.

Recall from Figure 3 that as a prerequisite to the MVC constant propagation, it is necessary to generate an intermediate representation (IR) that is suitable to generate an ICFG. The COAL solver is currently implemented using the Soot framework [39] and the Heros IDE solver [3]. Soot converts Java bytecode to an internal IR that is recognized by its Spark [23] call graph construction module, which is used to build an ICFG. However, Android applications present additional challenges. First, they are distributed in a platform-specific bytecode format. We therefore preprocess them with Dare [30], which converts Android to Java bytecode. Second, Android applications are composed of components that may be started in an arbitrary order. Additionally, they are event-based programs that declare callbacks that may be called in an arbitrary order. In order to address this challenge in a conservative manner, we adopt the call graph construction procedure from FlowDroid [1], which generates a wrapper entry point method that simulates the application lifecycle and the arbitrary event and component call order.

The COAL solver takes aliasing into account. When a method modifies a variable $o1$ that is a possible alias for another object $o2$, our analysis generates two values for $o2$. One of them takes the call into account and the other one does not. The one that does not models the case where the alias analysis results in a false positive (i.e., detecting that a value may point to a certain heap location even though it does not). This is similar to the standard idea of weak updates [7].

A. String Analysis

Strings are ubiquitous in Android applications. Many arguments to ICC methods are strings. Because of the limited set of predefined Intent fields (e.g., default action and category strings), in many cases, the value of string fields is determined by a finite set of constants. However, the way these constants are transferred or combined is not trivial and a string analysis is required to determine the set of possible values that a given variable can have. Our string analysis determines a safe overapproximation of such sets. It was inspired by JSA [9], although our analysis is highly customized for the purposes of Android. Conversely, JSA is more generic but significantly slower for our purposes. Our analysis works in two stages: constraint generation and constraint solving. Constraint generation simply gathers the dataflow facts for string variables. Constraint solving determines regular sets (described as regular expressions) that satisfy the constraints.

In the first stage we generate constraints for all string operations, modeling the String and StringBuilder classes. Our goal is to have a representation that can be used either by a constraint solver or by abstract interpretation. This is why the constraints are a symbolic representation of the original program operations. The analysis is flow-sensitive. Constraints model idioms that are common in Android operations: concatenation, string fields, function calls, etc.

In the second stage, a solver uses the constraints to answer queries about variable values. As a proof of concept, we implemented a simple solver that given a variable $\overline{x}$ produces a regular expression that overapproximates the set of values that $\overline{x}$ can take. It works by finding the constraints associated with $\overline{x}$ and by traversing the flow graph and interpreting the nodes. We avoid non-termination by detecting self dependence cycles and widening to `.*` (that is $\perp$). Similarly, we widen to `.*` when we detect calls to functions outside the analysis (for which we have not generated constraints). Although our widening method may be less accurate than that in [9], our simple solver is faster and can still be more accurate because of context sensitivity. Our analysis is interprocedural, context-sensitive, flow-sensitive, field-sensitive but not object-sensitive. Additional details about the string analysis can be found in the technical report [31].

B. Evaluation

The evaluation of our approach was aimed at answering three central questions:

Q1: Can IC3 precisely infer field values of ICC objects?

Q2: As an application of our analysis, how precisely can ICC messages be matched with their targets?

Q3: Are the computational costs of IC3 feasible in practice?

The answer to these questions determines how effectively our analysis can be used as the basis of inter-component analyses. Highlights of our evaluation are:

- IC3 infers precise field values for 85% of ICC values. Epicc can only infer 66%. This is a significant increase in precision.
- When matching components that may communicate with one another, specifications from IC3 lead to 78% fewer component links than the current state-of-the-art.
- On average, our analysis takes 140 seconds per application. This makes it feasible in practice to use our analysis as the first step of inter-component analyses.

For performance reasons, we generally do not allow the constant propagation to analyze the Android framework code. The only exception is when a framework class may create or modify ICC objects, which only occurs in a few classes of the framework. In the few cases where ICC method arguments are not strings of characters (e.g., integer arguments), we use a simple analysis that looks for definitions of constant values for that argument. It simply traverses the interprocedural control flow graph starting at the method call, keeping track of all possible values. When a constant value cannot be found, a special ω value is conservatively returned.

We performed our experiments on a corpus of 500 applications. They were randomly selected from a set of 453,525 applications downloaded from the Google Play store between January and September 2013. Some application could not be processed because of insufficient memory errors or timeout, so we report numbers for 460 applications.

Precision of field values - We first measured the precision of the fields of the ICC values inferred by IC3 at program points of interest (i.e., sending a message, or programmatically registering a component with an Intent Filter). The list of these program points is given in [31]. We counted the number of ICC values inferred by IC3 and Epicc [34] for which no field value used for Intent or URI resolution is completely unknown (e.g., a `.*` string value). We modified Epicc such that it used the same entry point construction procedure from [1]. The precision results are presented in Table I. The third line shows the results for Intents and Intent Filters, whereas the fourth line shows statistics for URIs. The *value count* column shows the

TABLE I
ICC VALUE FIELD PRECISION RESULTS.

	Value count	ICC values with precise fields		ICC values with imprecise fields		Missing ICC values	
		Epicc	IC3	Epicc	IC3	Epicc	IC3
Intents & Filters	5,306	3,660 (69%)	4,575 (86%)	1,474 (28%)	662 (12%)	172 (3%)	69 (1%)
URIs	522	176 (34%)	374 (72%)	158 (30%)	85 (16%)	188 (36%)	63 (12%)
Total	**5,828**	**3,836 (66%)**	**4,949 (85%)**	**1,632 (28%)**	**747 (13%)**	**360 (6%)**	**132 (2%)**

total number of ICC objects that were detected. The third and fourth columns present the number of ICC values discovered by Epicc and by IC3 that only have precise (e.g., not equal to `.*`) field values. The fifth and sixth columns show the number of imprecise values detected by each tool. Finally, the *missing* columns show the number of locations where an ICC value was missed by either tool.

We observe that the precision of the values inferred by IC3 for Intents, Intent Filters and URIs was high, with 85% of values being detected accurately by our tool. Epicc, on the other hand, could only precisely detect 66%. Of the 915 Intent and Filter values that IC3 detected precisely but Epicc did not, 591 were due to the presence of URI data in Intent values, which is not handled by Epicc. In 4 cases, Epicc missed a value that IC3 did not. The remaining 324 cases that were precisely detected by IC3 and not by Epicc were due to the more powerful string analysis. There was also a clear difference in the case of URIs, with IC3 precisely determining 374 values, compared to 176 for Epicc. That is because Epicc does not include a thorough model for URIs. In particular, a number of methods refer to other modeled objects. Since this is handled in an *ad hoc* manner in Epicc, good coverage of these methods cannot be achieved, resulting in a lot of missed values. On the other hand, using COAL specifications, IC3 achieves much better coverage of URI methods. In particular, references to modeled values are handled in a principled and generic manner. Finally, IC3 detected 73 fewer URI values imprecisely than Epicc, thanks to our new string analysis.

There are several reasons why IC3 missed 132 ICC values. First, some API callback methods have Intent or URI arguments that cannot be known statically. For example, method $onReceive()$ is a Broadcast Receiver callback that is called upon reception of an Intent. The received Intent is passed as an argument to that method by the framework upon activation of the Receiver. The value of that Intent is in general impossible to determine statically. We found 48 such cases. Another related case was when URIs were extracted from Intents that were callback arguments with the $getData()$ method, before being used to address Content Providers. Another cause for missed ICC values was when Intents were extracted from containers such as sets or lists. We will investigate handling these by tracking the values of these containers in future work. We note that handling containers is challenging, especially if tracking array indices is desired. Finally, we found a few pathological cases where a call to an interface or abstract method returning an Intent was not resolved to the proper possible subtypes by the call graph construction procedure.

In the 747 cases where imprecise values were inferred, the arguments to ICC API methods could not be determined. Some cases are not yet handled by our argument analyses (e.g., integer fields and string array fields), while other cases cannot be determined statically (e.g., sequences of complex string operations). We will continue investigating the cases that can be resolved while keeping good performance.

Component matching - As an application of inferring ICC values, we matched the computed Intents with potential target components for the 460 applications. This is a fundamental application of the ICC analysis, since the matching is necessary for any inter-component analysis. Matching precision determines the precision of the overall analysis. Its influence on analysis precision is similar to the influence of the call graph construction process in interprocedural program analyses: an imprecise call graph results in an overall imprecise analysis.

We implemented a matching process that was modeled after the Android Intent resolution process. We performed the matching using both the values computed by IC3 and those calculated by Epicc. Matching Intent-sending program locations with potential target components using values output by IC3 produced 26,872 links. In contrast, the matching that used Epicc values yielded 120,817 links. When performing inter-component analysis, fewer potential links imply fewer false positives (since the ICC value computation and matching are conservative). The 78% reduction in potential targets is a very significant gain in precision. The reason why a 19% gain in ICC value precision resulted in a 78% gain in matching precision is that imprecise ICC values often cause an explosion of the number of potential links. For example, when the *action* of an Intent is not known, the matching process conservatively matches it with all Intent Filter *action* values.

Performance - Processing all the applications took 64,571 seconds using our tool, or slightly less than 18 hours of compute time. That is about 140 seconds per application on average. The processing time was dominated by the IDE problem solver and the string solver, taking 73% of the time overall. The second most time-consuming function was the entry point building procedure of [1], taking 20% of the total time. Soot analyses (class loading, type inference, final call graph construction, etc.) took 4% of the time. Other parts of the analysis (e.g., COAL model parsing, result generation) took 3% of the total time. We did not find any clear trend describing how running time grows with size parameters of the input program. We leave this matter for future work.

VIII. Discussion

Writing COAL specifications requires some effort, which could be seen as a limitation. However, the effort to write a specification is much less than the effort required to produce separate data flow models for each object. We have also found that it is less prone to errors. Finally, handling cases where modeled objects reference other modeled objects in a principled way (for example see Lines 12 and 22 of Figure 2(c)) has allowed us to model complex inter-object relationships such as the one between Android Uri, UriBuilder and Intent. We estimate that writing specifications for all modifiers and queries for Android took us approximately five hours using the developer documentation for the classes involved. On the other hand, writing *ad hoc* composite constant propagation models for Epicc took longer than eight hours for each modeled object, with an incomplete coverage. In order to make writing specifications more effortless, we are looking into a semi-automated inference approach. We believe that COAL elements such as the list of fields, many modifiers and sources as well as queries can be inferred automatically.

We have successfully applied composite constant propagation to Android ICC, but it can also be applied to other problems where object values have to be inferred. In order to ensure that this is the case, the COAL solver can be extended by registering new COAL keywords for field operations and field types. This enables support for additional operations beyond add, remove, replace and clear, as well as for additional method argument analyses.

Our evaluation does not compute the number of spurious values that would be computed if we were not keeping track of the correlations between fields. We expect that, similarly to any analysis refinement, this will be useful in some contexts more than others. In future work, we will quantify the precision benefit and potential performance penalty of keeping track of field correlations over traditional constant propagation. We will also compare our results with values from dynamic traces.

IC3 has the traditional limitations of static analysis on Java. It does not handle native code or reflection. Some approaches [4] exist that can handle reflection for Java programs and could be adapted for Android. Loops and recursion are naturally handled for the operations that we defined (i.e., `add`, `remove`, `clear` and `replace`) because the corresponding field transformers are idempotent for composition. Other operations (e.g., appending to a list) would require carefully defining the composition of the corresponding field transformers.

IX. Related Work

Single-valued interprocedural constant propagation has been studied in the past [6], [16], [27], [36]. Unlike our work, for each constant these works seek to find a single value that is common to all interprocedural paths. Multi-valued constant propagation [2], [26] has also been studied. While our constant propagation is also multi-valued, it propagates composite types. As we explain in Section III, it is possible to simply consider fields to be separate, single variables. However, this approach limits the precision of the results.

We are not the first to consider tuples of values in the context of static analysis. Several works have used tuples or vectors to represents properties of sets of sets of variables [10], [19], [20], keeping track of correlations between properties of different variables. Our analysis is more restricted in that it only handles correlations between object fields. However, our goal is different: we aim to provide analysis designers with a relatively easy-to-use layer of abstraction to statically compute possible object values without having to write data flow functions. This has enabled us to write a thorough model of Android ICC with limited effort. We hope that it will allow other analysis designers to quickly prototype and run composite constant propagation analyses in various contexts.

Analysis of Inter-Component Communication in Android has been performed in past work. Dynamic analysis has attempted to enforce security policies related to ICC [5], [12]. Other work has performed inter-component dynamic taint analysis [13]. Static analysis has also been investigated. ComDroid [8] attempts to determine a limited number of properties of Intents. Epicc [34] is the first work that tried to determine most Intent attributes that are useful for component matching. It performs some *ad hoc* composite constant propagation, which is considerably more complex than writing COAL specifications. Another important difference is how we deal with cases where modeled classes reference other modeled types. Epicc deals with them in an *ad hoc*, class-specific manner. On the other hand, our iterative algorithm described in Section VI-C is completely generic and can apply to all occurrences of modeled value references. As a result, we can model all of ICC in Android. However, like Epicc, our analysis is context-sensitive and flow-sensitive. Apposcopy [15] uses static analysis as the basis of a signature-based malware detection system. The static analysis includes some ICC analysis limited to a subset of the Intent fields. In particular, URI data is not considered.

String analysis reasons about the set of values for string variables. While much work has been performed in this area [9], [17], [21], [28], [37], [40], JSA [9] is the closest to our analysis. However, JSA seeks to model all string operations, whereas we limit our analysis to the most common cases. Additionally, while JSA performs its own pointer analysis, we rely on the more efficient Spark [23] analysis, which is already performed as part of the ICFG building process. As a result, our analysis is much more efficient in the context of Android ICC analysis. Initial tests with JSA showed processing times well over an hour for medium sized applications, which made the entire ICC analysis impractical.

X. Conclusion

In this paper, we introduced the MVC constant propagation problem, and we presented the COAL language and the associated solver for MVC problems. We also developed IC3, an Android ICC analysis tool that is based on a reduction to an MVC problem. As a part of IC3, we developed a sound string analysis that offers an effective tradeoff of scalability and precision. We achieved a much greater accuracy in ICC inference than previous work. In the future we plan to investigate more ways to improve accuracy, and to what extent generating COAL specifications can be automated. Finally, we will apply our IC3 work to design novel inter-component analyses in Android.

Acknowledgements

We thank William Harris for editorial comments during the writing of this paper. This material is based upon work supported by National Science Foundation Grants No. CNS-1064900, CNS-1228700, CNS-1228620 and CNS-1219495. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research was also supported by a Google Faculty Research Award.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick 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*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[2] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer Berlin Heidelberg, 2004.

[3] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2012)*, pages 3–8, July 2012.

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[5] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual NDSS Symposium*, February 2012.

[6] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 152–161, New York, NY, USA, 1986. ACM.

[7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[8] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[9] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, New York, NY, USA, 1977. ACM.

[11] Xingmin Cui, Da Yu, Patrick Chan, LucasC.K. Hui, S.M. Yiu, and Sihan Qing. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In Willy Susilo and Yi Mu, editors, *Information Security and Privacy*, volume 8544 of *Lecture Notes in Computer Science*, pages 446–453. Springer International Publishing, 2014.

[12] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[14] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[15] Yu Feng, Saswat Anand, Isil Dillig, and Alex 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*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[16] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 90–99, New York, NY, USA, 1993. ACM.

[17] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 188–198, New York, NY, USA, 2009. ACM.

[18] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[19] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to dijkstra. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 185–190, 1980.

[20] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74, New York, NY, USA, 1982. ACM.

[21] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[22] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[23] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In Grel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin Heidelberg, 2003.

[24] Li Li, Alexandre Bartel, Tegawendé Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, May 2015.

[25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[26] E. Merlo, J.F. Girard, L. Hendren, and R. De Mori. Multi-valued constant propagation for the reengineering of user interfaces. In *Proceedings of the 1993 Conference on Software Maintenance*, pages 120–129, Sep 1993.

[27] Robert Metzger and Sean Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages Systems*, 2(1-4):213–232, March 1993.

[28] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.

[29] Trustlook News. Emergency: Android in-app billing verification bypass vulnerability, November 2013. Available from http://blog.trustlook.com/index.php/emergency-android-app-billing-verification-bypass-vulnerability/.

[30] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, New York, NY, USA, 2012. ACM.

[31] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Android inter-component communication analysis with the coal constant propagation language. Technical Report NAS-TR-0170-2014, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, 2014.

[32] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. Coal constant propagation language. http://siis.cse.psu.edu/coal/, 2014.

[33] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. IC3: Android inter-component communication analysis. http://siis.cse.psu.edu/ic3/, 2014.

[34] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves 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, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[35] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(12):206 – 263, 2005. Special Issue on the Static Analysis Symposium 2003 SAS '03 10th International Static Analysis Symposium.

[36] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, October 1996.

[37] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[38] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 515–526, New York, NY, USA, 2014. ACM.

[39] Raja Vallee-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000.

[40] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[41] Fengguo Wei, Sankardas Roy, Xinming 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*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[42] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS '14)*, 2014.