Composite Constant Propagation and its Application to Android Program Analysis

Damien Octeau, *Member, IEEE*, Daniel Luchaup, Somesh Jha, and Patrick McDaniel, *Fellow, IEEE*

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 489 applications from the Google Play store. The ICC values we infer are substantially more precise than previous work. The analysis is efficient, taking 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.

Index Terms—Composite constant, constant propagation, inter-component communication, ICC, Android application analysis

1 Introduction

Program analyses sometimes need to statically infer the possible values of object fields. Such a program analysis that has recently received interest [11], [17], [35] 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 Android 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 [16], [27]. Second, ICC messages can be intercepted by malicious recipients, with consequences ranging from data leaks [8] to piracy [31]. 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 [24], [26], [40], [43], patch generation for privilege escalation vulnerabilities [44] and detection of stealthy behavior [20].

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 [35] 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 [17] 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.

- *D. Octeau is with the Department of Computer Sciences, University of Wisconsin, Madison, WI, and with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA. E-mail: octeau@cse.psu.edu.*
- *D. Luchaup is with the CyLab, Carnegie Mellon University, Pittsburgh, PA. E-mail: luchaup@andrew.cmu.edu.*
- *S. Jha is with the Department of Computer Sciences, University of Wisconsin, Madison, WI. E-mail: jha@cs.wisc.edu.*
- *P. McDaniel is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA. E-mail: mcdaniel@cse.psu.edu.*

Manuscript received 6 Aug. 2015; revised 1 Feb. 2016; accepted 6 Mar. 2016. Date of publication 4 Apr. 2016; date of current version 18 Nov. 2016. Recommended for acceptance by C. Zhang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2550446

While MVC constant propagation was motivated by Android ICC analysis [34], 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 489 applications from the official Play store. Our analysis found 14,537 possible ICC values, whereas an analysis not taking field correlations into account would have found 4,807,771. This is a significant reduction in the number of inferred unfeasible values. We precisely inferred all fields of ICC values in 84 percent of cases. Epicc, on the other hand, could only infer 68 percent precisely. The remaining 16 percent of values could not be determined because of constructs not yet handled by our string analysis and some pathological cases. Computing ICC values was efficient, taking on average 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 between message-sending code locations and potential recipients is desirable. In our experiments with 489 applications, such a matching yielded 192,662 links with ICC values computed by Epicc, whereas values computed with IC3 produced only 42,238 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: http://siis.cse.psu.edu/coal/
- 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: http://siis.cse.psu.edu/ic3/

2 A Motivating Example: Android ICC

Android applications are composed of four different types of components. *Activities* represent the user interface. *Services* provide background processing. *Content Providers* enable 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

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

be specified by explicitly naming it, or it can be determined automatically by the Android 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.

Fig. 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. Fig. 1a 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.

Fig. 1b 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 Fig. 1a. Therefore, `MapRenderingActivity` could receive the Intent created in Fig. 1a.

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 Fig. 1a, 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 [35] 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 Fig. 1a with the component from Fig. 1b 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 Fig. 1a has to conservatively assume that the data field can take any value. Fig. 1c 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 its inability to infer Intent URI data, the current state-of-the-art [35] would conservatively assume 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 article. In Sections 3 through 6, we introduce a novel approach to statically infer the set of values that objects can take. In Section 7, we apply this approach to the problem of inferring Android ICC values.

3 Overview

3.1 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 $\texttt{OBJ.X} \in \{1, 2\}$ and $\texttt{OBJ.Y} \in \{10, 20\}$. 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* 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. Fig. 2a 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 Fig. 2a. Fig. 2b 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 summarization language we designed for this purpose. *The function of COAL (COnstant propAgation Language) is to specify Multi-Valued Composite constant propagation problems*. It describes three elements to specify the problem:

- The types of the variables for which we are trying to infer possible runtime values.
- How these variables are modified by methods.
- The program locations where the potential values of the variables should be inferred.

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.

Fig. 2c 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. The specification only describes classes of interest, for instance the Intent and Uri classes in our motivating example. It is composed of field declarations, modifiers and a query. The field declarations specify the fields that are being tracked and their type. 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.

```
1  public class Intent {
2    private String action;
3    private Set<String> categories = new
         HashSet<>();
4    private String data;
5    private String mimeType;
6
7    public void setAction(String act) {
8      this.action = act; }
9    public void addCategory(String cat) {
10     this.category = cat; }
11   public void setDataAndType(String d,
                                String t) {
12     this.data = d;
13     this.mimeType = t; }
14   public void setData(Uri u) {
15     this.data = u.getData();
16     this.mimeType = null; } }
17
18 public class Uri {
19   private String data;
20
21   public void setData(String d) {
22     this.data = d; }
23   public void getData() {
24     return this.data; } }
```

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

```
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://a/b/c");
8      intent.setData(uri);
9    } else {
10     uri.setData("file:///foo.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/*`.

```
1  class Intent {
2   String action; Set<String> categories; String
        data; String mimeType;
3
4   mod <Intent: void setAction(String)> {
5    0: replace action; }
6   mod <Intent: void addCategory(String)> {
7    0: add categories; }
8   mod <Intent: void setDataAndType(String,String)> {
9    0: replace data;
10   1: replace mimeType; }
11  mod <Intent: void setData(Uri)> {
12   0: replace data, type Uri:data;
13   clear mimeType; }
14  query <Context: void startActivity(Intent)> {
15   0: type Intent; } }
16
17 class Uri {
18   String data;
19
20   mod <Uri: void setData(String)> {
21     0: replace data; }
22   source <Uri: String getData(String)> {
23     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`}	null	null
data	`http://a/b/c`	`file:///foo.jpg`	`file:///foo.jpg`
mimeType	null	`image/jpg`	`image/*`

(d) Possible values of the fields of $intent$ at the $startActivity()$ call. 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.

The query statement indicates that we would like to infer the values of Intent arguments of 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.

Fig. 2d 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 Fig. 2b. 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.

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

3.2 MVC Constant Propagation Analysis

Fig. 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], [33] 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. Fig. 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

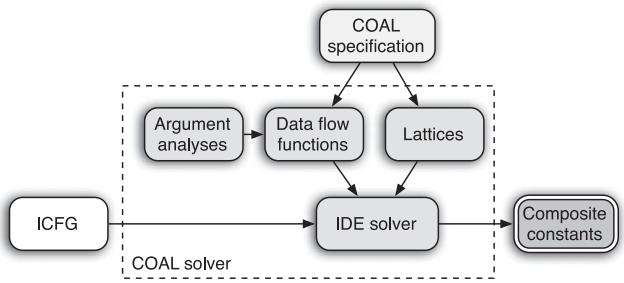

Fig. 4. The MVC data flow analysis process (Step 3 from Fig. 3).

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. Our analysis is flow-sensitive, context-sensitive, and it takes aliasing into account through the process described in Section 7.

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 [38]. We present the generic IDE model for constant propagation in Section 5. 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. In particular, in Section 7.1 we present a string analysis that is finely tuned for the purposes of Android. It efficiently models idiomatic constructs such as string concatenation and flows through fields.

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).

4 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 Fig. 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 Fig. 5, we use non-terminals $\langle type \rangle$ and $\langle field\ name \rangle$ to represent valid types and field names.

Modifiers. Modifiers represent method calls where values flow to the modeled object. The specification of the modifiers comprises a method signature $\langle method\ sig \rangle$ that identifies the

⟨*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.

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). The `add` and `remove` operations only apply to fields with container types and are undefined for primitive types such as strings or integers. A precise definition of these operations is presented in Section 5.2. 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 objects to be modeled in COAL. Where these values are used, the COAL solver uses the specified value.

Sources. Sources model the case where a field value flows out of an object specified in COAL. This is useful if the field value subsequently flows to another object modeled in COAL. For example, in Line 11 of Fig. 2b, the *data* field of a Uri object flows to the *data* field of an Intent object. The COAL source declaration at Line 22 of Fig. 2c enables the COAL solver to use the value of the Uri field to determine the value of the Intent variable.

Queries. Queries specify statements of interest where modeled values should be determined so that they are used by a client analysis.

The MVC constant propagation problem from Fig. 2b can be solved by inputting the program and the specification from Fig. 2c 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. Finally, while we designed the COAL language to be easy to use, other alternatives are possible for expressing data flows (e.g., XML schema). It is possible to add front-ends to the COAL solver that support these alternative designs.

5 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 problem, using the model defined in this section. Given an IDE problem, existing algorithms can compute a solution [38]. 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 6 to automatically build reductions to IDE problems.

5.1 The L Semilattice of Values

For any set X, we denote the power set of X (i.e., all subsets of X) by 2^X and the set of functions from X to X by X^X. In this section, we are trying to determine the value of an object with n fields, taking values in finite sets $V_1, \ldots, V_n$. For $i \in \{1, \ldots, n\}$, let $P_i = V_i \cup \{\omega\}$, where ω represents an undefined value. Let $B = P_1 \times \cdots \times P_n$. We define $L = (2^B, \cup)$ a join-semilattice with a bottom element $\perp = \varnothing$. The join operation on L is the set union $\cup$. The top element of L is the set of all elements in B. It should be noted that lattice L is specific to the modeled object and we use a single lattice for each object type.

The 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.

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

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

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

In the first branch of the `if` statement, the value associated with $intent$ is transformed to

$$\{(\texttt{VIEW}, \{\texttt{BROWSABLE}\}, \texttt{http://a/b/c}, null)\}. \tag{2}$$

We have used the fact that the *data* field of uri contains http://a/b/c. In the fall-through branch of the `if` statement, this value becomes

$$\begin{aligned}\{&(\texttt{VIEW}, \varnothing, \texttt{file:///foo.jpg}, \texttt{image/jpg}),\\ &(\texttt{VIEW}, \varnothing, \texttt{file:///foo.jpg}, \texttt{image/}*)\}.\end{aligned} \tag{3}$$

When the two branches merge, at Line 12, the value becomes

$$\begin{aligned}\{&(\texttt{VIEW}, \{\texttt{BROWSABLE}\}, \texttt{http://a/b/c}, null),\\ &(\texttt{VIEW}, \varnothing, \texttt{file:///foo.jpg}, \texttt{image/jpg}),\\ &(\texttt{VIEW}, \varnothing, \texttt{file:///foo.jpg}, \texttt{image/}*)\},\end{aligned} \tag{4}$$

which is the set union of (2) and (3). Note that we have used the fact that in Fig. 2b, the $mimeType$ argument may have value either `image/jpg` or `image/*`.

5.2 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 6.4.

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 $\phi = \phi_{GEN}^{KILL}$.
- Type (2): For all $X \in P_i$, $\phi(X) = GEN$, for some GEN in P_i. This case is also denoted by $\phi = \phi_{GEN}$.

It is easy to verify that the set of such field transformers is closed under composition. Using these notations, we can precisely define the operations introduced in Section 4:

- For any `add` operation, there exists a set $GEN \in P_i$ such that the `add` operation is modeled by $\phi_{GEN}^{\varnothing}$.
- For any `remove` operation, there exists a set $KILL \in P_i$ such that the `remove` operation is modeled by $\phi_{\varnothing}^{KILL}$.
- For any `replace` operation, there exists a set $GEN \in P_i$ such that the `replace` operation is modeled by ϕ_{GEN}.
- The `clear` operation is modeled by $\phi_{\varnothing}$ for sets and by ϕ_{null} for scalars.

Let us denote the identity field transformer by id. 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 Fig. 2b. Using the

definition above and the fact that this method replaces the existing action value, we can model it using type (2) field transformer ϕ_{VIEW}.

Field transformers are used as basic building blocks for data flow functions. We define $\mathcal{L}$, the set 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 Fig. 2b on field *action* is modeled by field transformer ϕ_{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.

The 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 $\mathcal{L}$:

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

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, in order to model m executions paths, 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, which only occurs 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 Fig. 2b. The influence of the first branch can be summarized by function $\{id \times \phi^{\varnothing}_{\{\texttt{BROWSABLE}\}} \times \phi_{\texttt{http://a/b/c}} \times \phi_{null}\}$, where ϕ_{null} clears the value of the *mimeType* field. The second branch is modeled by function

$$\begin{aligned}\{&id \times id \times \phi_{\texttt{file:///foo.jpg}} \times \phi_{\texttt{image/jpg}},\\ &id \times id \times \phi_{\texttt{file:///foo.jpg}} \times \phi_{\texttt{image/*}}\}.\end{aligned} \tag{5}$$

Finally, the influence of the two branches is summarized by

$$\begin{aligned}\{&id \times \phi^{\varnothing}_{\{\texttt{BROWSABLE}\}} \times \phi_{\texttt{http://a/b/c}} \times \phi_{null},\\ &id \times id \times \phi_{\texttt{file:///foo.jpg}} \times \phi_{\texttt{image/jpg}},\\ &id \times id \times \phi_{\texttt{file:///foo.jpg}} \times \phi_{\texttt{image/*}}\}.\end{aligned} \tag{6}$$

We can verify that applying this function to the value given by Equation (1) yields the value given by Equation (4).

We define the composition of two functions $f_1 = \{l_1^1, \ldots, l_m^1\}$ and $f_2 = \{l_1^2, \ldots, l_k^2\}$ in F to be the pairwise composition of all l_a^1 with all l_b^2, for $1 \leq a \leq m$ and $1 \leq b \leq k$:

$$f_1 \circ f_2 = \{l_1^1 \circ l_1^2, \ldots, l_1^1 \circ l_k^2, \ldots, l_m^1 \circ l_1^2, \ldots, l_m^1 \circ l_k^2\}.$$

In the Appendix in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2016.2550446, we prove the following proposition by defining the composition of other functions in F (e.g., Ω).

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 $f_2 = \{l_1^2, \ldots, l_k^2\}$, $f_1 \cup f_2 = \{l_1^1, \ldots, l_m^1, l_1^2, \ldots, l_k^2\}$.

6 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 [38]. We first outline the requirements of IDE problems.

6.1 Environment Transformers

Let D be the set that comprises all variables of the type modeled with L in the program and a special Λ symbol, which represents the absence of a data flow fact. An *environment* is a function from D to L, where L was introduced in Section 5.1. 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 distributive if, for every $e_1, e_2, \ldots \in E$ and for any $d \in D$, $(t(\sqcup_i e_i))(d) = \cup_i (t(e_i))(d)$.

6.2 The Pointwise Representation of Environment Transformers

It can be shown that any environment transformer t can be written in terms of a *pointwise representation* $\mathcal{R}_t$,[1] which is a function from $(D \cup \{\Lambda\}) \times (D \cup \{\Lambda\})$ to L^L. The pointwise representation is useful because it allows for easy specification of transformers. The pointwise representation answers the following question: given two symbols d' and d, how does the value associated with d' contribute to the value of d? More specifically, for any environment transformer t, for all $e \in Env(D, L)$ and $d \in D$, we have

$$t(e)(d) = \mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_{d' \in D} \mathcal{R}_t(d', d)(e(d'))). \tag{7}$$

In Section 5.2, we have used the pointwise representation, and more specifically we have defined functions in L^L that model the composite constant propagation problem. Please refer to Section 5.2 for examples of pointwise representations of transformers.

We now state a result that links the distributivity of the functions in L^L to the distributivity of environment transformers.

1. The exact expression of $\mathcal{R}_t$ is not useful for this section. Interested readers are referred to [38].

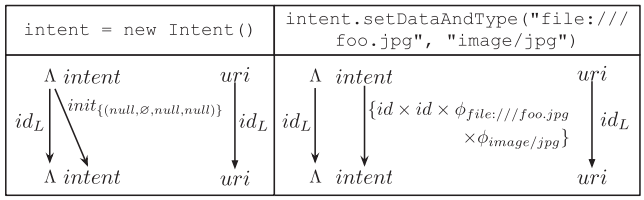

Fig. 6. Transformers for statements from Fig. 2b.

Definition 2. *We say that a function* $\mathcal{R}_t : (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \rightarrow L^L$ *is* codistributive *if all elements of its range are distributive functions from L to L.*

Proposition 2. *If* $\mathcal{R}_t$*:* $(D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \rightarrow L^L$ *is codistributive, then t defined as in Equation (7) is a distributive environment transformer.*

Proof. Let $e_1, e_2, \ldots \in Env(D, L)$ and $\mathcal{R}_t : (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \rightarrow L^L$ codistributive. Using Equation (7):

$$t(\sqcup_i e_i)(d) = \mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_{d' \in D} \mathcal{R}_t(d', d)((\sqcup_i e_i)(d')))$$
$$= \mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_{d' \in D} \mathcal{R}_t(d', d)(\cup_i (e_i(d'))))$$

by definition of $(\sqcup_i e_i)(d')$. Since $\mathcal{R}_t(d', d)$ is distributive [38], we have:

$$t(\sqcup_i e_i)(d) = \mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_{d' \in D} (\cup_i \mathcal{R}_t(d', d)(e_i(d'))))$$

Using the commutativity of the $\cup$ operator, we get:

$$t(\sqcup_i e_i)(d) = \mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_i (\cup_{d' \in D} \mathcal{R}_t(d', d)(e_i(d'))))$$
$$= \cup_i (\mathcal{R}_t(\Lambda, d)(\bot) \cup (\cup_{d' \in D} \mathcal{R}_t(d', d)(e_i(d'))))$$
$$= \cup_i t(e_i)(d)$$

by using the idempotence and the commutativity of $\cup$ and Equation (7). □

We define environment transformers by their pointwise representation $\mathcal{R}_t$ using functions in F. Examples of environment transformers with their representation are shown in Fig. 6. For example, for statement `intent = new Intent()`, the representation $\mathcal{R}_t$ for the corresponding transformer is defined as:

$$\mathcal{R}_t(d', d) = \begin{cases} id_L & \text{if } (d', d) = (\Lambda, \Lambda) \\ & \text{or } (d', d) = (src, src) \\ init_{\{(null, \varnothing, null, null)\}} & \text{if } (d', d) = (intent, \Lambda) \\ \Omega & \text{otherwise.} \end{cases}$$

This function describes the relationships between symbols before the statement (d) with symbols after the statement (d'). The first case (id_L) means that we are propagating the values of Λ (the empty data flow fact) and src without any changes. The second case means that we are creating a new data flow fact $intent$, as indicated by the edge between Λ and $intent$. We are associating function $init_{\{(null, \varnothing, null, null)\}}$ with that edge. Since the value associated with Λ is $\bot$, this informally means that the contribution of Λ to the final value of $intent$ is $init_{\{(null, \varnothing, null, null)\}}(\bot) = \{(null, \varnothing, null, null)\}$ (see Equation (7)). The final case (Ω) means that there exists no relationship between any other symbol.

Transformers are defined that way for all statements of interest in the program.

Proposition 3. *All elements of* F *are distributive functions.*

The proof of this proposition is trivial, given the definition of the functions in F. Since all elements in F are distributive, according to Proposition 2, the resulting environment transformers are distributive. It follows that the data flow problem can be solved using existing algorithms from [38].

6.3 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 Fig. 2c) and (iv) $field$, the index (or the name) of the modified field.

Algorithm 1. Generate Functions in F from COAL Modifiers

```
1:  procedure GenerateFunction(modifier, statement)
2:    result := id_L
3:    for all arguments arg in modifier.args do
4:      values := null
5:      if arg.number != null then
6:        values := GetArgumentValues(statement,
          arg.number, arg.type)
7:      arg_function := null
8:      if values ≠ null then
9:        for all argument values value in values do
10:         current := BuildFuncInF(arg.op, value, arg.field)
11:         if arg_function = null then
12:           arg_function = current
13:         else
14:           arg_function = arg_function ∪ current
15:     else
16:       arg_function := BuildFuncInF(arg.op, null, arg.field)
17:     result := result ∘ arg_function
18:   return result
```

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 7.1. 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 $f_1 \cup \ldots \cup f_n$ (Line 14).

To illustrate this procedure, let us consider Line 11 of Fig. 2b. The COAL solver determines that this is a modifier with two arguments (see Fig. 2c 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:///foo.jpg`. Since a `replace` operation is requested, the BuildFuncInF procedure generates field transformer $\phi_{\texttt{file:///foo.jpg}}$. Using the fact that *data* is the third field (Line 2 of Fig. 2c), it generates function

$$\{id \times id \times \phi_{\texttt{file:///foo.jpg}} \times id\}. \quad (8)$$

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

$$\{id \times id \times id \times \phi_{\texttt{image/jpg}}, id \times id \times id \times \phi_{\texttt{image/*}}\}, \quad (9)$$

where Line 14 utilizes the definition of the $\cup$ operator on F from Section 5. Finally, Line 17 of Algorithm 1 composes the two functions given by Equations (8) and (9), which yields the function given by Equation (5).

6.4 Fixed Point Iteration

Let us consider method $sendMessage()$ from Fig. 2b. So far, we have assumed that the value of the Uri uri at Line 8 of Fig. 2c 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 Fig. 2b by utilizing the value of uri computed in the previous iteration.

7 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 Fig. 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 [41] and the Heros IDE solver [3]. Soot converts Java bytecode to an internal IR that is recognized by its Spark [25] pointer analysis and 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 [33], 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 in a way similar to the standard idea of weak updates [7]. 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).

7.1 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 string analysis is flow-sensitive and interprocedural. It 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.

Constraint generation. In the first stage we generate constraints for all string operations. 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. Our analysis relies on the flow-sensitive use-def analysis provided by the Soot framework. We tried to use the Single Static Assignment (SSA) intermediate program representation, where local variables are defined exactly once [12], but Soot's SSA conversion was not robust enough to handle the code translated from Dalvik. Nevertheless, for simplicity in what follows we present the analysis as if SSA form was

```
1  String bar(String in)
2  {
3   if(cond1)
4     return in;
5   String s1 = "11";
6   String s2 = s1 + "22";
7   if (cond2)
8    s1 = foo(s1, s2);
9   else
10   s1 = s2;
11
12  return s1;
13 }
```

(a) Java code

```
1  bar() {
2   r0 := parameter0
3   if (cond1)
4     return r0;
5   r1 = "11";
6   r2 = r1.append("22");
7   if (cond)
8     r3 = foo(r1,r2);
9   else
10    r4 = r2;
11  r5 = φ(r3,r4);
12  return r5;
13 }
```

(b) Simplified SSA code

```
1  //bar
2  r0 = arg[bar,0];
3
4  ret[bar] ⊇ r0;
5  r1 = {"11"};
6  r2 = cat(r1,"22");
7
8  r3 = call(foo,r1,r2);arg[foo,0]⊇ r1;arg[foo,1]⊇ r2;
9
10 r4 = r2;
11 r5 = union(r3,r4);
12 ret[bar] ⊇ r5;
13 ..
```

(c) The process of adding constraints

Fig. 7. Running example.

used, adding clarifications when the distinction is relevant. We introduce the following symbolic values:

1) For each SSA variable r we introduce a symbol $\overline{r}$ which represents r's set of possible values.
2) For each function `foo`, **ret**`[foo]` represents the set of values returned by `foo`.
3) For each function `foo`, **arg**`[foo,` n`]` represents the set of values for the nth argument of `foo`.
4) For each class `C` and field `f`, $\overline{C.f}$ represents the set of values for field `f` in objects of type `C`.

We perform a whole-program analysis by simply traversing all instructions in all reachable functions and gathering the corresponding constraints. For instance, an assignment to an SSA variable $x = y$ generates the constraint $\overline{x} = \overline{y}$. If the assignment is $x = foo(y, z)$ we generate three constraints: one for the assignment $\overline{x} = \textbf{call}(foo, \overline{y}, \overline{z})$, and one for each argument **arg**`[foo,1]` $\supseteq \overline{y}$, and **arg**`[foo,2]` $\supseteq \overline{z}$. A `phi` statement $x = \phi(y, z)$ generates a union constraint $\overline{x} = \textbf{union}\ (\overline{y}, \overline{z})$. Here **union** $(\overline{y}, \overline{z})$ is the set union, $\overline{y} \cup \overline{z}$. In reality, without an SSA form, for each variable x and each location l where x is assigned we introduce a symbol $\overline{x_l}$, which represents the set of values of x at l. Additionally, consider a location l and a variable x which is used at location l, such that x has multiple reaching definitions from locations $l1, l2, \ldots$. The set of values x may have when used at location l, is represented by **union** $(\overline{x_{l1}}, \overline{x_{l2}}, \ldots)$.

A `return x;` statement inside function `bar` generates the constraint **ret**`[bar]` $\supseteq \overline{x}$, and if `bar` has one more return statement `return y;` we group all the constraints for the `bar`'s return statements into a single one, **ret**`[bar]` $=$ **union** $(\overline{y}, \overline{z})$. Similarly, by the end of the whole-program analysis, we group all the constraints for arguments and fields using **union** constraints. For instance, the first argument of `foo` becomes constrained by **arg**`[foo,1]` $=$ **union** $(\overline{y}, \ldots)$. Symbolic values that represent fields, or function arguments and results, such as **arg**`[foo,1]` **ret**`[bar]`, represent conservative, context insensitive solutions for the corresponding sets of values. However, we use the constraint graph in a context sensitive analysis which propagates more precise values for function arguments and results, and uses the context insensitive information only when widening is required.

As an example, consider the program in Fig. 7a, and its simplified SSA representation in Fig. 7b. We traverse the instructions in Fig. 7b, and for each one we generate the corresponding constraints. Fig. 7c shows how the constraints are added, step by step. When we finish processing `bar`, we group the two inclusion constrains on the return value with a single union constraint **ret**`[bar]` $=$ **union** $(\overline{r0}, \overline{r5})$. Note that at this point we can not do the same replacement for the arguments of `foo`, because `foo` may be called in other functions. At the end of the whole-program analysis we eliminate all inclusion constraints with the same left operand and replace them by a union constraint. For instance, we would replace **arg**`[foo,1]` $\supseteq \overline{y_1}, \ldots,$ **arg**`[foo,1]` $\supseteq \overline{y_n}$, with **arg**`[foo,1]` $=$ **union** $\{\overline{y_1}, \ldots, \overline{y_n}\}$.

Certain calls result in string concatenation, and because of its importance we consider concatenation as an operator allowed in constraints. The drawback is that we have to model those library calls that can yield string concatenation. However, in Android applications, it suffices to model just the *String* and *StringBuilder* classes to cover the majority of string operations. In such cases, a high level Java code $x = y + z$ would generate the intermediate code $x = y.append(z)$, for which we insert the constraint $\overline{x} = \textbf{cat}(\overline{y}, \overline{z})$. The expression **cat**$(\overline{y}, \overline{z})$ represents the set of elements of the form $w_y w_z$ with $w_y \in \overline{y}$, and $w_z \in \overline{z}$. If we are unable to model the right hand of an assignment $x = \ldots$, then we generate $\overline{x} = \bot$. The meaning of $\bot$ is *all strings*, i.e $.*$ in terms of a regular expression.

Fig. 9 shows the constraint language. The values can describe real variables (such as $\overline{r1}$ or $\overline{r2}$ in Fig. 7) or an abstract class of values (such as **ret**`[bar]` which represents the set of values that `bar` could return). Note that our analysis is field sensitive, but not object sensitive, a restriction that can be removed in the future.

We stay close to the semantics of the original program, and do not commit to a particular class of languages (such as context free, or regular) to allow more flexibility in picking an appropriate solver.

When modeling string operations we are faced with the following alias problem. Consider the example in Fig. 8a, where $s1$ and $s2$ are two variables of type *StringBuilder*. The statement `s2=s1.append("123")` makes $s1$ and $s2$ aliases, therefore modifications of one of them also modify the other. However this is not captured by the def-use in Soot. As a workaround to this problem, we replace $s2$ with $s1$ in the left hand side of the assignment, and we replace the uses of $s2$ from the original definition, with $s1$, as shown in Fig. 8b. We apply the workaround at all function calls known to introduce aliases (Fig. 8c). In one corner case, we cannot apply our changes without additional care. This happens, for instance, if variable $s2$ from Fig. 8a is also defined on another branch and if both definitions reach a common

```
1 s1 = new StringBuilder("")
2 s2 = s1.append("abc")
3 s3 = s2.append("def")
4 use(s3)
```

(a) Aliasing problem: after the first append (line 2) s1 and s2 become aliases; and after the second append (line 3) s2 and s3 become aliases.

```
1 s1 = new StringBuilder("")
2 s1 = s1.append("abc")
3 s3 = s1.append("def")
4 use(s3)
```

(b) Handling the first append: we replace s2 with s1 in the result at line 2, and in all uses of this result. Note that the append at line 3 now uses s1.

```
1 s1 = new StringBuilder("")
2 s1 = s1.append("abc")
3 s1 = s1.append("def")
4 use(s1)
```

(c) Handling the second append (line 3): we replace s3 with s1 in the result at line 3, and in all uses of this result. Note that line 4 now uses s1.

Fig. 8. Alias workaround example.

use of $s2$. This is another case where the SSA representation would enable an easier completion of our workaround. Our changes would break SSA's unique definition requirement (such as $s1$ being defined twice in Fig. 8b), but this would not be a problem since we could restore valid SSA. However, because we lack SSA, and because the code structure that triggers this corner case is seldom found in typical Android applications, we merely detect this corner case and issue a warning. In our experiments, only one application triggered this warning.

The set of constraints induce a graph on the set of values. For instance, in Fig. 7c the constraint $\overline{r5} = \mathbf{union}(\overline{r3}, \overline{r4})$ represents a *dependency* between $\overline{r5}$ and $\overline{r3}$, and also between $\overline{r5}$ and $\overline{r4}$. These dependencies form a *flow graph* and the nodes with outgoing edges can be viewed as nonterminals in a context-free grammar augmented with operation productions (see [9]).

Constraint solving. 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 over-approximates the set of values that $\overline{x}$ can take. It works by finding the constraint associated with $\overline{x}$ and by traversing the flow graph and interpreting the nodes.

Observe that if the flow graph rooted at a given node N has no cycles, and if there are no *call* operations, then the flow graph under that node actually describes a finite language. In the general case, the flow graph rooted in a given node N can be viewed as a program that computes the set associated with node N. We evaluate this program, and avoid non-termination by detecting cycles using a stack of nodes: when we are about to evaluate a node that is already on the stack, perform widening and consider its value to be `.*` (that is, $\bot$). 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. Assume that the function *foo* used by $r3 = foo(r1, r2)$ in Fig. 7 is:

```
1 String foo(String p1, String p2) {
2   String end = "";
3   for(int i=0; i<10; ++i)
4     end = "!"+end;
5   return p1+p2+end;
6 }
```

Our analysis is able to find a solution $\overline{r3} = \{111122|111122!.*\}$, even if *foo* is called in many other contexts, while the method in [9] loses precision and does not obtain the 111122 prefix.

⟨*constraint*⟩ ::= ⟨*value*⟩ ⟨*op*⟩ ⟨*expr*⟩
⟨*op*⟩ ::= = | ⊇
⟨*value*⟩ ::= ⟨*var*⟩ | ⟨*result*⟩ | ⟨*argument*⟩ | ⟨*field*⟩
⟨*var*⟩ ::= ⟨*identifier*⟩
⟨*result*⟩ ::= 'ret' '['⟨*func*⟩']'
⟨*argument*⟩ ::= 'arg' '['⟨*func*⟩, ⟨*integer*⟩']'
⟨*field*⟩ ::= ⟨*identifier*⟩
⟨*expr*⟩ ::= ⊤ | ⊥ | NULL | ⟨*value*⟩ | '{'STRING'}'
⟨*expr*⟩ ::= 'union' '('⟨*value*⟩, ⟨*value*⟩, ...')'
⟨*expr*⟩ ::= 'cat' '('⟨*value*⟩, ⟨*value*⟩')'
⟨*expr*⟩ ::= 'call' '('⟨*func*⟩, ⟨*value*⟩, ⟨*value*⟩, ...')'
⟨*expr*⟩ ::= ⟨*argument*⟩
⟨*func*⟩ ::= ⟨*identifier*⟩

Fig. 9. Constraint language.

7.2 Evaluation

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

Q1: Does the composite propagation lead to fewer values than considering fields to be separate variables?
Q2: Can IC3 precisely infer field values of ICC objects?
Q3: As an application of our analysis, how precisely can ICC messages be matched with their targets?
Q4: 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:

- For each code location that may send more than one ICC value, the composite constant propagation finds on average 19 percent fewer values than the traditional constant propagation. Overall, composite constant propagation reduces the number of values found by 99.7 percent by avoiding the combinatorial explosion that sometimes occurs when fields are considered as separate variables.
- IC3 infers precise field values for 84 percent of ICC values in a corpus of 489 Android applications. Epicc can only infer 68 percent. This is a significant increase in precision.
- When matching components that may communicate with one another, specifications from IC3 lead to 78 percent fewer links between message-sending locations and potential recipients than the current state-of-the-art. This implies a significant decrease in the number of unfeasible inter-component paths in client analyses.

- On average, our analysis takes two minutes 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 (data set described in [13]). Some applications could not be processed because of errors caused by insufficient memory or timeout, so we report numbers for 489 applications.

Size of value sets. Recall that the COAL solver takes into account field correlations to avoid considering object values with unfeasible field combinations. In order to measure the number of unfeasible values avoided by the composite constant propagation we compared the number of ICC values computed by our approach at message-passing code locations with the number of values that would be inferred if field correlations had not been taken into account. We considered all 7,103 message-sending locations. Overall, the composite constant propagation found 14,537 possible ICC values whereas traditional constant propagation would have found 4,807,771 values. This constitutes a 99.7 percent decrease in the total number of potential values.

Looking more closely at the distribution of the values, we found that in 5,766 of the message-passing locations, there was only a single possible ICC values. This implies that the sets of values are identical whether the field correlations are considered or not. In the remaining 1,193 cases, we observed that on average the composite constant propagation led to a reduction of over 19 percent of the number of potential values. For each message-sending location, we denote by N_c the number of ICC values inferred by composite constant propagation and by N_s the number of values that would be inferred by considering the fields to be separate. Fig. 10 shows the ratio $\frac{N_c}{N_s}$ for all 1,193 locations that send more than one potential ICC value. We notice that the reduction in unfeasible values is highly variable across code locations. In particular, for 87 values the number of potential values was reduced by over 90 percent, avoiding the combinatorial explosion that occurred when fields were considered as separate variables. This indicates that composite constant propagation can effectively reduce the number of unfeasible values in cases where multiple values may occur, thereby improving overall analysis precision.

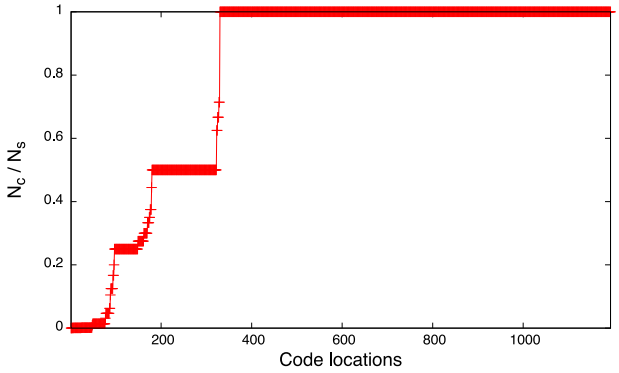

Fig. 10. Ratio $\frac{N_c}{N_s}$ for message-sending locations with multiple ICC values.

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). We counted the number of ICC values inferred by IC3 and Epicc [35] 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 1. 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 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 84 percent of values being detected accurately by our tool. Epicc, on the other hand, could only precisely detect 66 percent. Of the 1,129 Intent and Filter values that IC3 detected precisely but Epicc did not, 656 were due to the presence of URI data in Intent values, which is not handled by Epicc. In 23 cases, Epicc missed a value that IC3 did not. The remaining 269 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 430 values, compared to 249 for

TABLE 1
ICC Value Field Precision Results

		ICC values with precise fields		ICC values with imprecise fields		Missing ICC values	
	Value count	Epicc	IC3	Epicc	IC3	Epicc	IC3
Intents & Filters	6,474	4,607 (71%)	5,555 (86%)	1,764 (27%)	839 (12%)	103 (2%)	80 (1%)
URIs	629	249 (40%)	430 (68%)	195 (31%)	96 (15%)	185 (29%)	103 (16%)
Total	**7,103**	**4,856 (68%)**	**5,985 (84%)**	**1,959 (28%)**	**935 (13%)**	**288 (4%)**	**183 (3%)**

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 99 fewer URI values imprecisely than Epicc, thanks to our new string analysis.

There are several reasons why IC3 missed 80 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 31 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 935 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 489 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 42,238 links. In contrast, the matching that used Epicc values yielded 192,662 links. When performing inter-component analysis, fewer potential links imply fewer false positives (since the ICC value computation and matching are conservative) [32]. The 78 percent reduction in potential targets is a very significant gain in precision. The reason why a 16 percent gain in ICC value precision resulted in a 78 percent 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 60,502 seconds using our tool, or slightly less than 17 hours of compute time. That is about 123 seconds per application on average. The processing time was dominated by the IDE problem solver and the string solver, taking 85 percent of the time overall. The second most time-consuming function was the entry point building procedure of [1], taking 11 percent of the total time. Soot analyses (class loading, type inference, final call graph construction, etc.) took 2 percent of the time. Other parts of the analysis (e.g., COAL model parsing, result generation) took 2 percent 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.

8 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 full data flow models (including semilattices and data flow functions) for each object, as it was done in Epicc [35]. We have also found that it is less prone to errors, since it is simpler to verify COAL specifications than it is to check the correctness of complex flow functions and semilattices. In addition, the 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 using an IDE model [38], but it is possible to use reductions to other types of problems [37]. Finally, addressing cases where modeled objects reference other modeled objects in a principled way has allowed us to model complex inter-object relationships such as the one between Android Uri, UriBuilder and Intent. For example, at Line 8 of Fig. 2b, data flows from the uri variable to the $intent$ object. The COAL language enables seamless support for such flows by providing the constructs demonstrated at Line 12 of Fig. 2c. The COAL solver supports these constructs with the fixed point iteration described in Section 6.4.

We estimate that writing specifications for all modifiers and queries for Android took us approximately five hours using the 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.

In addition to the fields used for resolving their targets (e.g., *action* and *categories*), Intents have an *extras* field, which stores data of arbitrary types in the form of key-value pairs. Since the values may have arbitrary types, IC3 does not model them. However, since the keys are strings, IC3 propagates them in a way similar to the *categories* field.

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.

9 Related Work

Single-valued interprocedural constant propagation has been studied in the past [6], [18], [29], [38]. 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], [28] has also been studied. While our constant propagation is also multi-valued, it propagates composite types. As we explain in Section 3, 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], [21], [22], 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], [14]. Other work has performed inter-component dynamic taint analysis [15]. Static analysis has also been investigated. ComDroid [8] attempts to determine a limited number of properties of Intents. Epicc [35] 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 6.4 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 [17] 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], [19], [23], [30], [39], [42], 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 [25] 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. Using a language of constraints that reflect the structure of the program was also used in other domains, such as the inference of reference immutability, in the Javari extension of Java [36].

10 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.

Acknowledgments

The authors thank Matthew Dering for writing the initial version of the ICC matching program. They also thank William Harris for comments provided during the writing of this article. This material is based upon work supported by National Science Foundation Grants Nos. 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. The material in this paper was presented in part at the 37th International Conference on Software Engineering (ICSE'15).

References

[1] 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 *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2014, pp. 259–269.

[2] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proc. 13th Int. Conf. Compiler Construction*, 2004, pp. 5–23.

[3] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proc. 1st ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, Jul. 2012, pp. 3–8.

[4] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 241–250.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. 19th Annu. NDSS Symp.*, Feb. 2012.

[6] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proc. SIGPLAN Symp. Compiler Construction*, 1986, pp. 152–161.

[7] D. R. Chase, M. Wegman, and F. Kenneth Zadeck, "Analysis of pointers and structures," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 1990, pp. 296–310.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. 9th Int. Conf. Mobile Syst. Appl. Serv.*, 2011, pp. 239–252.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th Int. Conf. Static Anal.*, 2003, pp. 1–18.

[10] P. Cousot and R. Cousot, "Automatic synthesis of optimal invariant assertions: Mathematical foundations," in *Proc. Symp. Artif. Intell. Program. Lang.*, 1977, pp. 1–12.

[11] 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," in *Proc. 19th Australasian Conf. Inf. Security Privacy*, 2014, pp. 446–453.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[13] M. Dering and P. McDaniel, "Android market reconstruction and analysis," in *Proc. Mil. Commun. Conf.*, 2014, pp. 300–305.

[14] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. 20th USENIX Conf. Security*, 2011, pp. 23–23.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. Operating Syst. Design Implementation*, 2010, pp. 1–6.

[16] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. 20th USENIX Conf. Security*, 2011, pp. 22–22.

[17] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587.

[18] D. Grove and L. Torczon, "Interprocedural constant propagation: A study of jump function implementation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 1993, pp. 90–99.

[19] P. Hooimeijer and W. Weimer, "A decision procedure for subset constraints over regular languages," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2009, pp. 188–198.

[20] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1036–1046.

[21] N. D. Jones and S. S. Muchnick, "Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra," in *Proc. 21st Annu. Symp. Found. Comput. Sci.*, 13–15 Oct. 1980, pp. 185–190.

[22] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Proc. 9th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 1982, pp. 66–74.

[23] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for string constraints," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 105–116.

[24] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. 3rd ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2014, pp. 1–6.

[25] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using spark," in *Proc. 12th Int. Conf. Compiler Construction*, 2003, pp. 153–169.

[26] L. Li, A. Bartel, T. Bissyande, 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 *Proc. 37th Int. Conf. Softw. Eng.*, May 2015, pp. 280–291.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 229–240.

[28] E. Merlo, J. F. Girard, L. Hendren, and R. De Mori, "Multi-valued constant propagation for the reengineering of user interfaces," in *Proc. Conf. Softw. Maintenance*, Sep. 1993, pp. 120–129.

[29] R. Metzger and S. Stroud, "Interprocedural constant propagation: An empirical study," *ACM Lett. Program. Lang. Syst.*, vol. 2, no. 1-4, pp. 213–232, Mar. 1993.

[30] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proc. 14th Int. Conf. World Wide Web*, 2005, pp. 432–441.

[31] Trustlook News. (2013, Nov.). Emergency: Android in-app billing verification bypass vulnerability [Online]. Available: http://blog.trustlook.com/index.php/emergency-android-app-billing-verification-bypass-vulnerability/

[32] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2016, pp. 469–484.

[33] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to Java bytecode," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 6:1–6:11.

[34] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. 37th Int. Conf. Softw. Eng. - Volume 1*, 2015, pp. 77–88.

[35] 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 *Proc. 22nd USENIX Conf. Security*, 2013, pp. 543–558.

[36] J. Quinonez, M. S. Tschantz, and M. D. Ernst, "Inference of reference immutability," in *Proc. 22nd Eur. Conf. Object-Oriented Program.*, 2008, pp. 616–641.

[37] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Sci. Comput. Program.*, vol. 58, no. 1-2, pp. 206–263, 2005.

[38] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comput. Sci.*, vol. 167, no. 1-2, pp. 131–170, Oct. 1996.

[39] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for Javascript," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 513–528.

[40] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. John Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 515–526.

[41] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the soot framework: Is it feasible?" in *Proc. 9th Int. Conf. Compiler Construction*, 2000, pp. 18–34.

[42] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2007, pp. 32–41.

[43] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2014, pp. 1329–1341.

[44] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications," in *Proc. 21th Annu. Netw. Distrib. Syst. Security Symp.*, 2014.

Damien Octeau received the BSc and master's degrees from Ecole Centrale de Lyon, France, in 2007 and 2010, respectively, and the MSc and PhD degrees in computer science and engineering from the Pennsylvania State University, in 2010 and 2014, respectively. He was a research associate with a joint appointment at the Department of Computer Sciences at the University of Wisconsin-Madison and at the Department of Computer Science and Engineering at the Pennsylvania State University. He is currently at Google in the area of mobile security. His research interests include systems, mobile and software security, and program analysis. He is a member of the IEEE.

Daniel Luchaup received the PhD degree in computer science from the University of Wisconsin, Madison in 2015. He spent a number of years as a software engineer working on compilers and software tools. He is interested in programming languages, security, and software engineering. He is curently a postdoctoral researcher at CyLab, Carnegie Mellon University, working on static analysis of binary code.

Somesh Jha received the BTech degree from the Indian Institute of Technology, New Delhi in electrical engineering and the PhD degree in computer science from Carnegie Mellon University in 1996. He is currently a professor in the Computer Sciences Department at the University of Wisconsin, Madison, which he joined in 2000. His work focuses on analysis of security protocols, survivability analysis, intrusion detection, formal methods for security, and analyzing malicious code. Recently, he has also worked on privacy-preserving protocols. He has published more than 150 articles in highly refereed conferences and prominent journals. He has received numerous best-paper awards. He also received the US National Science Foundation career award in 2005.

Patrick McDaniel received the PhD degree at the University of Michigan. Prior to pursuing his PhD, he was a software architect and a project manager in the telecommunications industry. He is a distinguished professor in the School of Engineering and Computer Science at The Pennsylvania State University, codirector of the Systems and Internet Infrastructure Security Laboratory. He is also the program manager and the lead scientist for the Army Research Laboratory's Cyber-Security Collaborative Research Alliance. His research efforts centrally focus on a wide range of topics in security and technical public policy. He a fellow of the IEEE and ACM.