FLOWMINER: Automatic Extraction of Library Data-Flow Semantics for Partial Program Analysis

Tom Deering, Ganesh Ram Santhanam, Suresh Kothari
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa 50011
{tdeering, gsanthan, kothari}@iastate.edu

***Abstract*—We propose FLOWMINER, a tool for mining *expressive, fine-grained* data-flow summaries from Java library bytecode. FLOWMINER captures enough information to enable context, type, field, object and flow-sensitive partial program analysis of applications using the library. FLOWMINER's summaries are *compact*- flow details of a library that are non-critical for future partial program analysis of applications are elided into simple edges between elements that are accuracy-critical. Hence, summaries extracted by FLOWMINER are an order of magnitude smaller than the original library. We present (i) novel algorithms to extract *expressive, fine-grained, compact* summary data-flows from a Java library, (ii) *graph summarization* paradigm that uses a multi-attributed directed graph as the mathematical abstraction to represent summaries, (iii) open-source implementation (FLOWMINER) of the above that saves summaries in a portable format usable by existing analysis tools, and (iv) experiments with recent versions of Android showing that FLOWMINER significantly advances the state-of-the-art tooling in accuracy.**

Website: http://powerofpi.github.io/FlowMiner/

I. INTRODUCTION

Static analysis has emerged as a powerful paradigm [1], [2], [3] for the analysis of real world software, as evidenced by the widespread use of static analysis tools in the government and industry [4], [5]. Despite their success, static analysis techniques share a common Achilles heel when it comes to *partial program analysis*, i.e., the analysis of a proper subset of a program's implementation (as opposed to *whole-program analysis* [6]). Real-world software applications are built on top of reusable libraries and frameworks (see Figure 1) that are often much larger than the applications which use them. For example, Android [7], [8] applications are often *three orders of magnitude* smaller than the Android framework itself. This makes whole-program analysis, wherein such pieces would be included, infeasible. Yet the alternative of excluding these components leads to unsound and/or incomplete results in practice, which is unacceptable for safety and security critical analysis use cases (e.g., malware detection [9], [10], [11], [12]). Prior work to summarize libraries by relating inputs and outputs [13] provides a better alternative to excluding libraries from analysis of an application altogether; however it is inadequate as it may be too *coarse* (e.g., flows to or from a field in a class are counted as flow to or from the object.) to be used accurately in a future analysis. Hence, there is a pressing need for algorithms and tools that compute fine-grained and application-agnostic summaries of a library's semantics in a way that can be reused in future analyses.

*This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

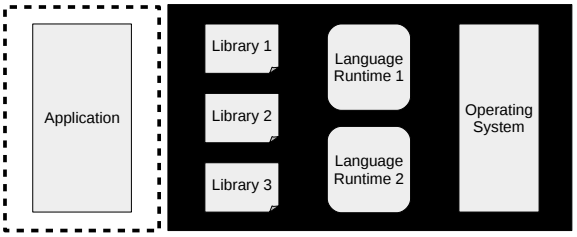

Figure 1: Partial program analysis omits reusable libraries and frameworks.

In this work we present FLOWMINER, a novel approach to automatically reason about a library and extract *fine-grained* yet *compact* data-flow summaries of a given library. We employ a *graphical summarization* paradigm wherein the library summary is expressed as a multi-attributed directed graph, which is more expressive than coarse, binary relationships between inputs and outputs. FLOWMINER extracts application-agnostic summary data-flow graph semantics through a one-time analysis of library bytecode. This summary is serialized in a portable format, and can be reused by other analysis tools to accurately and scalably analyze applications of interest.

Motivation. Our motivation for FLOWMINER comes from a challenge we faced as participants of DARPA's Automated Program Analysis for Cybersecurity (APAC) program [14], where we were tasked with creating partial program analyses for Android apps to detect malware. The typical size of apps we were asked to analyze was small (1kLOC - 100kLOC). However, Android apps are effectively plugins to the much larger Android framework – Android 4.4.4 (KitKat) [15] contains over 2 million LOC, which is orders of magnitude larger than the size of a typical app. Interaction between apps and the framework is ubiquitous. For example, there are many information flows that pass back and forth between the app and framework, often asynchronously, that must be tracked to uncover possible malicious behaviors. In this scenario, whole-program analysis of the app (by including the entire Android framework) solves the problem, but limits scalability.

Optimizing Expressiveness and Compactness. When summarizing the data-flow semantics of a library, certain key artifacts in the library will be crucial to its data-flow. For example, individual field definitions must be present if a summary is

to be used in a field-sensitive way, and individual call sites must be preserved if library callbacks are to be captured. We find that more than 90% of summarized field flows will be false positives if field definitions are not retained (we present empirical results of our experiments that support this claim in Section VII). Consequently, fields, method call sites, literal values, and formal and informal method parameters and return values are all *key* artifacts of a flow that must be preserved in a summary data-flow.

On the other hand, non-key features such as uninteresting def-use chains of assignments do not add value to the paths in which they participate, and can be abstracted away in the summary. FLOWMINER *elides* (replaces paths with direct edges) uninteresting flow details to arrive at an abstract data-flow graph that contains the key artifacts crucial to the data-flow and reachability information between them, and is much more compact than the original program graph. This allows us to achieve significant savings and enhanced scalability versus the original library, while preserving *soundness*, i.e., the flows that are preserved in FLOWMINER's summary are precisely those that are actually possible at runtime. We find that our summaries are compact, containing only about a third of the nodes and a fifth of the edges of the original program graph when tested on recent versions of Android.

Contributions. In summary, the following are the contributions of this paper.

- We develop a static analysis technique to automatically generate *fine-grained*, *expressive* summary specifications given the source or bytecode of any Java library.
  - Our algorithms identify and retain *key artifacts* of the program semantics necessary *to allow context, object, flow, field, and type-sensitive data-flow analyses* in the future when using our summaries.
  - Our summaries use a *rich, multi-attributed graph as the mathematical abstraction* to encode fine-grained summaries, rather than coarse binary relations between the inputs and outputs of library API.
  - The generated summaries are *compact* and significantly smaller than the original library, as *non-key features* in the flows of the original library are elided into key paths.
- We provide FLOWMINER, *an open-source reference implementation* [16] of our algorithms that extracts summaries given the source or bytecode of a library and exports them to a portable, tool-agnostic format.
- We validate FLOWMINER by demonstrating that our summaries of popular libraries are much smaller than the original programs, yet more expressive and accurate than other state-of-the-art summary techniques.

Organization. The rest of the paper is organized as follows. Section II provides a motivating example of an Android application whose malicious behavior cannot be detected without data-flow semantics for the Android library. Section III describes the program graph abstraction for representing program semantics used by FLOWMINER to extract summaries. Section IV outlines our approach, Section V provides algorithmic details, and Section VI discusses implementation details of FLOWMINER. We evaluate and characterize our work in Section VII, compare it with prior work in Section VIII, and conclude in Section IX.

II. MOTIVATING EXAMPLE

We put forward a motivating example of an Android application with a malicious behavior that cannot be detected without including the data-flow semantics of the library (Android) or its summary in an analysis. While we illustrate the need to summarize data-flow semantics of libraries using an Android example, it arises in many applications not limited to malware detection, Android, or even the Java programming language. The techniques we propose in this paper for data-flow summarization are generic and widely-applicable.

Malicious App. Consider the Android app shown in Listing 1. `MainActivity` is a subclass of `Activity`, so it defines an application screen. It overrides two lifecycle methods; the Android framework will call `onCreate` when `MainActivity` is initialized for the first time, and it will call `onPause` when `MainActivity` loses user focus. Therefore, at some point when this app is run, there will be a call to `onCreate` followed by a call to `onPause`. This triggers a latent malicious behavior.

Consider the `onCreate` method. On lines 8-9, the app retrieves the device ID and SIM card serial number, writing them to member fields. Lines 10-20 define and instantiate an anonymous `AsyncTask`, which is a threading mechanism defined by the Android library. A call to `AsyncTask.execute(params)` causes Android to run the object's `doInBackground(params)` method in a new thread, passing along the same arguments. Line 10 writes this anonymous `AsyncTask` object to a member field.

If we examine `onPause()`, we see that the `AsyncTask` is asynchronously executed with the device ID and SIM card serial number as arguments. The `doInBackground` method constructs a shady `URL` for a server operated by an attacker on

```
1  public class MainActivity extends Activity {
2    private String deviceID;
3    private String simSerial;
4    private AsyncTask<String,Void,Void> at;
5    @Override
6    protected void onCreate(Bundle savedInstanceState) {
7    TelephonyManager tm = (TelephonyManager) getSystemService
         (Context.TELEPHONY_SERVICE);
8    deviceID = tm.getDeviceId();
9    simSerial = tm.getSimSerialNumber();
10   at = new AsyncTask<String,Void,Void>(){
11    @Override
12    protected Void doInBackground(String... params) {
13     try {  String url = "http://evil.com/";
14            for(String s : params){ url += "&" + s; }
15            new URL(url).openConnection();
16     } catch (IOException e) {}
17     return null;
18     }
19    };
20   }
21  @Override
22  protected void onPause(){at.execute(deviceID, simSerial);}
23 }
```

Listing 1: Malicious Android app that uses Android's AsyncTask library class to leak data

lines 13-16, appending the sensitive information to the `URL`. Line 15 opens a connection, causing an HTTP GET request to be issued to the malicious server. This application behavior clearly will leak sensitive device data to `http://evil.com`.

Analysis Without Summaries. Consider how an analyst would hope to detect the malicious flow using a state-of-the-art static analysis tool without including the entire Android framework in the analysis. The analyst would first define `TelephonyManager.getDeviceId` and `TelephonyManager.getSimSerialNumber` to be sensitive information *sources*, and any constructor of `URL` to be a sensitive information *sink*. The analyst would then run a static analysis tool, hoping to detect data-flows from any of the sources to any of the sinks. Observe that static analysis tools can follow the data-flows from Android's `TelephonyManager` into the `onCreate` method, then through member field definitions, leading to the parameters of a call to `AsyncTask.execute` (defined by Android). The analyzer can follow the flow no further, as it has no information about the internal (private) implementation of `AsyncTask`. Thus static analysis fails to detect the malicious data-flow because data-flow semantics for the Android library are unavailable.

To solve this problem and identify the malicious flow via static analysis, we either have to (a) resort to whole-program analysis by including the entire Android implementation along with the app as input to the static analyzer, which is prohibitively expensive; or (b) include *summary data-flow semantics* for Android that precisely define the data-flow information between Android components necessary to track data-flow through Android. In this example, we require a summary of how data passed to `AsyncTask.execute` flows through the private implementation of Android and back into the app via asynchronous callback.

In Section IV, we provide an overview of our solution for computing precise summaries of a library. We perform an automatic, one-time extraction of summary data-flow semantics within a given library (such as Android). We demonstrate how these summaries can be grafted into the partial program analysis context, enabling us to detect the malicious program behavior presented in the example above. The resolution of this example is described in Section VI.

III. Background: Graph Schema to Represent Program Semantics

We first introduce the graph paradigm for representing and reasoning with a program's structure and semantics.

Program Graph. We represent the structure and semantics of a program $\mathfrak{P}$ as a rich multi-attributed software graph called *program graph*, denoted $G(\mathfrak{P})$. The nodes of $G(\mathfrak{P})$ correspond to artifacts of $\mathfrak{P}$ such as variables, parameters to a method, call sites, classes, methods, etc., and the edges correspond to structural (e.g., contains, overrides, extends, etc.) and semantic (e.g., data-flow, call, control flow, etc.) relationships between those artifacts. To enable reasoning about several possible runtime behaviors of $\mathfrak{P}$, we allow some information in $G(\mathfrak{P})$ to be a conservative estimate of the semantics of $\mathfrak{P}$; in particular, call sites for dynamic dispatches are connected by edges to all potential targets. The eXtensible Common Software Graph (XCSG) [17] schema is an open standard that defines several attributes and tags to represent $G(\mathfrak{P})$ in XML. XCSG tags such as *public, protected, private, abstract, native, static, synchronized, transient* and *volatile* are used to represent keywords. XCSG attributes express multi-valued properties of an element in $G(\mathfrak{P})$, such as *name* and a unique *identifier* for each element, *parameter index* for each parameter of a method, *object alias* for each object in the local context, and *array dimension* of each array type. Some of the important kinds of nodes and edges supported by XCSG[1] are shown in Figure 2. We use Atlas[18], a tool that implements the graph paradigm to generate XCSG representation of $G(\mathfrak{P})$ from $\mathfrak{P}$.

Nodes

- *Project*: Java project
- *Library*: Java library
- *Type*: Type definition
  - *Primitive*: Primitive type
  - *Array Type*: Array type
  - *Annotation*: Annotation type
  - *Interface*: Interface type
  - *Class*: Class type
  - *Enum*: Enum type
- *Method*: Method of a class
- *Variable*
  - *Field*: Field of a type
  - *Parameter*: Parameter of a method
  - *Identity*: Implicit "this" parameter
  - *Return*: Return value of a method
  - *Enum Constant*: Constant enum value
  - *Local*: Local variable

Structural Edges

- *Declares*: Origin declares destination
- *Overrides*: Origin method overrides destination method
- *Supertype*: Destination is a supertype of origin
  - *Extends*: Origin extends destination
  - *Implements*: Origin implements destination
- *Typeof*: Origin is of type destination
- *Returns*: Origin method return type is destination
- *Element type*: Type of array components

Figure 2: Interpretation of nodes and edges in the eXtensible Common Software Graph (XCSG) for representing Java programs.

Querying $G(\mathfrak{P})$. By specifying constraints on XCSG tags and attributes, we can query the program graph $G(\mathfrak{P})$ to obtain call graphs, control flow graphs, data-flow graphs, and type hierarchies. For example, we can identify the parameters named p of methods of local types (types declared within a method) by selecting the nodes tagged as *'methods'*, traversing forward on *'contains'* edges to find local types and their declarations, filtering to nodes tagged as *'parameters'*, then filtering by those whose name is p. Atlas provides a language to query $G(\mathfrak{P})$ with respect to constraints on XCSG tags and attributes, which we will use to extract relevant information needed for the data-flow summary of $\mathfrak{P}$ (an example Atlas script for the above example is shown in Listing 2).

The artifacts in $G(\mathfrak{P})$ that serve as raw material for our summary extraction approach include:

- Program declarative structure
- Type hierarchy relationships (type points to a type it extends

[1] XCSG is a programming language-agnostic generic representation standard for program structure and semantics; for example, the semantics of programs written in C as well as Java can be represented using XCSG tags.

```
Q u = Common.universe();
Q contains = u.edgesTaggedWithAny(XCSG.Contains);
Q methodDeclarations = contains.successors(u.
    nodesTaggedWithAny(XCSG.Method));
Q localTypes = methodDeclarations.nodesTaggedWithAny(XCSG.
    Type);
Q result = contains.forward(localTypes).nodesTaggedWithAny(
    XCSG.Parameter).selectNode(XCSG.Name, "p");
```

Listing 2: Atlas script to find parameters named "p" of methods of local types.

or implements)

- Method override relationships (method points to a method definition that it overrides)
- Static type relationships (variable points to its declared type)
- Call site information: Method signature, Type to search, Informal parameters
- Pre-computed data-flow relationships (variable points to its flow destination): Field reads and writes, Local def-use chains, Local array accesses

We finally represent the extracted summary information using an extension of the XCSG schema (to distinctly represent summary flows). The details of this extension schema are described later in Section VI-A.

IV. Approach

In this section we provide a high-level overview of our novel approach to automatically-extract summary library data-flow semantics. Our approach has the following desirable attributes:

- Targets JVM bytecode for wide applicability
- Automatically extracts summaries without manual effort
- Retains enough details to enable context, object, field, flow and type-sensitive analysis of applications using the library
- Uses portable encoding to allow use by any analysis tool
- Summaries are much smaller than a library itself

Notation. We introduce the following notation and concepts needed to explain the algorithmic aspects of our approach. Let $\mathfrak{P}$ be a program, and $G(\mathfrak{P})$ be its corresponding program graph. Let M be the set of methods defined in $\mathfrak{P}$. For each method $m_i \in M$ let the set $P_i = \{p_1^i, p_2^i \dots p_{|P_i|}^i\}$ denote the formal parameters to m_i, and r_i its return. We denote a method call site by $c := \langle m_j, t^c, P^c, r^c \rangle$ with P^c denoting the set of arguments (parameters passed) from the call site c to m_j and r^c denoting the returned type from m_j. t^c denotes either the `Class` where m_j is defined (if c is a static dispatch), or else the stated type of the reference on which m_j is invoked (if c is a dynamic dispatch). Statically-dispatched call sites do not require runtime information to calculate the target of the call. These include calls to static methods and constructors. Dynamically-dispatched call sites *do* require runtime information to calculate the destination, as is the case for calls to general member methods.

Remark 1. An interesting case arises when an application defines a subtype of a library type – this may introduce new potential runtime targets in the application for dynamic dispatch call sites in the library (callbacks). For example, an application may define implementations of the `java.util.List` interface and pass instances of these types as parameters of calls to the library. Hence, in order for the computed data-flow summaries of the library to be strictly application-agnostic and complete, they cannot pre-resolve a dynamically-dispatched callsite a priori. Our approach to computing data-flow summaries adheres to this principle, which we call the *open world assumption* for computing summaries.

```
static int average(List<Integer> l)
{  int lSum = sum(l); int lLength = l.size(); return lSum/
    lLength; }
static int sum(List<Integer> l)
{ int s = 0;  for(Integer i : l) s += i;  return s; }
```

Listing 3: Malicious Android app using AsyncTask library class to leak data.

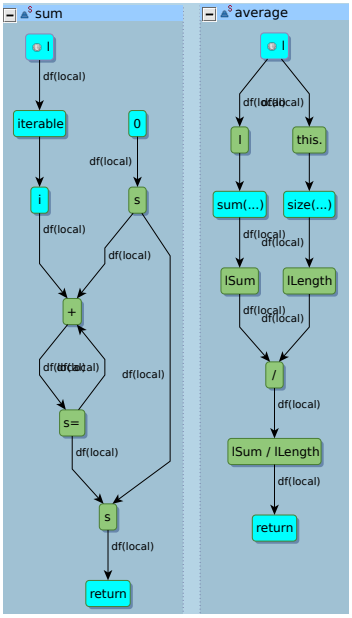

Figure 3: Partial program graph for Listing 3 with key nodes colored cyan

Illustration of Approach. To illustrate the approach taken to extract summaries from $G(\mathfrak{P})$, consider the two methods, `sum` and `average`, defined in Listing 3. A subset of the program graph $G(\mathfrak{P})$ for the corresponding code is shown in Figure 3. Our goal is to arrive at the data-flow summaries in Figure 4. Observe that the summary graph is derived from the original program graph $G(\mathfrak{P})$; undistinguished nodes from $G(\mathfrak{P})$ are removed to simplify the summary flow semantics. However, the summary graph retains critical features of the flows such as literal values, call sites, method signature elements, which we identify as *key* nodes in the program graph, and the flows between them.

To get from $G(\mathfrak{P})$ in Figure 3 to $G^S(\mathfrak{P})$ in Figure 4, we perform the following high-level steps:

1) Compute the program graph $G(\mathfrak{P})$
2) Identify key nodes in $G(\mathfrak{P})$ (colored cyan in Figure 3)
3) Compute flows between key nodes, eliding paths through non-key nodes into simple edges.
4) Compute inter-procedural summary flows by analyzing callsites

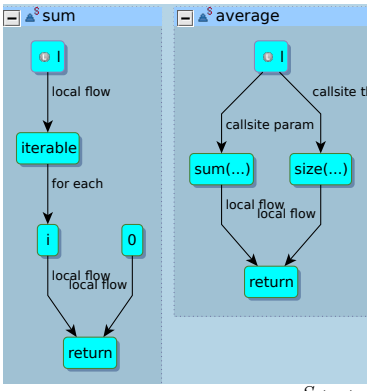

Figure 4: Elided local flow summary $G^S(\mathfrak{P})$ for Figure 3

There are important differences between the program graph $G(\mathfrak{P})$ and the summary graph $G^S(\mathfrak{P})$ obtained. Nodes in $G(\mathfrak{P})$ that are important or *key* features of a data-flow, such as formal method parameters, method return nodes, and literal values, are all retained in $G^S(\mathfrak{P})$. On the other hand, intermediate nodes and edges in the program graph between *key* nodes are *elided* in the summary. For Listing 3, the *key* nodes in $G(\mathfrak{P})$ are colored cyan in Figure 3; these are the only nodes retained in the summary graph (Figure 4).

When intermediate nodes along a flow from key node k_1 to k_2 are removed from the program graph, a summary edge is introduced between k_1 to k_2 to convey the existence of a summary data-flow. For example, in the summary of the `average` method, the nodes corresponding to the variables `lSum`, `lLength`, and the operator `/` are intermediate nodes in Figure 3 that are *elided* in the summary in Figure 4. In their place are direct summary flow edges from the callsites of `sum` and `List.size` to the return value of the method.

Problem Statement. Given a program $\mathfrak{P}$, we formulate the problem of summary extraction as the procedure of automatically generating the summary graph $G^S(\mathfrak{P})$. In the next section, we describe algorithms for each high-level step listed above to automatically compute the summaries from $G(\mathfrak{P})$.

V. Automatic Summary Extraction

Let $\mathfrak{P}$ be a Java library for which we would like to extract summary data-flows. We perform a one-time analysis of $\mathfrak{P}$ to construct the program graph $G(\mathfrak{P})$. We explain our technique for summary computation in two parts. The first subsection describes in detail our algorithm for computing summaries of (local) data-flows within each method, and the following subsection describes the corresponding algorithms for summarizing interprocedural data-flows.

A. Mining Local Flows

Before describing the algorithm to mine summary data-flows local to a method, we first identify *key* nodes in $G(\mathfrak{P})$.

Key Nodes. We define *key* nodes as precisely those nodes in the $G(\mathfrak{P})$ that must be preserved in the summary graph $G^S(\mathfrak{P})$. For the language of Java, the nodes we consider key include: (i) *method signature elements* (formal parameters, formal implicit identity parameter, return node), (ii) *call sites* (informal parameters, informal implicit identity parameter, return value), (iii) *fields*, (iv) *literal values*, (v) *definitions written to and read from fields*, (vi) *array access operators and operands* (array reference operand, array index operand), (vii) *for-each loop iterables and receivers*, (viii) *array components*.

Remark 2. The key nodes in $G(\mathfrak{P})$ will differ based on the language of the library, and hence the notion of key nodes must be well defined for the library's language prior to using our approach. For example, $G(\mathfrak{P})$ for a library written in the C language may contain other key nodes such as pointers to fields and functions.

The algorithm for extracting a summary of local data-flows (i.e., within a method) is based on the idea of *eliding pre-processed def-use chains with respect to the set of* key *nodes* in the method. Given the program graph $G(\mathfrak{P})$, we begin by identifying the set K of *key* nodes in the graph, and then reduce $G(\mathfrak{P})$ by preserving only the nodes in K and the reachability information among them. As a result, all intermediate data-flow nodes and edges that occur on paths between key nodes are *elided* for each method, resulting in a summary graph $G^S(\mathfrak{P})$ that is much smaller than $G(\mathfrak{P})$. Def-use paths occurring between key nodes in a method are merged into simple edges, but key nodes are never elided.

Extracting Summary Flows. Given the set K and the *pre-processed* data-flow graph of def-use chains that can be derived from $G(\mathfrak{P})$, Algorithm 1 computes elided summary data-flows with respect to K. The procedure `MineFlow` iterates over the key nodes in K. For each $k \in K$, `MineFlow` finds the set $K' \subseteq K$ of other key nodes that are reachable along data-flow paths that *do not include other key nodes as intermediates*, using the procedure `ElidedFlow` (Line 3). For each such key node $k' \in K'$, `MineFlow` introduces a summary flow edge from k to k' (Lines 4-5).

Eliding Intermediate Nodes. The procedure `ElidedFlow` computes the set of nearest-reachable key nodes K' for a given key node k by exploring the data-flow graph breadth-first starting from k. The procedure maintains a `frontier` containing the set of nodes that have to be processed, initialized to $\{k\}$. In each iteration, it adds each node f' in the frontier that has a key node successor to the return value (Lines 14-16); and otherwise, it is added to the `frontier` so that further key nodes potentially reachable from k via f' can be searched in a future iteration (Lines 14,17-18). `ElidedFlow` terminates when all nodes in the `frontier` have been processed (Line 12). The set of nodes returned by `ElidedFlow` is exactly the set of key nodes reachable from k via non-key intermediate nodes.

Remark 3. The attributes labeling each summary edge are determined based on the kind of summary relationship being represented. For instance, if the origin or destination is a field definition, then the edge will be labeled with attributes indicating that it is a data-flow from or to a field.

Our summary schema, described in Section VI-A, defines other kinds of relationships as well, including array accesses,

Algorithm 1 Mining summary data-flows

```
    procedure MineFlow(K, G(𝔓))
 2:   for all k ∈ K do
        K′ ← ElidedFlow(k, K, G(𝔓))
 4:     for all k′ ∈ K′ do
          Add summary flow edge from k to k′
 6:     end for
      end for
 8: end procedure

    procedure ElidedFlow(k, K, G(𝔓))
10:   frontier ← {k}
      result ← {∅}
12:   for all f ∈ frontier do
        frontier ← frontier - f
14:     for all f′ s.t. (f, f′) is a data-flow edge in G(𝔓) do
          if f′ ∈ K then
16:         result ← result ∪f′
          else if f′ ∉ frontier then
18:         frontier ← frontier ∪f′
          end if
20:     end for
      end for
          return result
22: end procedure
```

dynamic callsite information information, for-each iteration, and resolved flows to methods. Mining these relationships is straightforward, as they can be taken directly from $G(\mathfrak{P})$ for inclusion in $G^S(\mathfrak{P})$.

B. Mining Interprocedural Flows

The task of mining interprocedural flows involved in method calls, as well as dynamic call site information, is somewhat more complex. First, we must decide which call sites to resolve at present (during summary generation) and which cannot be resolved until summaries are applied in the context of an analysis. If a potential target of a call site may lay outside of the library after an application is introduced into the analysis context, then we *must not* resolve targets of the call site at this time. Clearly static dispatches can be resolved during summary generation, because the targets are unambiguous even with an open-world assumption about future analysis contexts (see Remark 1).

Resolvable and Unresolvable Call Sites. It is important to distinguish between call sites that can be statically-resolved and those which cannot at the time of summary generation. By pre-resolving those which are statically-resolvable to their targets, we generate *sound* data-flow relationships that a client can use, and prevent future rework by clients. Additionally, direct interprocedural flows are more compact to express than leaving a callsite description in the summaries. Thus, it is preferable to identify and resolve statically-dispatchable callsites at the time of summary generation.

Although dynamic dispatches are not statically-resolvable in general, they become so under certain circumstances. For instance, a call to a member method marked `final` or `private` cannot possibly have polymorphic behavior, even under an open-world assumption. Similarly, a call to a member method within a type that is marked `final` or `anonymous` is also unable to result in polymorphism.

The algorithm to mine interprocedural summary flows is shown in Algorithm 2. The procedure `MineCallsiteSummaries` in Algorithm 2 calls the procedure `ClassifyCallsites` to partition the set $\mathcal{C}$ of call sites as described above and returns (a) R^+ containing call sites for which targets may be unambiguously resolved even in the face of an open-world assumption at the time of summary generation, and (b) R^- containing call sites for which multiple targets (presently, or in a future analysis context), may be resolved.

Next, the procedure `MineMethodFlows` is called for R^+. For each call site, this procedure resolves the target using a dispatch calculation[2] (line 23) and adds summary flow edges in $G^S(\mathfrak{P})$ connecting the informal call site parameters Pc to the corresponding formal parameters P_j in the (resolved) target method m_j's definition (lines 24-27). `MineMethodFlows` concludes by connecting the return flows from the return value in the resolved method m_j to the receiving variable at the call site (line 29). Finally, the procedure `MineDynamicDispatch` is called for R^-, wherein the dynamic dispatch information for each call site in the $G(\mathfrak{P})$ is retained in the summary $G^S(\mathfrak{P})$ (lines 34-37) so that a client can resolve them in a future analysis context.

```
1 public final class Integer extends Number implements
      Comparable<Integer> {
2     private final int value;
3     public Integer(int value) { this.value = value; }
4     public byte byteValue() { return (byte) value; }
5     public int compareTo(Integer object) { return compare(
          value,object.value);  }
6     public static int compare(int lhs, int rhs) {
7       return lhs < rhs ? -1 : (lhs == rhs ? 0 : 1); }   ...
8 }
```

Listing 4: Partial implementation of Integer from the Java standard library

Summary Extraction Example Consider the `Integer` class from the Java standard library, a subset of which we show in Listing 4. Its summaries are shown in Figure 5, where elements of $G^S(\mathfrak{P})$ are colored magenta. Note that due to Algorithm 1, e.g., the conditional operators and intermediate definitions in the `compare` method have been elided; and due to Algorithm 2 `compareTo` method has a statically-resolvable call to `compare`. FlowMiner has resolved the call automatically, showing the flow of the two informal parameters in `compareTo` to the formal parameter and identity parameters of `compare`, and the corresponding flow of the return value back to `compareTo`. This example also illustrates field reads and writes, which were imported directly to $G^S(\mathfrak{P})$ from $G(\mathfrak{P})$ during mining. This summary graph enables accurate tracking of flows through the `Integer` class.

VI. Implementation

In this section we describe the implementation details of our approach for statically-extracting, expressing, and subsequently employing data-flow summaries of Java libraries.

Architecture. FlowMiner is implemented as a plugin for the popular Eclipse IDE. As shown in the architectural diagram of

[2] Recall that each call site in R^+ can be resolved to a single target.

Algorithm 2 Mining method flows and dynamic callsite information relationships

procedure MINECALLSITESUMMARIES($\mathcal{C}$)
  $\langle R^+, R^- \rangle$ = CLASSIFYCALLSITES($\mathcal{C}$)
  MINEMETHODFLOWS(R^+)
  MINEDYNAMICDISPATCH(R^-)
end procedure

procedure CLASSIFYCALLSITES($\mathcal{C}$)
  $R^+ \leftarrow \emptyset$
  $R^- \leftarrow \emptyset$
  for all $c \in \mathcal{C}$ **do**
    if c is a static dispatch **then**
      $R^+ \leftarrow R^+ \cup c$
    else if m_i is `final` $\vee$ `private` $\vee$ `constructor` **then**
      $R^+ \leftarrow R^+ \cup c$
    else if t is `final` $\vee$ `private` $\vee$ `anonymous` $\vee$ `array` **then**
      $R^+ = R^+ \cup c$
    else
      $R^- = R^- \cup c$
    end if
  end for
    return $\langle R^+, R^- \rangle$
end procedure

procedure MINEMETHODFLOWS($\mathcal{C}$)
  for all $c := \langle m_i, Pc, r^c, t^c \rangle \in \mathcal{C}$ **do**
    $m_j \leftarrow$ dispatch(c) ▷ *Unambiguous resolution of c to m_j*
    $P^c \leftarrow \{p_1^c, p_2^c \ldots p_{|P^c|}^c\}$ ▷ *Arguments passed at callsite c*
    $P_j \leftarrow \{p_1^j, p_2^j \ldots p_{|P_j|}^j\}$ ▷ *Formal parameters to m_j*
    for all $p_k^c \in P^c$ **do**
      Add method flow summary edge (p_k^c, p_k^j) to $G^S(\mathfrak{P})$
    end for
    Add return flow summary edge (r_j, r^c) to $G^S(\mathfrak{P})$
  end for
end procedure

procedure MINEDYNAMICDISPATCH($\mathcal{C}$)
  for all $c := \langle m_i, Pc, r^c, t^c \rangle \in \mathcal{C}$ **do**
    Add dynamic callsite method edge (c, m_i) to $G^S(\mathfrak{P})$
    Add dynamic callsite type edge (c, t) to $G^S(\mathfrak{P})$
    for all $p_k^c \in P^c$ **do**
      Add dynamic callsite param edge (p_k^c, c) to $G^S(\mathfrak{P})$
    end for
  end for
end procedure

Figure 6, FLOWMINER takes Java library bytecode as input, typically in the form of a JAR archive. This is passed to Atlas that constructs an XCSG representation of the program graph (see Section III) for the library. FLOWMINER then runs the algorithms described in Section V to extract a summarized version of the library's data-flow semantics from the library's program graph. This summary data-flow graph is packaged into a portable XML format according to a schema that extends the XCSG schema (described in the following subsection), which can be used to parse and import summaries into existing tools.

A. A Summary Graph Schema Extension

To represent the summary data-flow semantics, we propose an extension to the XCSG schema from Figure 2. To represent the summary program graph $G^S(\mathfrak{P})$, we introduce several new kinds of local variables (nodes), as well as new relationships (edges), shown in Figure 7. This schema extension allows us to express the semantics of data-flows within methods, flows to and from fields, as well as flows involved in static and virtual

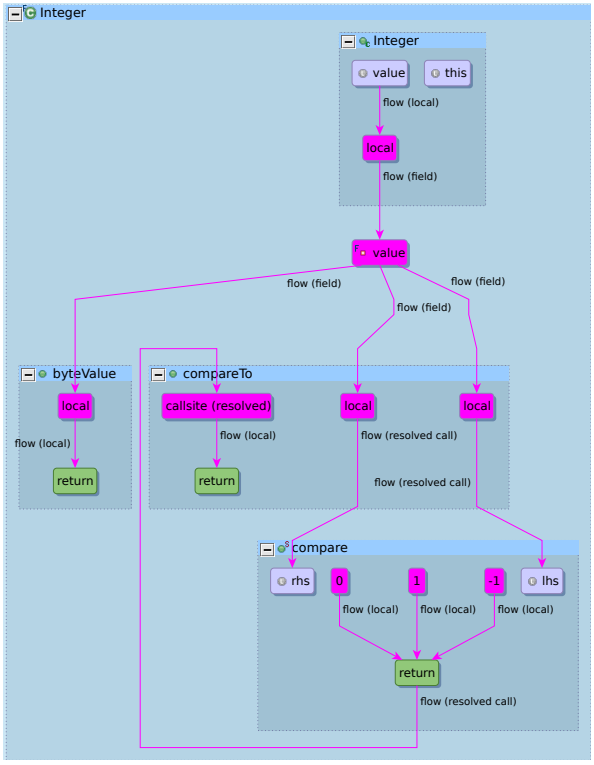

Figure 5: Summary extraction results for the `Integer` class

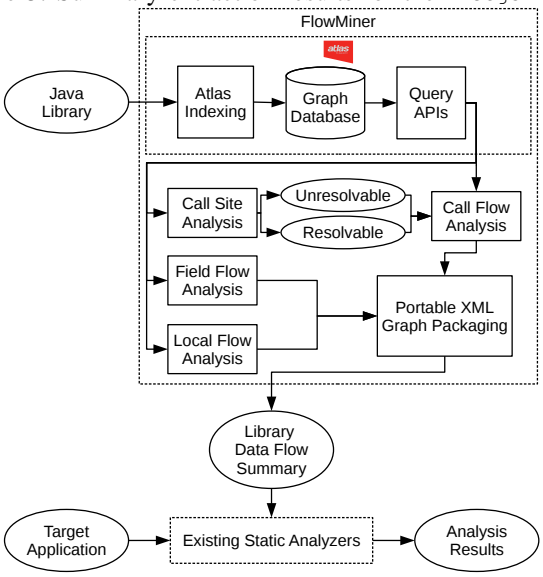

Figure 6: Architecture of FLOWMINER

calls between methods. Our schema extension is organized hierarchically and expressed via the use of tags for kinds and subkinds of nodes and edges in the program graph.

Important Features. There are several important features to note about our summary schema extension. *First*, it is strictly an extension of the XCSG. The node types we introduce are specialized subtypes of local variables. These specializations represent literal values, array components and access operators, method call sites, and other important local definitions. The edge types we introduce represent summary data-flows, array accesses, for-each iteration, and call site information.

Second, our summaries pertain only to data-flow. While a

Summary Nodes

- *Array Component*: Array components (on heap) of a referenced array
- *Array Index Op*: Array access operator, takes array reference and idx as operands and selects a component
- *Call Site*: Represents a method call, as well as the value returned.
  - *Call Site Resolved*: Statically-resolved call site
  - *Call Site Unresolved*: Dynamic call site
- *Literal*: Literal primitive or String value

Summary Edges

- *Array Access*: Connects array index operator to array component node
- *Dynamic Callsite*: Describes an aspect of a dynamic dispatch call site
  - *Dynamic Callsite Param*: Origin stack param is informal parameter for destination call site
  - *Dynamic Callsite This*: Origin object reference is implicit identity parameter for destination call site
  - *Dynamic Callsite Signature*: Call site points to its invoked method signature
  - *Dynamic Callsite Type*: Call site points to its stated identity param type
- *Flow*: data-flow relationship
  - *Array Flow*: Flow to or from array component
  - *Field Flow*: Flow to or from a field
  - *Local Flow*: Flow between local variables
  - *Resolved Method Flow*: Flow to method parameters or from a method return
- *For Each*: Iteration over Iterable or array type to local receiver variable

Figure 7: Summary nodes and edges in FLOWMINER's XCSG schema extension

flow edge (A, B) implies the existence of a control flow path along which this flow happens, we do not retain control flow nodes and edges from $G(\mathfrak{P})$. This allows $G^S(\mathfrak{P})$ to be much more compact than the library itself.

Third, our summaries retain sufficient information to be used with context, type, field, object, and flow sensitivity. The client using the summaries for subsequent analysis is able to decide which categories of sensitivity to employ in order to achieve the desired level of accuracy and speed. One consequence of this philosophy is that we only resolve flows for method call sites when the target can be unambiguously resolved to a single possibility with an open-world assumption, i.e., no matter what other types and methods are introduced into an analysis context by an application, the resolution decision for the call site cannot be changed. We leave dynamic dispatch call sites to be resolved when summaries are applied to an analysis context, since we cannot know ahead of time if that context may introduce new possibilities for the target of the call site. However, we do provide the signature of the call site, as well as the informal stack parameters involved in the call, so that clients may resolve it later.

A Portable XML Schema FLOWMINER serializes the computed $G^S(\mathfrak{P})$ for the library into a portable XML format, so that the summary data-flow semantics can be subsequently parsed and imported by other static analysis tools for a partial program analysis of an application that uses the library. An XML schema document (XSD) defining the grammar for expressing software graphs is provided with the open source reference implementation of FLOWMINER [16].

B. Using Summaries.

Existing static analyzers can apply summaries generated by FLOWMINER to perform a complete and accurate program analysis. What it means to *apply* summaries will differ based on the tooling used by the analyzer. For instance, an analyzer implemented on top of the Atlas platform would 'apply' summaries by translating the portable XML summary document into additional nodes and edges from $G^S(\mathfrak{P})$ for insertion into the program graph $G(\mathfrak{P})$ of an application. Once inserted, these supplementary data-flow semantics will be included in any subsequent analysis.

Recall the example malicious Android app from Section II, for which a static analyzer was unable to detect the malicious behavior. The application asynchronously leaks the user's device ID and SIM card number to an attacker. We defined the values returned by `TelephonyManager.getSimSerialNumber` and `TelephonyManager.getDeviceId` to be sensitive information, and asked our analyzer to track forward data-flows from these artifacts. The result ran into a dead end as soon as the flow disappeared into the private implementation of Android's `AsyncTask.execute` API.

After applying the summary $G^S(\mathfrak{P})$ extracted from a one-time analysis of Android 4.4.4 using FLOWMINER, we are able to obtain the result in Figure 8 on Atlas. Summary nodes and edges ($G^S(\mathfrak{P})$) are highlighted in magenta to distinguish them from elements of the original program graph ($G(\mathfrak{P})$). By employing $G^S(\mathfrak{P})$, our static analyzer is able to detect the entirety of the malicious flow. Observe that after the sensitive information enters `AsyncTask.execute`, our summaries of Android track the asynchronous data-flow involving local flows, a method call, a write and read of a field, and finally a callback into the application (`MainActivity$1.doInBackground`) on a new thread. From there, our analyzer uses $G(\mathfrak{P})$ to follow the flow through an enhanced for loop, string concatenation, and ultimately to the `URL` constructor, completing the leak.

VII. EVALUATION

Experiments. With the goal of evaluating FLOWMINER's accuracy and compactness, we summarized recent versions of the Android operating system listed in column 1 of Table I[3]. We ran our experiments on a multi-core computer with 64 GB RAM, and Eclipse Luna installed with Atlas and FLOWMINER. We created a simple Atlas analyzer to gather the summary statistics listed in Table I.

Expressiveness. The data-flow summaries extracted by FLOWMINER are fine-grained and expressive. For example, the coarse information flow specifications at the granularity of object tainting generated by Clapp et al. [13] can be directly inferred from our summaries – When information in a FLOWMINER summary reaches a member field definition, the corresponding "taint" on the object is implied; and when information flows from a member field to a method return, it is implied that the object "taints" the method return. Hence, FLOWMINER summaries are strictly more expressive than the most closely-related prior work. The presence of registration/callback pairs identified by EdgeMiner [19] can also be

[3] For each version, we downloaded the Android framework from the build for the *aosp_arm-user* device configuration and then generated corresponding JVM bytecode that can be analyzed with Atlas.

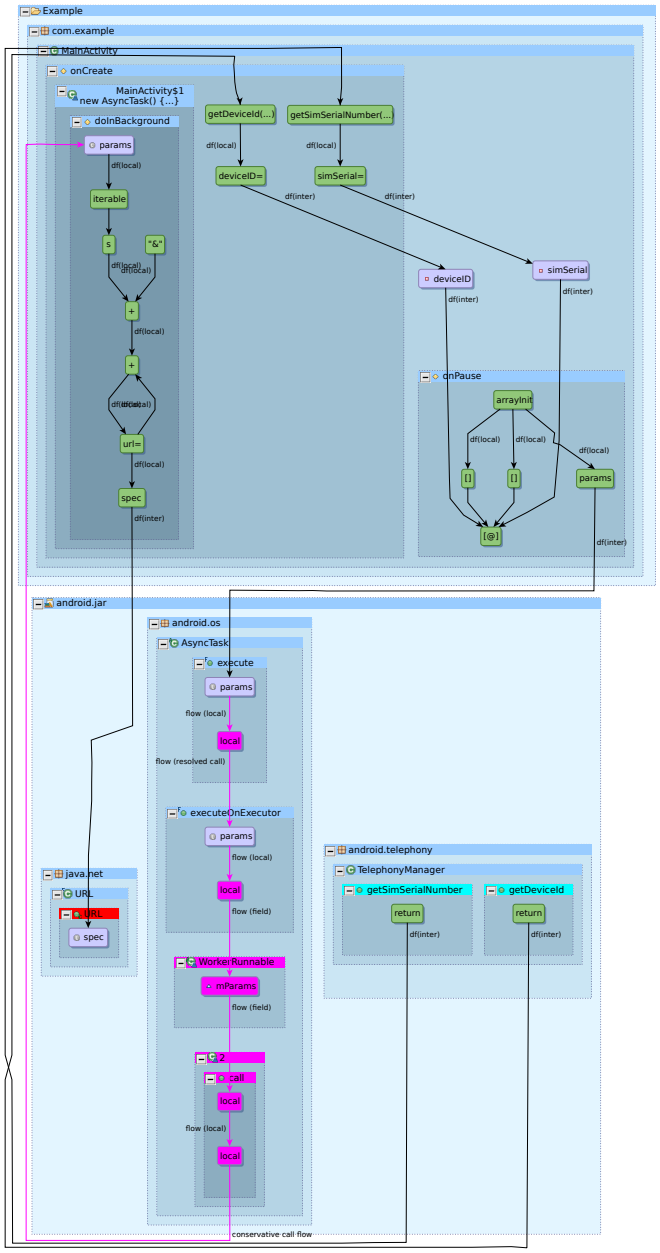

Figure 8: Partial program analysis of malicious app from Listing 1 *with* FLOWMINER *summaries of Android*

inferred from FLOWMINER summaries using details of virtual callsites (for which multiple runtime targets may exist) stored in $G^S(\mathfrak{P})$. More importantly, our summaries can be used more *accurately*. Figure 9 shows how coarse specifications that taint entire objects can lead to an exponential number of implied false positive flows. The figure shows three types with two fields each. Dashed arrows represent transfer of taint at the granularity of objects, while solid arrows represent transfer of taint with field granularity. While a subset of the flows implied by object granularity are true positives (black), the majority of flows will be false positives (red). In general, a flow involving object-granularity summaries that traverses through N classes (with K unrelated fields each) will produce on the order of

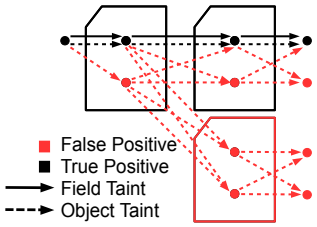

Figure 9: Coarse flow specifications that taint entire objects rather than fields lead to false positive flows.

K^N false positive flows!

Table I shows the number of data-flow edges induced in the summary by FLOWMINER (fine-grained approach that tracks data-flows at field level granularity), which is just over 7% of that induced by the coarse-grained approach (that tracks data-flows at object level granularity). This means that over 92% of the flows induced by coarse-grained approach are false positives compared to those produced by FLOWMINER.

Soundness and Completeness. We observe that the data-flow summaries produced by FLOWMINER are *sound*, i.e., there is never a false positive; if a flow is indicated, it *can* actually occur at runtime. In other words, the removal of any summary flow edge would remove critical information needed later to compute a data-flow in some partial program analysis context. This follows from the way in which our summaries are generated (see Section IV for details). FLOWMINER provides *complete* summaries of data-flow semantics, i.e., does not miss any true flows, except those induced (i) as side effects of reflective calls, and (ii) by mixed-language library code (e.g., Java library calling native C code). This follows from the facts that (i) Atlas fully supports the features of the Java 7 programming language, and hence captures all local, field, and method flows between Java program elements in the program graph it constructs; and (ii) the program graph that is used by FLOWMINER for extracting summary information contains all the possible edges from call sites to potential targets for dynamic dispatches (see Section III).

We also empirically verified the correctness of our FLOWMINER implementation for the Android versions via an Atlas script as follows. We first computed both the program graph $G(\mathfrak{P})$ and the summary graph $G^S(\mathfrak{P})$, and then successfully verified the property that there is a data-flow path from one key node k to another k' in $G^S(\mathfrak{P})$ if and only if there is a corresponding data-flow path from k to k' in $G(\mathfrak{P})$.

Compactness. The compactness of extracted summary artifacts is important for practical use. As an example of the significant compaction achieved by FLOWMINER, $G^S(\mathfrak{P})$ for Android 4.4.4 contains only 36.98% of the nodes and 20.06% of the edges of $G(\mathfrak{P})$ (other versions follow this trend). Hence, our summaries provide an order-of-magnitude savings versus a fully-detailed program graph of a library, yet retain the critical details for use in a partial-program data-flow analysis.

Scalability. The Android framework serves as an adequate test of FLOWMINER's scalability due to its large size. For example, Android 4.4.4 (KitKat) contains roughly 2 million

Library	$\|V\|$	$\|E\|$	$\|V^S\|$	$\|E^S\|$	$\|V^S\|/\|V\|$ (%)	$\|E^S\|/\|E\|$ (%)	Field Flows	Object Flows	% False Positives* avoided
Android 4.2.2	6651277	33964070	2467991	7664280	37.11%	22.57%	1129523	16053060	92.96%
Android 4.3.1	6867245	35165616	2547558	7915450	37.10%	22.51%	1206542	16816490	92.83%
Android 4.4.4	7707688	44150241	2850104	8855585	36.98%	20.06%	1216178	17069468	92.88%
Android 5.0.2	8684208	45649066	3217476	10010647	37.05%	21.93%	1556027	21874691	92.89%

Table I: Experimental results showing the performance of FLOWMINER on four recent versions of Android in terms of compactness and accuracy. (* Percentage of object-granularity flows that are false positives compared to those produced by FLOWMINER)

lines of Java code, omitting comments and white space. At this scale, FLOWMINER completes its one-time analysis and export of data-flow summary semantics within an additional 45 minutes after constructing the original program graph.

VIII. RELATED WORK

Summarizing Call Graphs. There has been a lot of interest in summarizing control flow transitions within a software library. Such control-flow summaries are useful for routine static analysis tasks such as call graph generation [20], [21], [22], [23], tracking of non-trivial calling relationships between application and the library (e.g., asynchronous callbacks in Android) [19] and visualization of control flows from the application to the library and vice-versa [24].

Summarizing Data Flow Graphs. Mining data flows from object-oriented software libraries is an important problem, and is particularly crucial for security-critical analyses. Malware detection in Android apps [14], for example, requires tracking the flow of sensitive information (source, e.g, IMEI number) from the mobile device to potentially harmful destinations (sinks, e.g., a location on the internet).

Callahan first proposed the program summary graph as implemented in PTOOL [25] as a way to compactly represent the inter-procedural call and data flow semantics of the whole program. Rountev et al. [26] pointed out the need to use summaries of data flow semantics when analyzing applications that are dependent on large libraries. They proposed a general theoretical framework for summarizing data flow semantics of large libraries, using pre-computed summary functions per library component and building on the work of Pnueli [27].

Similarly to Rountev et al., Chatterjee et al.[28] compute a summary function for each procedure in the bottom up traversal order of the call graph such that the summary of a caller is expressed in terms of the summary of the callee component(s). More recently, Rountev et al. [29] described an approach called interprocedural distributive environment (IDE) data-flow analysis for summarizing object-oriented libraries that subsumes the class of interprocedural, finite, distributive subset (IFDS) problems [30] by using a graph representation of the data-flow summary functions; their approach abstracts away redundant data-flow facts that are internal to the library, in a similar vein to our concept of *eliding* flows.

Some approaches compute summary information for a software component independently of the callers and callees of that component. For example, Ali et al. developed a tool, AVERROES[31], to generate a placeholder that overapproximates the behaviour of a given library. Their overapproximation may be too coarse to be useful in some scenarios such as malware detection, where we need summaries to retain enough information for various kinds of sensitive analyses.

Summarizing Android Flows. To the best of our knowledge, the most closely related work in summarizing libraries in the context of Android is by Clapp et al. [13], who employ a dynamic analysis approach to mine information flows from Android. Their approach successfully recovers 96% of a set of hand-written information flow specifications. In contrast, FLOWMINER uses static analysis instead of dynamic analysis to identify possible flows within the library, hence avoiding the possibility that some execution paths are not covered. Furthermore, the flow specifications extracted by FLOWMINER track and preserve data flows at the granularity of individual variables and definitions (rather objects) within methods and objects, so we avoid falsely merging unrelated flows. Also, our flow specifications express flows among program elements that are not necessarily on the library API. This allows subsequent analyses to be context, field, type, object, and flow-sensitive. We retain the details of virtual call sites so that flows involving potential callbacks into an application are captured.

IX. CONCLUSION

We presented FLOWMINER [16], a novel solution that uses static analysis techniques to automatically generate an expressive, fine-grained summary of a Java library that can be used for accurate data-flow analyses of applications that use the library. FLOWMINER identifies and retains key artifacts of the program semantics in the summary that are necessary to allow context, object, flow, field, and type-sensitive data-flow analyses of programs using the summarized library. FLOWMINER uses a rich, multi-attributed graph as the mathematical abstraction to store summaries. FLOWMINER's summaries are compact, containing only about a third of the nodes and a fifth of the edges of the original program graph when tested on recent versions of Android, as non-key features in the flows of the original library are elided into key paths. Because FLOWMINER retains individual flows through individual field definitions, in contrast to existing coarse-grained methods that taint entire objects, over 92% of the false positive flows indicated by tainting entire objects are avoided (for the Android framework). FLOWMINER extracts summaries given the bytecode of a library and exports them to a portable, tool-agnostic format. We validate FLOWMINER by demonstrating that our summaries of recent versions of Android are *much smaller than the original library*, yet more expressive and accurate than other state-of-the-art techniques.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[2] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 1999.

[3] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[4] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 217, pp. 5–21, 2008.

[5] T. Ball and S. K. Rajamani, "The s lam project: debugging system software via static analysis," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 1–3.

[6] D. C. Atkinson and W. G. Griswold, "Effective whole-program analysis in the presence of pointers," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 46–55, 1998.

[7] E. Burnette, *Hello, Android: introducing Google's mobile development platform*. Pragmatic Bookshelf, 2009.

[8] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike, *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 2009.

[9] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. IEEE, 2007, pp. 421–430.

[10] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046618

[11] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[12] S. Rosen, Z. Qian, and Z. M. Mao, "Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 221–232.

[13] L. Clapp, S. Anand, and A. Aiken, "Mining explicit information flow specifications from concrete executions," 2014. [Online]. Available: http://www-cs.stanford.edu/people/saswat/research/summarysynthesis.pdf

[14] (2011, July) Automated program analysis for cybersecurity (apac). DARPA. [Online]. Available: https://www.fbo.gov/index?s=opportunity&mode=form&id=a14e4533c2a44c3288b6a29fa6fc5841&tab=core&_cview=1

[15] (2015, May) Android 4.4.4 (kitkat). Google. [Online]. Available: http://www.android.com/versions/kit-kat-4-4/

[16] T. Deering. (2015, April) Iowa State University. [Online]. Available: http://powerofpi.github.io/FlowMiner/

[17] (2015, March) Extensible common software graph. EnSoft Corp. [Online]. Available: http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph

[18] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: A new way to explore software, build analysis tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 588–591. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591065

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," *22nd Annual Network and Distributed System Security Symposium, NDSS San Diego, California, USA*, 2015.

[20] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 685–746, 2001.

[21] W. Zhang and B. Ryder, "Constructing accurate application call graphs for java to model library callbacks," in *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*. IEEE, 2006, pp. 63–74.

[22] D. Yan, G. Xu, and A. Rountev, "Rethinking soot for summary-based whole-program analysis," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 2012, pp. 9–14.

[23] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP 2012–Object-Oriented Programming*. Springer, 2012, pp. 688–712.

[24] T. LaToza and B. Myers, "Visualizing call graphs," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, Sept 2011, pp. 117–124.

[25] D. Callahan, *The program summary graph and flow-sensitive interprocedual data flow analysis*. ACM, 1988, vol. 23, no. 7.

[26] A. Rountev, S. Kagan, and T. J. Marlowe, "Interprocedural dataflow analysis in the presence of large libraries," in *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria*, 2006, pp. 2–16.

[27] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," *Program flow analysis: Theory and applications*, pp. 189–234, 1981.

[28] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999, pp. 133–146.

[29] A. Rountev, M. Sharp, and G. Xu, "Ide dataflow analysis in the presence of large object-oriented libraries," in *Compiler Construction*, ser. Lecture Notes in Computer Science, L. Hendren, Ed. Springer Berlin Heidelberg, 2008, vol. 4959, pp. 53–68.

[30] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[31] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, 2013, pp. 378–400. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39038-8_16