FlowMiner: Automatic Summarization of Library Data-Flow for Malware Analysis*

Tom Deering[1], Ganesh Ram Santhanam[2], and Suresh Kothari[2]

[1] Workiva, 2900 Unversity Blvd, Ames, Iowa 50010 `tomdeering7@gmail.com`
[2] Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa 50011 `{gsanthan, kothari}@iastate.edu`

Abstract. Malware often conceal their malicious behavior by making unscrupulous use of library APIs. Hence any accurate malware analysis must track data-flows not only through the application but also through the library. Libraries like Android (2 mLOC) are too large to be analyzed repeatedly with each application, hence we need to compute data-flow summaries of libraries that are expressive enough to reveal possible malicious flows, and compact to be included in malware analysis along with each application.

We present FlowMiner, a novel approach to automatically extract the data-flow summary of a Java library, given its source or bytecode. FlowMiner's summaries are **fine-grained**, i.e., preserve key artifacts from the original library to enable accurate context, object, field, flow and type-sensitive malware analysis of applications in conjunction with the library. Unlike prior summarization techniques, FlowMiner resolves method calls to **anonymous classes** *to a single target*, making it more precise. FlowMiner's summaries are **compact**, e.g., contain only about a third (fourth) of the nodes (edges, resp.) in the data-flow semantics of recent versions of Android. FlowMiner's summaries are stored in XML, allowing any analysis tool to use them for analysis.

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

1 Introduction

Modern software is increasingly built on top of reusable libraries, and when such libraries are large, the static analysis of an application together with its libraries becomes prohibitively expensive. An alternative is to analyze an application without its libraries. However, for security-critical analyses, this is inaccurate because it discounts data flow through the libraries. In particular for malware detection [17,14,26,20], such inaccuracies are unacceptable because the data flows

*This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

through the library may provide the critical piece of missing evidence to reveal the malicious behavior of an application.

For example, Android [7,19] applications (apps) are often significantly smaller than the Android framework itself. While a typical app may have of the order of 100 *k* LOC, Android 4.4.4 (KitKat) [2] contains over 2 million LOC. Further, Android allows many mechanisms for information flows that pass back and forth between the app (many of which are asynchronous), all of which must necessarily be incorporated into an analysis to uncover possible malicious behaviors. This is not specific to Android however; in most malware analysis and other security audit use cases, it is essential to account for data flows through the library in order to avoid missed detections.

Library Summaries for Malware Detection. This paper focuses on the creation of the data flow summary of a Java library, which is a subset of the original data flow semantics of the library. It is desirable that the summary is (a) *compact*, i.e, is smaller than the library, and (b) *fine-grained*, i.e., preserves enough information so that it can be used (instead of the entire library) when analyzing an application to allow accurate detection of malicious flows. Summaries are application-agnostic, and once created be reused for analyzing any application.

Prior work on summarizing libraries are inadequate as their summaries are too *coarse* to be used accurately in a future analysis. For example, in [11], flows to or from a field in a class are counted as flow to or from the object, and the summary of a method is represented as simple mappings between its input parameters and return values. These preclude the summary from being used in a subsequent sensitive analysis accurately. Similarly, [22] does not resolve calls to anonymous classes as monomorphic, although such calls only have a single target. Hence, there is a need for algorithms and tools that compute fine-grained, compact and application-agnostic summaries of a library's semantics with enough information to be reused accurately in any future analyses of an application that uses the library.

FlowMiner. In this work we present FlowMiner, a novel approach to automatically extract *fine-grained* yet *compact* data-flow summaries of a Java 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 stored in a portable format, and can be reused by other analysis tools to accurately, scalably analyze applications.

FlowMiner's summaries are **fine-grained** because they preserve *key artifacts* in the library that provide crucial information about its data-flow semantics. 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 found 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 6). Consequently, FlowMiner preserves fields, method call sites, literal values, and formal and informal method parameters and return values as *key* artifacts in the summary data-flow.

FlowMiner's summaries are **compact** because FlowMiner removes from the summary non-key features (e.g., irrelevant def-use chains of assignments that do not contribute flow information), which are of value to subsequent analyses. FlowMiner also resolves method calls to **anonymous classes** *to a single target*, making it more precise. FlowMiner *elides* (replaces paths with edges) these uninteresting flow details to arrive at a compact data-flow graph containing only the key artifacts crucial to the data-flow and reachability information between them (e.g., FlowMiner's Android summary contains only about a third of the nodes and a fifth of the edges of the original program graph). Arguably, this makes subsequent analyses to be more scalable when using our summary versus the original library. Importantly, FlowMiner is **sound** in the following sense – each flow preserved in FlowMiner's summary is *actually possible* at runtime in the context of some application.

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

– A static analysis technique to automatically generate *fine-grained*, *expressive* data flow summary given the source or bytecode of any Java library that
  • Preserves *key artifacts* of the program semantics needed *to allow subsequent context, object, flow, field, and type-sensitive data-flow analyses*
  • Uses a *rich, multi-attributed graph as the mathematical abstraction* to encode fine-grained summaries
  • Extracts *compact* summaries much smaller than the original library by eliding *non-key features* in the flows of the original library into key paths.
– FlowMiner*, an open-source reference implementation* [12] of our algorithms that extracts summaries given the source or bytecode of a library and exports them to a portable, tool-agnostic format.
– Evaluation of FlowMiner's compactness and expressiveness on the recent versions of Android, and a comparison with the state-of-the-art.

Organization. The rest of the paper is organized as follows. Section 2 provides a motivating example of an Android application whose malicious behavior cannot be detected without data-flow semantics for the Android library. Section 3 outlines our approach, Section 4 and Section 5 provide algorithmic and implementation details of FlowMiner. We evaluate our work in Section 6, compare it with prior work in Section 7, and conclude in Section 8.

2 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

```
 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 1.1: Malicious Android app that uses Android's AsyncTask library class to leak data

the Java programming language. The techniques we propose in this paper for data-flow summarization are generic and widely-applicable.

Malicious App. Let us see the difficulty an analyst would encounter in detecting malware in an app without including the Android library or its appropriate summary. In the Android app shown in Listing 1.1, `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 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 3, 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 5.

2.1 Background: Graph Schema to Represent Program Semantics

We use the graph paradigm for representing and reasoning with a program's structure and semantics. In this paradigm, the structure and semantics of a program $\mathfrak{P}$ is represented 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. We use the Atlas [13] platform to generate $G(\mathfrak{P})$ given $\mathfrak{P}$ [3]. Atlas stores $G(\mathfrak{P})$ in an XML format following the eXtensible Common Software Graph (XCSG) [3] schema, an open XML standard, and provides a language to query $G(\mathfrak{P})$. We use this query language in our implementation to extract relevant information needed for constructing the data-flow summary of $\mathfrak{P}$.

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

[3] We omit details of the Atlas platform; the interested reader can refer [13].

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

3 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 begin by introducing the 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 $P_i = \{p^i_1, p^i_2 \dots p^i_{|P_i|}\}$ 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.

```java
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 1.2: Computing the average and sum of a set of Integers.

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 1.2. A subset of the program graph $G(\mathfrak{P})$ for the corresponding code is shown in Figure 1. Our goal is to arrive at the data-flow summaries in Figure 2. 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.

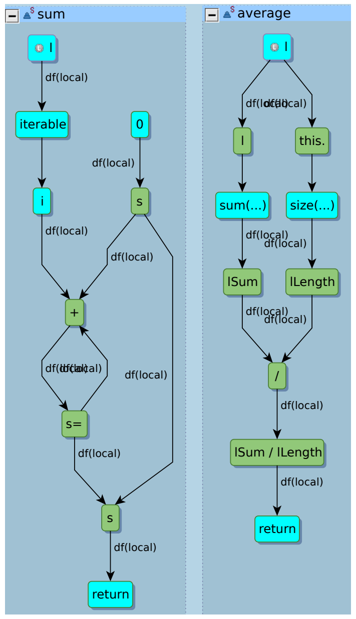

Fig. 1: Partial program graph for Listing 1.2 with key nodes colored cyan

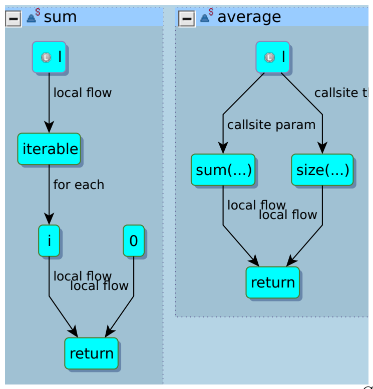

Fig. 2: Elided local flow summary $G^S(\mathfrak{P})$ for Figure 1

To get from $G(\mathfrak{P})$ in Figure 1 to $G^S(\mathfrak{P})$ in Figure 2, 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 1)
3. Compute flows between key nodes, eliding paths through non-key nodes into simple edges (details in Section 4.1)
4. Compute inter-procedural summary flows by analyzing callsites (in Section 4.2)

We note 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 1.2, the *key* nodes in $G(\mathfrak{P})$ are colored cyan in Figure 1; these are the only nodes retained in the summary graph (Figure 2).

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 1 that are *elided* in the summary in Figure 2. In their place are direct summary flow edges from the callsites of `sum` and `List.size` to the return value of the method.

In the next section, we describe algorithms for each high-level step listed above to automatically compute $G^S(\mathfrak{P})$ from $G(\mathfrak{P})$.

4 Automatic Summary Extraction

Given a Java library program $\mathfrak{P}$, we perform a one-time analysis of $\mathfrak{P}$ to construct the program graph $G(\mathfrak{P})$ (see Section 2.1). We explain our technique for summary computation in two parts. Section 4.1 describes in detail our algorithm to compute summaries of (local) data-flows within each method. Section 4.2 describes the corresponding algorithms to compute interprocedural data-flows.

4.1 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 procedure `ElidedFlow` (Line 3). For each 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); since there are clearly finite number of nodes in a frontier, `ElidedFlow` always terminates. 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 also stores other kinds of relationships including array accesses, dynamic callsite information information, for-each iteration, and resolved flows to methods. These relationships from $G(\mathfrak{P})$ are included in $G^S(\mathfrak{P})$.

4.2 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

Algorithm 1 Mining summary data-flows

```
    procedure MineFlow($K$, $G(\mathfrak{P})$)
2:    for all $k \in K$ do
        $K' \leftarrow$ ElidedFlow($k$, $K$, $G(\mathfrak{P})$)
4:      for all $k' \in K'$ do
          Add summary flow edge from $k$ to $k'$
6:      end for
      end for
8:  end procedure

    procedure ElidedFlow($k$, $K$, $G(\mathfrak{P})$)
10:   frontier $\leftarrow \{k\}$
      result $\leftarrow \{\emptyset\}$
12:   for all $f \in$ frontier do
        frontier $\leftarrow$ frontier - $f$
14:     for all $f'$ s.t. $(f, f')$ is a data-flow edge in $G(\mathfrak{P})$ do
          if $f' \in K$ then
16:         result $\leftarrow$ result $\cup f'$
          else if $f' \notin$ frontier then
18:         frontier $\leftarrow$ frontier $\cup f'$
          end if
20:     end for
      end for
          return result
22: end procedure
```

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[4] (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

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

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

Listing 1.3: Partial implementation of Integer from the Java standard library

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, `MineDynamicDispatch` is called on 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. `MineCallsiteSummaries` terminates despite the presence of recursive calls, as it iterates over the (finite number of) callsites only once.

4.3 Summary Extraction Example

Consider the `Integer` class from the Java standard library, a subset of which we show in Listing 1.3. Its summaries are shown in Figure 3, 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.

5 Implementation

Architecture. FlowMiner is implemented as a plugin for the popular Eclipse IDE. As shown in the architectural diagram of Figure 4, 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 2.1) for the library. FlowMiner then runs the algorithms described in Section 4 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 [3] that can be used to parse and import summaries into existing tools.

An XML schema definition (XSD) for expressing summary graphs in XML is provided with the open source reference implementation of FlowMiner [12]. This can be used by other static analysis tools to parse and import the data-flow summary of a library for analysis of an application that uses the library.

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 $m_j$
    $P^c \leftarrow \{p_1^c, p_2^c \dots p_{|P^c|}^c\}$        ▷ Arguments passed at callsite $c$
    $P_j \leftarrow \{p_1^j, p_2^j \dots p_{|P_j|}^j\}$        ▷ 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
```

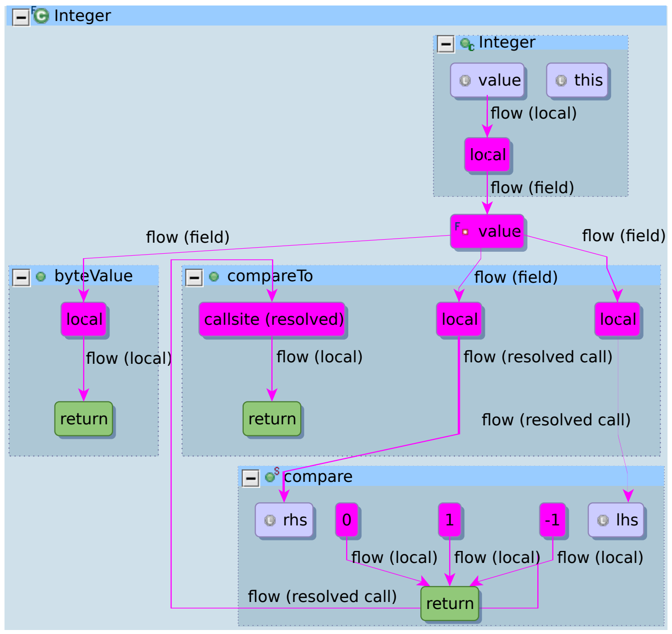

Fig. 3: Summary extraction results for the `Integer` class

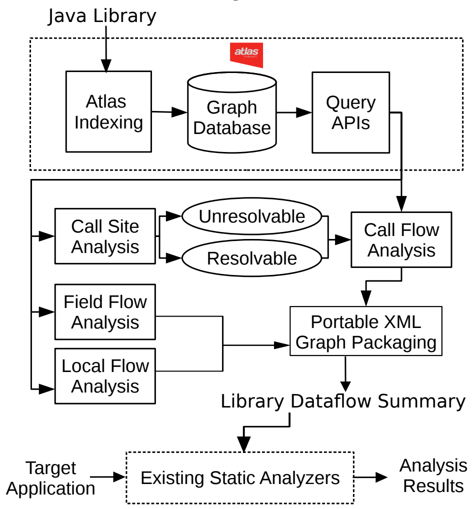

Fig. 4: Architecture of FlowMiner

It is worth noting two important features of our summary schema. *First*, our summaries pertain only to dataflow. While a 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. *Second*, 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.

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 2, 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 5 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.

6 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 1[5]. 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 1.

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. [11] 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,

[5] 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.

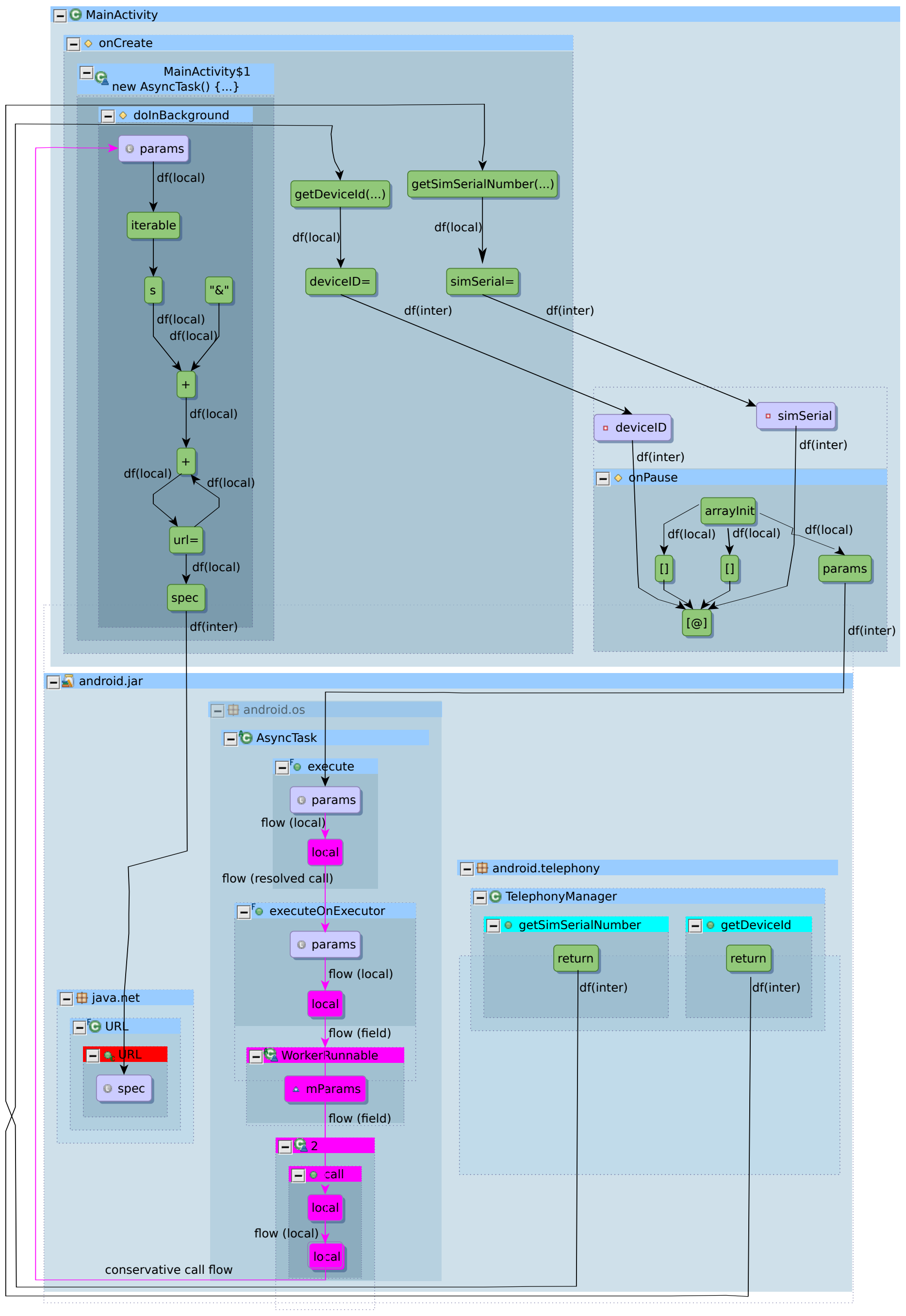

Fig. 5: Partial program analysis of malicious app from Listing 1.1 *with* FLOWMINER *summaries of Android*

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 [9] can also be 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 6 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 K^N false positive flows!

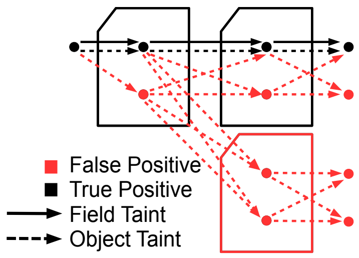

Fig. 6: Coarse flow specifications taint entire objects rather than fields, leading to false positives.

Table 1 shows the number of data-flow edges induced in the summary by FlowMiner in column 6 (fine-grained approach that tracks data-flows at field level granularity), which is about 8% of that induced by the coarse-grained approach that tracks data-flows at object level granularity (shown in column 7). 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 FlowMiner is *sound* in the following sense – each flow preserved in FlowMiner's summary is *actually possible* at runtime in the context of some application. 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 3 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 2.1).

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

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

Table 1: Experimental results showing the compactness and accuracy of FlowMiner on recent versions of Android. $|V|$, $|E|$ denote the number of nodes and edges in the original program graph; $|V|^S$, $|E|^S$ denote the same for the summary graph. Column 6 represents the number of data-flow edges in FlowMiner's summary that tracks flows at field level granularity, and column 7 shows the corresponding number of edges when flows are tracked at the object level granularity (* Percentage of object-granularity flows that are avoided due to the field-sensitive flow summarization performed by FlowMiner)

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 shown in Table 1, $G^S(\mathfrak{P})$ produced by FlowMiner 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 significant savings versus a fully-detailed program graph of a library, and yet retain the critical details for use in a partial-program data-flow analysis.

Scalability. We tested FlowMiner's scalability on the Android framework. For example, Android 4.4.4 (KitKat) contains roughly 2 million 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.

7 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 [15,25,24,4], tracking of non-trivial calling relationships between application and the library (e.g., asynchronous callbacks in Android) [9] and visualization of control flows from the application to the library and vice-versa [16].

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 [1], 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 [8] as a way to compactly represent the inter-procedural call and data flow semantics of the whole program. Rountev et al. [21] 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 [23].

Similarly to Rountev et al., Chatterjee et al.[10] summarize 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. [22] 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 [18] which is used by FlowDroid [6]) 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. Our summaries differ from that computed by Rountev et al. in that when mining inter-procedural flows, we resolve calls to a member method within an anonymous type to single target, because such calls can only have one possible runtime target, whereas Rountev et al. do not consider such calls to be monomorphic. Malicious applications often use custom anonymous classes to camouflage malicious behavior, and hence our approach of resolving calls to methods in anonymous classes to a single target is particularly useful to a security analyst or a subsequent analysis detect malicious flows by presenting more accurate and precise flow information. Secondly, the scalability of our approach has been validated on the Android framework, which is significantly larger in size (of the nodes and edges in the original and summary graphs) compared to the Java libraries evaluated by Rountev et al.

Some approaches summarize a software component independently of its callers and callees. For example, AVERROES[5] generates a placeholder that over-approximates the behavior of a given library. Their over-approximation may be too coarse to be useful in malware detection scenarios 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. [11], 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.

8 Conclusion

We presented FlowMiner [12], a novel solution that uses static analysis techniques to automatically generate an expressive, fine-grained summary of a Java library that is particularly useful for accurate detection of malicious data-flows in 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 demonstrated how FlowMiner's summary can be used in the malware analysis of an Android app. Validation of FlowMiner on recent versions of Android show that our summaries of are *significantly smaller than the original library*, yet more expressive and accurate than other state-of-the-art techniques. In the future, we plan to summarize Java libraries other than Android, and study the impact of using our summaries on specific data-flow analyses for malware detection.

Acknowledgements

We would like to thank the talented team at EnSoft Corp who developed the Atlas platform, including Jeremias Sauceda, Jon Matthews, and Nikhil Ranade; and ISU APAC team members who contributed to the malware detection tooling.

References

1. Automated program analysis for cybersecurity (apac) (July 2011), `https://www.fbo.gov/index?s=opportunity&mode=form&id=a14e4533c2a44c3288b6a29fa6fc5841&tab=core&_cview=1`
2. Android 4.4.4 (kitkat) (May 2015), `http://www.android.com/versions/kit-kat-4-4/`
3. Extensible common software graph (March 2015), `http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph`
4. Ali, K., Lhoták, O.: Application-only call graph construction. In: ECOOP 2012–Object-Oriented Programming, pp. 688–712. Springer (2012)
5. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings. pp. 378–400 (2013)
6. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. SIGPLAN Not. 49(6), 259–269 (2014)

7. Burnette, E.: Hello, Android: introducing Google's mobile development platform. Pragmatic Bookshelf (2009)
8. Callahan, D.: The program summary graph and flow-sensitive interprocedual data flow analysis, vol. 23. ACM (1988)
9. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: 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)
10. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: ACM Symposium on Principles of Programming Languages. pp. 133–146. ACM (1999)
11. Clapp, L., Anand, S., Aiken, A.: Modelgen: Mining explicit information flow specifications from concrete executions. In: International Symposium on Software Testing and Analysis. pp. 129–140. ACM (2015)
12. Deering, T.: (April 2015), `http://powerofpi.github.io/FlowMiner/`
13. Deering, T., Kothari, S., Sauceda, J., Mathews, J.: Atlas: A new way to explore software, build analysis tools. In: Companion Proc. of the International Conference on Software Engineering. pp. 588–591. ACM (2014)
14. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. pp. 3–14. SPSM '11, ACM (2011)
15. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. on Prog. Lang. and Sys. (TOPLAS) 23(6), 685–746 (2001)
16. LaToza, T., Myers, B.: Visualizing call graphs. In: Visual Languages and Human-Centric Computing (VL/HCC), Symposium on. pp. 117–124. IEEE (2011)
17. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer security applications conference. pp. 421–430. IEEE (2007)
18. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61. ACM (1995)
19. Rogers, R., Lombardo, J., Mednieks, Z., Meike, B.: Android application development: Programming with the Google SDK. O'Reilly Media, Inc. (2009)
20. Rosen, S., Qian, Z., Mao, Z.M.: Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In: Proc. of the ACM conference on Data and application security and privacy. pp. 221–232. ACM (2013)
21. Rountev, A., Kagan, S., Marlowe, T.J.: 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. pp. 2–16 (2006)
22. Rountev, A., Sharp, M., Xu, G.: Ide dataflow analysis in the presence of large object-oriented libraries. In: Hendren, L. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 4959, pp. 53–68. Springer Berlin Heidelberg (2008)
23. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. Program flow analysis: Theory and applications pp. 189–234 (1981)
24. Yan, D., Xu, G., Rountev, A.: 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. pp. 9–14. ACM (2012)
25. Zhang, W., Ryder, B.: 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. pp. 63–74. IEEE (2006)
26. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 95–109. IEEE (2012)