Taking Android App Vetting to the Next Level with Path-sensitive Value Analysis

Michael Backes, Sven Bugiel
Erik Derr and Christian Hammer

Technischer Bericht Nr. A/02/2014

Taking Android App Vetting to the Next Level with Path-sensitive Value Analysis

Michael Backes
Saarland University, CISPA, MPI-SWS
Saarbrücken, Germany
backes@mpi-sws.org

Sven Bugiel, Erik Derr, Christian Hammer
Saarland University, CISPA
Saarbrücken, Germany
{bugiel,derr,hammer}@cs.uni-saarland.de

***Abstract*—Application vetting at app stores and market places is the first line of defense to protect mobile end-users from malware, spyware, and immoderately curious apps. However, the lack of a highly precise yet large-scaling static analysis has forced market operators to resort to less reliable and only small-scaling dynamic or even manual analysis techniques.**

In this paper, we present Bati, an analysis framework specifically tailored to perform highly precise static analysis of Android apps. Building on established static analysis frameworks for Java, we solve two important challenges to reach this goal: First, we extend this ground work with an Android application lifecycle model that includes the asynchronous communication of multi-threading. Second, we introduce a novel value analysis algorithm that builds on control-flow ordered backwards slicing and techniques from partial and symbolic evaluation. As a result, Bati is the first context-, flow-, object-, and path-sensitive analysis framework for Android apps and improves the status-quo for static analysis on Android. In particular, we empirically demonstrate the benefits of Bati in dissecting Android malware by statically detecting behavior that previously required manual reverse engineering. Noticeably, in contrast to the common conjecture about path-sensitive analyses, our evaluation of 19,700 apps from Google Play shows that highly precise path-sensitive value analysis of Android apps is possible in a reasonable amount of time and is hence amenable for large-scale vetting processes.

I. Introduction

Modern smartphones are used for a wide range of functions that store and access highly sensitive personal data, like online banking or social networking. Together with the ubiquity of these devices, smartphones have become an interesting target for malware developers and overly curious apps. Due to its popularity, the Android platform is at the heart of this trend [1], [2], [3], [4].

Since a growing number of apps behaves maliciously or handles private data inappropriately, application vetting has become imperative for market operators. While Google's developer policy is explicit about the handling of sensitive user information, it is hard to verify the intended security policies in an automated way. This situation is aggravated by numerous third party (ad) libraries that immoderately report user information to their networks or even exhibit malware-like behavior [5].

Due to the rapidly increasing number of published apps, precise and scalable analysis tools are required for an automated large-scale application vetting. Approaches based on static analysis are widely accepted as well-suited tools for precise, in-depth program analysis. In contrast to dynamic approaches, static program analysis can guarantee properties about any program run, irrespective of the particular input or the state of the environment the app executes in. Although static analysis is more suitable for large-scale analysis in theory, it comes with certain challenges, which when not considered, impede its acceptance in practice: First, any non-trivial static analysis is inherently undecidable [6]. As a remedy, static analyses resort to over-approximations, which means that if the analysis cannot guarantee a certain property (like the absence of malicious behavior) it returns a "do not know" result. That result signifies that either the property could be violated or that the analysis is too imprecise to prove it, which is called a *false positive*. Second, dynamic language features like reflection, dynamic code loading, and Android's inter-component communication, are notoriously hard to handle for static approaches as they depend on runtime values – strings or primitive values – that may not be available. Moreover, the complex Android application model complicates matters even more. Approaches that approximate this model will miss essential parts of the program (*false negatives*).

Recent scientific research on Android has thus proposed numerous techniques to approach these challenges. However, due to the overall complexity most works only consider a subset of them [7], [8], [9], [10], [11], which yields unsound results. Others leverage heuristics that handle the simplest and most common cases but capitulate to more involved instances [12], [13], [14]. Even state-of-the-art static analysis tools on Android, such as FlowDroid [15], aim at very specific subproblems of app vetting. Their forward taint-tracking approach is tailored to data leak detection and inherently lacks the necessary properties to approach challenges like dynamic language features that require a precise path-sensitive value analysis.

To cover the previously described challenges, this work proposes a novel, holistic approach based on path-sensitive value analysis. Particulary, we introduce Bati, a static analysis framework for Android applications that is the first to provide path-sensitive analysis on Android. Bati builds on top of the established Java analysis frameworks

Wala[1]/Joana [16], [17]. We integrated Android-specific adaptions such as multiple program entry points, the Android application lifecycle, and asynchronously executing app components to these frameworks. Additionally, we solved the technically involved challenge of extending these existing systems with a scalable, path-sensitive program analysis that reasons accurately about information flow in general and strings passed as parameters to dynamic language features. To counter a path explosion during analysis—a possible implication of path-sensivity—Bati abtracts from loops by making the number of iterations explicit. Strings and primitive values are resolved in Bati by using techniques from partial and symbolic evaluation. Due to undecidability, but also to support a human analyst, we leave API calls symbolic in our analysis. Domain knowledge (cf. Section IV-D) and human intervention in a semi-automatic vetting process can resolve these symbolic values further if necessary. As a result, we receive a very precise information flow analysis that surpasses the state-of-the-art in Android.

In summary, we make the following contributions:

- We propose the first context-, flow-, object-, and path-sensitive analysis for Android that models the complex Android lifecycle correctly, including the asynchronous communication of multi-threading.
- We propose a novel value analysis to precisely determine primitive values and string parameters by leveraging a combination of backward slicing and techniques from partial and symbolic evaluation.
- We evaluate our analysis using a number of external and newly created benchmarks. Our evaluation reveals that none of the previous tools for Android takes the full intricacies of Android into account and finds as many illicit information flows as ours. At the same time our tool displays a very low number of false positives.
- We perform a set of experiments on 19,700 apps from Google Play and conduct a comprehensive study on an Android malware repository (from the Malware Genome Project [18]) where Bati finds a number of malicious practices that have only been described in virus databases after manual code reviews or not at all. As expected, we reaffirm well-known illicit behavior that other tools had determined previously.

Outline: The remainder of this paper is structured as follows. We start in Section II by motivating our development of a novel path-sensitive value analysis framework for Android applications. We continue in Section III with elaborating on how we model the complex Android application lifecycle and in particular how we advance and complement the state-of-the-art lifecycle models with a precise model of threading. Afterwards, we present in Section IV the Bati architecture and our algorithm for path-sensitive value analysis. We present the results of our evaluation of Bati in Section V. In Section VI, we compare Bati to closest related work in the areas of static and dynamic analysis and we discuss our solution in Section VII. We conclude this paper in Section VIII.

[1]http://wala.sf.net

```
1  public class MainActivity extends Activity {
2
3    protected void onCreate(Bundle savedInstanceState) {
4      super.onCreate(savedInstanceState);
5      setContentView(R.layout.premium_sms);
6      String frag1 = "10";
7      int frag2 = 66953930;
8      if (Math.random() > 0.5d) {
9        frag1 = "106618";
10       frag2 = getFragment();
11     }
12     SmsManager sm = SmsManager.getDefault();
13     sm.sendTextMessage(frag1 + frag2, null, "95pAHD", null, null);
14   }
15
16   private int getFragment() {
17     return 5829;
18   }
19 }
```

Listing 1: Premium SMS example

II. Motivation

We start by making a case for path-sensitive analysis using common, yet unsolved, classes of challenges for static analysis on Android. We illustrate the need for a precise value analysis and point out why identifying only the instructions that might be influenced by a source of sensitive information or that influence outgoing data is insufficient for reasoning about information flows in programs. Further, we demonstrate the importance of a precise model for Android's lifecycle.

A major challenge for static analysis is that the behavior of many static and dynamic language features strongly depends on their actual input. Often this is a string value like the class name used for reflection or the receiver number of an SMS. For instance, in malware analysis one needs to discriminate premium SMS from standard SMS to classify app behavior correctly. Existing approaches frequently fail in this classification for one or more reasons, even though discriminating these two cases is only a matter of analyzing the receiver number and the message body. Listing 1 shows an abstracted premium SMS example that is common for many Android SMS malware variants [18]. In line 13, a premium SMS with activation code *95pAHD* is sent (without user interaction) once the *MainActivity* is displayed. The receiver is selected randomly and assembled via string concatenation, a simple form of obfuscation. Although simple, this technique is effective against current string analyses that simply search for string fragments and combine them in arbitrary order to find meaningful combinations. These approaches will miss *frag2*, an integer variable that is implicitly converted to a string during concatenation (which internally constitutes a `StringBuilder.append` call). Moreover, intra-procedural String analyses miss that the number *5829*, which is returned by the method call `getFragment`, is also a potential fragment, as it again is implicitly converted to a string. However, even having identified all fragments does not suffice to handle this case precisely. The analysis

```
1  public class DataLeakage extends Activity {
2
3    private String deviceId;
4
5    protected void onCreate(Bundle savedInstanceState) {
6      super.onCreate(savedInstanceState);
7      setContentView(R.layout.data_leakage);
8      new ATask().execute();
9    }
10
11   protected void onPause() {
12     super.onPause();
13     deviceId = "fakeId";
14   }
15
16   // Button.onClick callback handler defined in XML file
17   public void leakDeviceId(View view) throws IOException {
18       File extDir = Environment.getExternalStorageDirectory();
19       FileWriter writer = new FileWriter(extDir);
20       writer.write("Device ID: " + deviceId);
21       writer.close();
22   }
23
24   private class ATask extends AsyncTask<Void,Void,String>{
25     protected String doInBackground(Void... params) {
26         TelephonyManager tm = (TelephonyManager)
27             getSystemService(Context.TELEPHONY_SERVICE);
28         return tm.getDeviceId();
29   }
30
31   protected void onPostExecute(String result) {
32       deviceId = result; }
33   }
34 }
```

Listing 2: Android lifecycle example

would additionally require the information about which fragments are possible on which path. Enumerating all possible combinations results in many false positives. Thus, path-sensivity is crucial for being able to assemble the number *1066953930* and the number *1066185829*, which will overwrite the former within the if-statement.

Another, only partially solved, challenge is that static Android analysis requires a precise execution model that is tailored to the complex Android environment. Listing 2 shows an abstracted real-world example in which the device identifier is eventually leaked. If the user clicks on a specific button, `leakDeviceId` is executed to write the content of *deviceId* to the SD card where it can be read by any application. An analysis that does not parse the app's metadata will fail in detecting this method as an event listener that is implicitly called by the framework. Thus, a code reachability analysis will flag this method as dead code and the `write` instruction in line 20 will remain undetected.

Moreover, an accurate model of the Android lifecycle is required to find field updates in the framework callback method `onCreate` and the AsyncTask callback `onPostExecute`. Algorithms that approximate the AsyncTask lifecycle will not report that the actual device ID (line 28) is accessed in a dedicated thread and that it is only leaked if the user clicks the button before the app is paused (for the first time).

III. Modeling the Android Lifecycle

Android apps adhere to a complex application lifecycle that challenges static analysis algorithms. Approximations to this model or even disregarding certain parts like multi-threading leads to incomplete data models. Code reachability analysis on these models will then fail to detect large parts of the application. This implies that sensitive data leaks or malicious code segments are missed by the actual analysis. To account for this problem we adopt state-of-the-art modeling of app components and event listeners. We add a precise model of threading, which has previously been disregarded by related work [13], [8], [11], [15], to receive a comprehensive lifecycle model.

In the following, we will elaborate on the Android application model, focus on its characteristic features, and explain how Bati creates a tailored lifecycle model for each app to be analyzed.

A. Asynchronously executing components

Every Android application consists of several components. There are four main component classes: *Activities* are single user-interface screens, *Services* are long running background tasks that do not interact with the user, *ContentProviders* provide a read and write interface to structured data, and *BroadcastReceivers* listen for global events that are broadcast via *Intent* messages. Every application declares its components in a manifest file. Components execute asynchronously and are triggered by events or launched (and stopped) due to user interaction. To account for this complex event-driven application model, Bati allows components to be executed in arbitrary order.

Android apps do not have a single entry point, as, e.g., regular Java programs do. Instead, each component defines callback methods, which are implicitly invoked by the system during the app's lifecycle. Developers can override callback methods such as `onCreate` or `onPause` (cf. Listing 2) to initialize data structures, save state before an app is switched into the background, or closed. Each class of components has its own lifecycle specifying the potential order of callbacks. Bati generates a lifecycle method per application component that models the framework's runtime behavior in accordance with the official specification[2].

B. Event Listeners

Interaction of apps with the application framework is, to a large extent, event-driven. For instance, apps (that hold the corresponding permission) can register callback objects as listeners to the location service, which will inform them whenever the device's location has changed.

In order to register callback objects, developers have two options: They can either declare callbacks in XML files that specify an activity's layout to render its user interface; or they can register callbacks programmatically via a well-defined API, or by overriding callback methods of system classes.

To detect callback methods (and their enclosing classes) Bati includes an Android XML file parser that analyzes layout definitions. As shown in Listing 2 it is common to register `onClick` handlers for buttons that way.

[2] http://developer.android.com/guide/components

Identifying programmatically registered callbacks requires iterative code scanning. We first perform a reachability analysis starting from the callback methods identified so far. We then scan each reachable class for overridden system callbacks and method calls that use system interfaces as input arguments. We iterate this process until we reach a fixed point to account for nested methods that register callbacks. All identified event listeners are incorporated into the lifecycle method of their enclosing component. As their execution order cannot be predicted in general, Bati assumes an arbitrary order.

C. Threading

Android's application model imposes very strict response times onto application components and forces developers to off-load potentially long-running code into separate threads. Apart from Java's default packages like `java.util.concurrent`, Android provides a widely adopted API for `AsyncTasks` to support the developer in separating such code into background tasks (e.g., downloading a file from the Internet) and feeding results back to the originating thread (e.g., as a progress monitor).

In contrast to Java's Thread class that contains a single thread entry method (`run`), AsyncTask features a series of callbacks that are discharged in a specific order once its `execute` command is invoked. If a task is triggered, the `onPreExecute` method is executed to set the task up. Subsequently, the `doInBackground` method is executed in a background thread to realize the main functionality. This method takes a *varargs* argument, (i.e. basically syntactic sugar for an array). Its return value is passed to the callback methods `onPostExecute` or `onCancel`, depending on whether or not the task was cancelled. Within the `doInBackground` method, the method `publishProgress` can be invoked with a varargs argument which in turn triggers the callback `onProgressUpdate` with the same argument. Bati automatically identifies the concrete types for the generic parameter types like `<Void,Void,String>` in Listing 2 and generates a tailored lifecycle method that reflects this behavior. During analysis, this lifecycle method is then invoked instead of `execute`.

The callback array arguments represent another challenge for static analysis. Section IV-D explains in detail how Bati resolves them as part of the value analysis. To the best of our knowledge, we are the first to generate a comprehensive model of threading in Android, including the complex AsyncTask lifecycle with precise parameter passing. As the prevalent design pattern for multi-threading in Android apps, it is essential for static analysis algorithms to handle AsyncTasks accurately.

D. Synthetic Main Method

Finally, like previous approaches [15], we generate a synthetic main method, that constitutes a single, unique entry point of the application. It models the Android runtime behavior and forms the basis for Wala/Joana to create the precise data model Bati builds on. In this method each component is instantiated by invoking its constructor, and subsequently its respective lifecycle method is called.

To account for the asynchronous nature of components it is important that the control-flow model matches the runtime behavior. *ContentProviders* are the first components created at application launch time.[3] Subsequently, custom application classes follow in the order provided by the class hierarchy. Finally, Bati assumes that all other components can execute in arbitrary order including repetition.

IV. Path-sensitive Value Analysis

Bati builds upon the established information flow control framework Joana [16], [17]. The frontend of Joana is based on the widespread Java analysis framework Wala that comes with an intermediate-representation (IR) in static-single assignment (SSA) form [19], [20], a precise points-to analysis framework, and various dataflow solvers.

The high-level architecture of Bati is depicted in Figure 1. It is structured into two main phases, a pre-processing phase and an analysis phase. In the pre-processing phase the Android application file is parsed and the application lifecycle is modelled. The output, a lifecycle-enhanced bytecode file, is subsequently used during the multi-layered analysis phase that works in a nutshell as follows: First, Joana's SDG (system-dependence graph) builder module creates a precise data model. This model is then used by our slicer to analyze framework methods specified as point of interest (sink). Path-sensivity is added in a follow-up step by the sink tree builder. Finally, a minimal, expressive string representation of the result is rendered by our novel path merging algorithm that adopts techniques from partial and symbolic evaluation. In the following, we elobarate on the details of the individual modules.

A. Pre-processing phase

In order to be able to analyze Android applications, the information included in the application package (*apk*) needs to be parsed. Although Joana provides a strong foundation for static analysis, it is designed with Java programs in mind. It adopts the Dalvik frontend from the SCanDroid project [13] to transform Android bytecode directly into Wala's IR but it lacks a model of the Android lifecycle. Thus, we add modules for parsing Android application manifests and layout files used by Activities. The manifest file is the starting point for our Android lifecycle analysis as it contains meta data like the requested permissions or the components the application is composed of. In combination with the event listeners declared in layout files this allows us to initiate the application lifecycle modeling as described in Section III.

The final model is then passed to our lifecycle generator that alters the original bytecode such that the subsequent analysis faithfully takes the Android peculiarities into account.[4] To that end, we implemented a bytecode instrumentation framework based on the popular dexlib[5] library that generates bytecode according to the lifecycle model.

[3] http://developer.android.com/reference/android/content/ContentProvider.html

[4] Technical reasons exclude direct altering of Wala's IR

[5] https://code.google.com/p/smali/

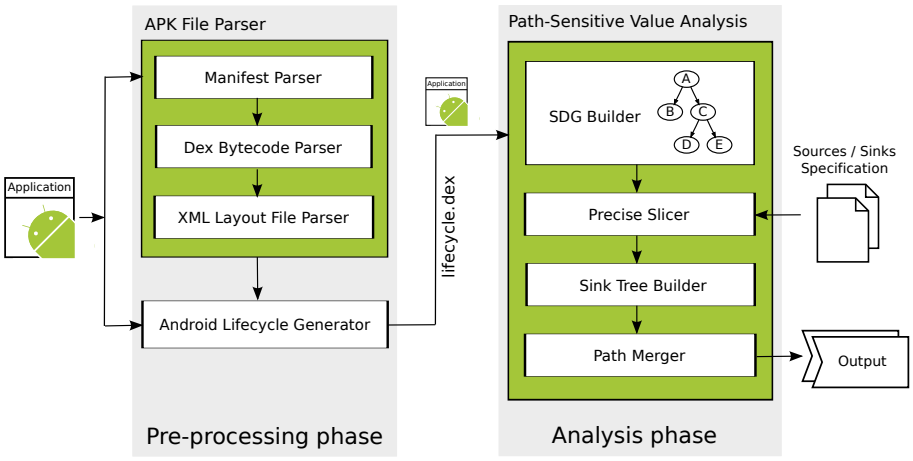

FIGURE 1: HIGH-LEVEL ARCHITECTURE

The lifecycle-enhanced bytecode is used as input for Joana's backend that comprises object-, context-, field-, and flow-sensitive system dependence graphs for inter-procedural program analysis. An SDG is a language-independent representation of dependencies between statements of a program. In the process of generating the SDG, dynamic method dispatches are resolved and method side-effects are computed. The SDG includes the lifecycle model tailored to the analyzed app and forms the basis for our slicing algorithm which we describe in the following.

B. Path-tracking slicing algorithm

Traditional SDG-based slicing approaches [21] compute a slice by traversing the underlying data representation backwards starting at some sink. Data-dependent statements are iteratively added until a fixed point is reached. The resulting summary slice is an unordered set of statements that influences the sink but does not contain any path information. As illustrated by Listing 1 such path information is essential to make a precise value analysis possible.

However, building a fully path-sensitive slicer, i.e. considering all possible execution paths, is impossible in practice as computing loop conditions statically is undecidable. To account for this problem, we devised a *path-tracking slicing algorithm* that works as follows: The slicer traverses app instructions in the reverse order of runtime execution (reverse control-flow order) while exploiting data-dependency information of the SDG to only account for instructions that influence arguments of the sink. During branch traversal path information derived from the SDG and the SSA-based IR is added to the slice that allows recreating individual paths in a subsequent step.

The following kinds of path information is used by BATI:

- **Phi instructions**. These functions are an essential building block for Wala's SSA-based IR. A program (Listing 3) transformed into SSA form (Listing 4) offers flow-sensitivity for local variables because every variable is statically only assigned to once. They are located at intra-procedural control-flow merge points after conditionals and switch statements and exhibit information on how variables are modified across paths.
- **Entry statements** are added to the slice if an argument depends on a method parameter. Every caller of this method represents a distinct path.
- **Field setter instructions**. Static and instance fields are not tracked by the method-local phi instructions. If their content is altered in different paths, multiple values might be possible when the field is accessed.

```
1 x ← 23
2 if(c):
3     x ← 42
4 y ← x − 10
5
```

LISTING 3: NON-SSA FORM

```
1 x1 ← 23
2 if(c):
3     x2 ← 42
4 x3 ← φ(x1, x2)
5 y ← x3 − 10
```

LISTING 4: SSA FORM

Output 1 shows the resulting slice output of the *DataLeakage* example in Listing 2 in a backward control-flow order. Beginning at the `write` method (line 1), the slicer iteratively resolved parameters and objects that influence this call. The string written to the SD card is assembled via a series of invocations on a `StringBuilder` object (lines 5-8). Its value depends on the field *deviceId* that can have two values depending on the actual state of the application. The field is either updated with the constant string *fakeId* (line 10) or with the actual device id that is accessed (line 17) in a new instance of the `AsyncTask` class *ATask*.

Abstracting framework calls: BATI treats Android framework invocations like `android.telephony.TelephonyManager->getDeviceId()` symbolically, i.e. we include only a String representation of the method call in the slice. Since framework methods have a well-defined semantics this usually suffices to understand the program behavior. This abstraction dramatically lowers the complexity of the underlying data structures

```
1  java.io.FileWriter−>write{v6}(v16)
2  java.io.FileWriter−><init>{v6}(v4)
3  v6 = new java.io.FileWriter
4  v4 = android.os.Environment−>getExternalStorageDirectory()
5  v16 = java.lang.StringBuilder−>toString{v13}()
6  v13 = java.lang.StringBuilder−>append{v8}(v11)
7  java.lang.StringBuilder−><init>{v8}("Device ID: ")
8  v8 = new java.lang.StringBuilder
9  v11 = com.example.DataLeakage{this}.deviceId
10 com.example.DataLeakage{this}.deviceId = "fakeId"
11 com.example.DataLeakage{this}.deviceId = p1
12 ENTRY
13   com.example.DataLeakage$ATask.onPostExecute(Ljava/lang/String;)V
14 com.example.DataLeakage$ATask.onPostExecute{this}(v3)
15 v3 = com.example.DataLeakage$ATask.doInBackground{this}()
16 return v12
17 v12 = android.telephony.TelephonyManager−>getDeviceId{v8}()
18 v8 = com.example.DataLeakage−>getSystemService{v2}("phone")
19 v2 = com.example.DataLeakage$ATask{this}.this$0
```

Output 1: Slice of DataLeakage example

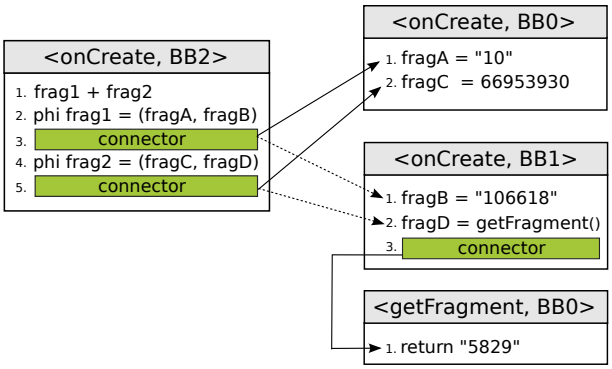

Figure 2: Sink tree for Listing 1

as no framework code has to be considered during SDG construction and results in a more compact SDG.

C. Recreating paths

In this step, we exploit the path-information included in the slice to discriminate individual paths reaching a sink. However, considering all possible execution paths, is impossible in practice as computing loop conditions statically is undecidable. To account for this problem, Bati assumes that loops and recursive methods are either executed once or not executed at all. While this abstraction seems to restrict the analysis at first glance, it does not have a large impact on the precision in practice since processing the loop/method body once generally suffices to approximate code behavior.

Bati transforms the slice into a tree representation (*sink tree*) using the collected path information. All instructions within the slice carry supplemental information about the control-flow-graph of their enclosing method and the basic block (BB) in which they are located. Tree nodes are generated for each unique basic block that contains at least one instruction of the slice. Figure 2 shows the sink tree generated for Listing 1 with four nodes.

A specific feature of our tree model is that instructions across nodes are connected rather than nodes themselves. If consecutive instructions within the slice reside in different nodes, a special connector is created to connect these instructions by means of a directed edge. For phi-instructions that reside in control-flow merge points, the connector creates outgoing edges for any successor instruction, e.g. the connector after the `phi frag1` instruction points to the assignment instructions of `fragA` and `fragB`.

This complex model is required since the smallest unit of resolution is not an instruction but arguments within an instruction. The receiver number in Listing 1 is assembled via the string fragments `frag1` and `frag2`. `frag1` depends on assignments that reside in basic block `BB0` and `BB1` of method `onCreate` which yields two paths. `frag2` is assigned in `BB0` of `onCreate` and `getFragments`, which again yields two paths. A naïve path extraction would produce all combinations, in this case four distinct paths. To avoid such over-approximation, the extraction algorithm has to detect that there are only two distinct paths reaching the sink. Technically, we solve this problem by enforcing that the path extraction algorithm traverses the same outgoing edge on connectors of the same node (either the solid or dashed arrows in `BB2`).

Subsequent traversal of the tree yields the individual paths that reach the sink, which in turn are control-flow ordered lists of instructions. Outgoing edges of connectors are traversed before its immediate successor within the same node (if any). Within a node the algorithm stops if either the end of the instruction list or an instruction that is pointed to from a different node is reached. Applying this algorithm to Listing 1 yields two paths that contain the instructions to reconstruct the correct receiver numbers *1066953930* and *1066185829*.

D. Path merger

In the following, we present a path merging algorithm that allows precise reasoning about values flowing into a sink. Having a slice for a single path allows aggressive optimization by using techniques of partial evaluation and Android domain knowledge. This allows recreating method arguments as minimal, expressive, and precise string representations.

Evaluating unary/binary operations: As a first means of shrinking the slice we statically evaluate unary and binary operations. This is a crucial requirement for computing indices used for array and list accesses/updates. We adopt partial evaluation techniques from the area of compiler construction to evaluate such operations if the operand values are constants or can be merged into a constant of a primitive type, e.g., an integer addition $17 + 23$ is evaluated to 40. If there is at least one non-constant operand, for instance, a symbolic framework call to `Math.round()` that returns an integer value, the algorithm outputs a string representation of this unary/binary operation, e.g., 17 + Math.round(.7f).

Array access resolution: Precise analysis of array access is a quite challenging task. Related work based on static analysis [7], [8], [10], [15] usually overestimates array

modifications, as they do account for array modifications across paths. Moreover, without a value evaluation positions within an array cannot be tracked statically. This results in false positives when sensitive data is written at position `x` but later position `x'` is accessed.

As one of the prime data structures in programming, not being able to handle array access in a static way lowers the precision of the analysis dramatically (see AsyncTask parameter passing in Section III-C). Having an ordered set of array updating instructions for a single path allows an accurate resolution of an array access. BATI statically reconstructs the content at the time of the access and uses expression evaluation to compute the access index. Depending on the way the array has been updated and accessed along the path, there are essentially three outcomes possible:

1) BATI can exactly determine the content that is accessed, iff the access index i is statically computable and it is possible to unambiguously determine the data at position i. We then discard all array instructions and return the reassembled value at position i.
2) The index is computable but this position can hold different data at the time of the access. This may happen if the array is both updated with statically computable and non-computable indexes. In this case, the algorithm returns a list of possible candidates.
3) In the worst case scenario, the index is not statically computable. The algorithm then returns a template consisting of how the index is computed and a string representation of the reconstructed array. In this case this is notably the best one can do with static analysis.

Output 2 shows the part of the slice relevant for resolving the array testcase `ArrayAccess2` of the open-source testsuite DroidBench (see Section V). This testcase checks whether the analysis overestimates array operations. An array is filled with both sensitive (device Id, line 3-4) and non-sensitive data (*"no taint"*, line 2), whereas the latter one is finally accessed. Algorithms that cannot distinguish between different positions and that are not able to statically compute the index from a series of operations will erroneously report a leak. In contrast, BATI improves over this by statically resolving the array access. Given the array register `v8` and the instructions that update the array (line 2-3), the content is statically reconstructed as follows:

Step	Instruction	Reconstructed Array
0	∅	[ ]
1	v8["4"] = "no taint"	[x,x,x,x,"no taint"]
2	v8["5"] = v16	[x,x,x,x,"no taint",v16]

The access index `v25` is computed by a series of expressions within the `calculateIndex` method (line 6-10). The expression evaluator starts with the return statement and iteratively assembles and solves the expression

```
1  v27 = v8[v25]
2  v8["4"] = "no taint"
3  v8["5"] = v16
4  v16 = android.telephony.TelephonyManager->getDeviceId{v11}()
5  v25 = de.ecspride.ArrayAccess2->calculateIndex{this}()
6  return v10
7  v10 = v8 + "4 I"
8  v8 = v6 % "10 I"
9  v6 = v4 * "5 I"
10 v4 = "1" + "1 I"
```

OUTPUT 2: PARTIAL SLICE OF ARRAY TESTCASE `ArrayAccess2`

$(((1+1)*5)\%10)+4 = 4$. Finally, the non-sensitive value *"no taint"* at position four of the reconstruct array is returned.

Incorporating domain knowledge: BATI abstracts from framework internals by treating calls to framework methods symbolically. By using domain knowledge about the framework API it is possible to reduce the output size of the slice even further. Especially for strings, which are internally assembled via a series of `StringBuilder/StringBuffer` methods, it is important to return the concrete string value *"Hello World!"* rather than the instructions `java.lang.StringBuilder-><init>("Hello")` and `java.lang.StringBuilder->append(" World!")`. Concrete string values are not only more precise but they further allow automatic reasoning about dynamic language features like reflection or inter-component communication. We model this domain knowledge by adding semantic rules that work on a series of method calls on the same object. In case of the StringBuilder class we add a rule that concatenates the arguments and outputs the result.

Besides strings there are more classes for which specific domain knowledge increases the precision of the analysis, in particular Java collection classes including various kinds of List, Map, and Set implementations. These methods internally behave like arrays and provide convenience functions for the developer. We encode the getter/setter methods of these classes as domain knowledge to statically handle them similar to arrays. Output 3 shows part of a slice that creates a list object, subsequently adds sensitive and non-sensitive data and finally retrieves the first element. Without domain knowledge, the slice of Output 3 would be incorrectly flagged as sensitive data leak, as the algorithm cannot distinguish between different positions of the `LinkedList`. By adding semantic rules for the `add`, `get` and `init` methods, we can reconstruct the list content and output the string *"abc"* from the first position. This dramatically reduces the number of false positives, when sensitive data is stored in a collection but later on different data is accessed. In case the resolution fails, e.g. if the argument of the getter method is not statically computable or there is no semantic rule for a framework method, we conservatively return the original slice.

Generating call-chains: In scenarios where the output is to be reviewed by a human analyst, the slice is rendered as a series of call-chains. This post-processing technique improves the readability by generating compact string representations of a series of method calls on a framework class object. Instead of printing a list of instructions like in Output 3 the slice is merged as follows: Starting at the `get`

```
1 v34 = java.util.LinkedList->get{v7}("0")
2 v27 = java.util.LinkedList->add{v7}("def")
3 v23 = java.util.LinkedList->add{v7}(v20)
4 v12 = java.util.LinkedList->add{v7}("abc")
5 java.util.LinkedList-><init>{v7}()
6 v7 = new java.util.LinkedList
7 v20 = android.telephony.TelephonyManager->getDeviceId{v16}()
8 v16 = MyActivity->getSystemService{this}("phone")
```

Output 3: Slice containing list operations

invocation in line 1, the algorithm collects all invocations on the object *v7* (line 1-6), reorders them according to the runtime execution order and iteratively merges them. In particular, the class name is omitted as the object class does not change between two consecutive invocations. This results in a more compact representation that does not sacrifice information:

```
java.util.LinkedList-><init>()->add("abc")
  ->add($FNC)->add("def")->get("0")
```

In a subsequent step all non-constant arguments, here denoted with the placeholder `$FNC`, are processed the same way. This yields another call-chain:

```
MyActivity->getSystemService("phone")
  =>android.telephony.TelephonyManager->getDeviceId()
```

In this case, the object type changes, i.e. `getSystemService` returns an `Object` that is cast to a `TelephonyManager` object on which `getDeviceId` is called and thus we cannot omit the class name. To prevent nested call-chains, each individual chain is printed on an new line and placeholders (like `$FNC`) are used to connect these chains.

V. Evaluation

The implementation of Bati comprises approximate 9.7 kLOC. For the evaluation of our approach we configure our slicer with the comprehensive list of sources and sinks in the Android API reported by the *SuSi* project [22]. Data leaks are automatically reported, if the optimized path slices contain a privacy sensitive sink from that list. Moreover, Bati disregards sinks, if the application does not hold the required permissions (e.g. sinks in library code that probes for certain permissions of its host app). This information is derived by the Android API permission map by Porter Felt *et al.* [23], the parsed app permissions, and the *sharedUserId* information from the manifest that indicates a permission sharing across apps with the same signature.

To account for both large-scale analysis and specific analysis of a single app, Bati supports various levels of verboseness for the analysis result. Appendices A and B show the Bati output for the examples in Section II. By default, it prints – for every processed sink – the unique sources leaked, the number of paths reconstructed from the slice, and the minimal string representation of data reaching the sink. Optionally, the instructions per path and the complete slice including auxiliary information like the methods and basic blocks can be returned. This may be required to support reverse engineering of an app by a human analyst.

In the following, we present the results of several benchmarks and large-scale application studies to demonstrate the benefits of our analysis. All experiments are conducted on a test server with four Intel Xeon CPU E5-4650L @ 2.60GHz processors with 8 cores each and Hyperthreading, 768GB RAM, and SSD-based storage. For our large-scale application studies, we ran 64 parallel, *single-threaded* analyses, i.e. at most 64 apps were analyzed in parallel. We first run Bati on the open-source test suite DroidBench to test the precision of our approach and to compare it with state-of-the-art analysis tools. Under the assumption that the DroidBench test cases are complete and our precision holds for real-life apps as well, we use Bati to analysis 19,700 apps from the Google Play Store to show that our approach is indeed practical on large scale. Finally, we conduct a comprehensive study on an Android malware repository where Bati finds a number of malicious practices that have only been described in virus databases after manual code reviews or not at all.

A. DroidBench Test Suite

DroidBench is an open-source test suite[6] containing Android applications crafted to evaluate the quality of static and dynamic analysis tools. This set of apps contains a variety of challenges for analysis tools that have to be detected and correctly classified either as actual data leaks or as a false positive. DroidBench v1.0 contains a set of 39 applications. Table 1 shows the results as reported in [15] for the two commercial analysis tools IBM AppScan Source [24] and HP Fortify SCA [25] as well as for the state-of-the-art FlowDroid [15] framework in comparison to Bati's results. It lists the results for each testcase and the derived precision, recall, and F-measure values.

The commercial tools display weaknesses in the Android-specific test cases that test the Android lifecycle model and callback methods. Despite showing a reasonable precision, which is usually required by customers to improve adoption, they miss almost 50% of all leaks. FlowDroid significantly improves both precision and recall as a result of their enhanced lifecycle model. Even though, their limitations [15] result in false positives or missed leaks. In contrast, our value analysis yields a perfect precision and recall, which illustrates the improvement with respect to the state-of-the-art tools. Our analysis excels particularly in the category *arrays and lists* where traditional tools fail.

Although DroidBench already includes testcases for many Android and Java-specific design patterns, there are no testcases to account for Android's common asynchronous communication via threads. We therefore added another five testcases, which we are going to contribute to the test suite. These test whether the analysis handles Java threads and Android's `AsyncTask` class correctly.

Table 2 shows the results for Bati and FlowDroid of these test cases. The `AsyncTask` test cases both test for correct parameter passing and for correct modeling of the series of callback methods invoked by the framework when an `AsyncTask` is executed. The results show that FlowDroid does not model the lifecycle of `AsyncTasks`

[6] https://github.com/secure-software-engineering/DroidBench

⊛ = correct warning, ★ = false warning, ○ = missed warning
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	Fortify	FlowDroid	Bati
Arrays and Lists				
ArrayAccess1			★	
ArrayAccess2	★	★	★	
ListAccess1	★	★	★	
Callbacks				
AnonymousClass1	○	⊛	⊛	⊛
Button1	○	⊛	⊛	⊛
Button2	⊛ ○ ○	⊛ ○ ○	⊛ ⊛ ⊛ ★	⊛ ⊛ ⊛
LocationLeak1	○ ○	○ ○	⊛ ⊛	⊛ ⊛
LocationLeak2	○ ○	○ ○	⊛ ⊛	⊛ ⊛
MethodOverride1	⊛	⊛	⊛	⊛
Field and Object Sensitivity				
FieldSensitivity1				
FieldSensitivity2				
FieldSensitivity3	⊛	⊛	⊛	⊛
FieldSensitivity4	★			
InheritedObjects1	⊛	⊛	⊛	⊛
ObjectSensitivity1				
ObjectSensitivity2	★			
Inter-App Communication				
IntentSink1	⊛	⊛	○	⊛
IntentSink2	⊛	⊛	⊛	⊛
ActivityComm1	⊛	⊛	⊛	⊛
Lifecycle				
BroadcastRecvLifecycle1	⊛	⊛	⊛	⊛
ActivityLifecycle1	⊛	⊛	⊛	⊛
ActivityLifecycle2	○	⊛	⊛	⊛
ActivityLifecycle3	○	○	⊛	⊛
ActivityLifecycle4	○	⊛	⊛	⊛
ServiceLifecycle1	○	○	⊛	⊛
General Java				
Loop1	⊛	○	⊛	⊛
Loop2	⊛	○	⊛	⊛
SourceCodeSpecific1	⊛	⊛	⊛	⊛
StaticInitialization1	○	⊛	○	⊛
UnreachableCode1		★		
Miscellaneous Android-Specific				
PrivateDataLeak1	○	○	⊛	⊛
PrivateDataLeak2	⊛	⊛	⊛	⊛
DirectLeak1	⊛	⊛	⊛	⊛
InactiveActivity	★	★		
LogNoLeak				
Sum, Precision, and Recall				
⊛ , higher is better	14	17	26	28
★ , lower is better	5	4	4	0
○ , lower is better	14	11	2	0
Precision $p = ⊛/(⊛ + ★)$	74%	81%	86%	100%
Recall $r = ⊛/(⊛ + ○)$	50%	61%	93%	100%
F-measure $2pr\,/\,(p+r)$	0.60	0.70	0.89	1.00

TABLE 1: DROIDBENCH TEST RESULTS

App Name	FlowDroid	Bati
AsyncTasks & Threads		
AsyncTask1	★	
AsyncTask2	○ ○	⊛ ⊛
AsyncTask3	○	⊛
Threads1	○	⊛
Threads2	⊛	⊛
Sum, Precision, and Recall		
⊛ , higher is better	1	5
★ , lower is better	1	0
○ , lower is better	4	0
Precision $p = ⊛\,/\,(⊛ + ★)$	50%	100%
Recall $r = ⊛\,/\,(⊛ + ○)$	20%	100%
F-measure $2pr\,/\,(p+r)$	0.29	1.00

TABLE 2: ADDITIONAL THREAD TESTCASES

(*AsyncTask2+3*) and overestimates the array passed to the task in *AsyncTask1*. The thread test cases check whether an analysis detects data leaks within the `run` method of a thread. The thread is either implemented as anonymous inner class (*Thread1*) or as individual class (*Thread2*). Although both tools use an approximate thread execution model, i.e. they assume that threads execute in arbitrary but sequential order, the results differ strongly. The testcases revealed strong limitations in FlowDroid's thread model whereas BATI passes all test cases due to the correct `AsyncTask` model and the ability to resolve array accesses in a static way.

B. Google Play Store Apps

We conducted a large-scale analysis of 19,700 apps from the Google Play Store. The analysis results show that most of the apps—accidentally—leak sensitive data by including third-party libraries for advertising, billing, or tracking purposes. The advertising libraries strongly differ in the amount of data they require or silently probe for. Google's Mobile Ad library does not require user data in the default settings, only suggests the addition of location or gender data to deliver tailored ads. In contrast to this, BATI found that the Mobclix library accesses and sends, among others, eight unique location data properties. Together, these location properties allow the creation of an accurate movement profile of the user. We also found cases, in which equally precise location data was accessed but not leaked. For example, the `Evernote` and `Evernote widget` apps both use location data and the mobile country code to check whether the user's current location is in China, but they do not leak it. The reason is that Evernote runs a separate service called Yinxiang Biji[7] for Chinese users and thus ensures that the adequate service is selected by the client app.

BATI found a number of apps with third-party libraries that optionally send SMS, e.g. `Fortumo` (In-App Billing) or `Vserv` for displaying ads. Our results showed, that none of the analyzed apps makes use of this optional feature. We assume that app developers do not want to scare users off because of the required `SEND_SMS` permission. In contrast to BATI, heuristics-based algorithms would very likely flag this app erroneously as (potentially) malicious.

We also found apps like *com.dangnh* that contains code for sending premium SMS when the user clicks on a dialog button. BATI reported that the receiver number is read from the asset file *config.dat*. We manually extracted the number *09832014 32* from this file and could confirm that this number is registered at the Russian *New Telephone Company*. However, using the permission map and manifest information, BATI indicates that the app neither declares the required permission nor does it have the `shareUserId` flag. Thus, this code is never executed in practice. Besides detecting illicit behavior, BATI could verify its absence in the SMS code of *com.krospik.speechcloud*. It is a simple speech recognition tool that also allows to send SMS by means of dictating number and message. Our results showed that indeed only the unaltered user input flows into the SMS sink.

[7] http://blog.evernote.com/blog/2012/05/09/evernote-launches-separate-chinese-service

In contrast to the common conjecture about path-sensitive analysis, BATI has, despite its precision, a reasonable running time of about 26min on average per app and a worst case running time of 45min on highly complex apps like Facebook or WhatsApp.

C. Malware Genome Repository

In order to demonstrate the accuracy and reverse engineering capabilities of BATI, we analyzed a data set of known malware from the Malware Genome Project [18]. The data set includes 1260 malware samples of 50 different malware families. For this semi-automated analysis we extended the sink list of the SuSi project with numerous methods of framework classes to account for reflection and code execution. Typically, no sensitive data flows into methods of the classes `Runtime`, `System`, and `(Dex)ClassLoader`. However, knowing how they are used is paramount to reverse-engineer malware samples and supports deeper understanding of the program behavior. Furthermore, we additionally double-check our findings against public malware reports of known security and antivirus companies to confirm that the results of BATI are indeed correct.

Most of the malware from this repository is designed for surreptitiously stealing money (via premium SMS) and/or personal information. In the following, we present in detail the analysis results for a subset of the malware families and conclude with findings that are valid across most samples. The selected subset focuses on SMS malware families as they have not been covered by tools from related work (which concentrate on privacy leaks) and usually require manual analysis.

The `ADRD` malware family leaks unique identifiers like the device identifier (IMEI) and the subscriber ID (IMSI) via system log and SMS. At the time of publication most Android devices ran Android 4.0 or lower. In these versions the system logs are publicly readable by any app that declares the `READ_LOGS` permission. Writing sensitive user data to logs is therefore considered a severe privacy breach. The results of BATI confirm another interesting result from Symantec's malware report[8]: The malware issues crafted requests to Chinese search engines to increase the pagerank for certain sites. In contrast to the mentioned search engine *baidoo.com* we also detected destinations including *google.cn*.

Samples of the `BaseBridge` family try to impose costs on the user by silently sending premium SMS. The actual bytecode does not reveal any malicious behavior except that it loads and executes certain files from the assets. In particular it tries to root the device with a *rage against the cage* exploit. If it succeeds another file called *anserverb* is read in as *AppSMS.apk* and installed via Android's package installer to */data/data/xxx.apk*. We manually extracted the supplemental apk file and run our analysis to reveal its malicious behavior. The malware contacts a C&C server by sending unique identifier and the phone number. Then, it starts a background service, in which a ThreadPool is initialized to periodically send out new premium SMS.

[8]http://www.symantec.com/security_response/writeup.jsp?docid=2011-021514-4954-99&tabid=2

The `Beanbot` family implements a functionality very similar to the BaseBridge family, however, it does not hide its SMS sending code in a supplemental apk file. It is also controlled via a C&C server that initially retrieves the IMEI, IMSI and phone number. Subsequent control messages are sent via SMS. Premium SMS are sent back to the number the initial control SMS is received from.

`jSMSHider` is an SMS malware that especially targets Android users with custom ROM. The devices are already rooted and its supplemental apk, named *testnew.apk*, can be installed without the user's consent. The installation process is invoked via the console command:

```
java.lang.Runtime->getRuntime()
  ->exec("pm install -r " +
    new java.io.File-><init>(Activity->getFilesDir(),
                             "testnew.apk"));
```

The automatic analysis of the app bytecode already reports 13 unique sources leaked per sample, including IMEI, IMSI and various location and network information. However, the analysis of the supplemental apk revealed the truly malicious behavior, including code to contact a C&C server and to hide its activity by deleting premium SMS from the sent directory and command SMS from the inbox directory.

`NickyBot` and `GPSSMSSpy` can also be classified as SMS malware. However, they differ in the fact that they use SMS messages to track the user. Both malware families register a location listener to get updates about the user movement and then periodically report this movement via SMS messages the malware operator.

Besides that, most malware samples share common functionality discovered by BATI. Often the `WifiManager` is accessed to enable/disable Wi-Fi. The analysis also revealed that samples frequently access the private `ITelephony` class via reflection and call `disableDataConnectivity` and `enableDataConnectivity` to toggle mobile data connections on/off. Almost every malware sample comes with a set of either native libraries, supplemental bytecode files, or a set of binaries that are executed at runtime. The former two are generally utilized to hide malicious functionality and to increase the complexity for analysis tools. Shipped executable binaries found during analysis either contained root exploits, e.g. to be able to perform any operations on the file system, shell or super user binaries, or even VPN binaries to create point-to-point connections to a remote server.

VI. RELATED WORK

Improving Android's security has received a lot of attention by the security community in the recent years. A high number of works has concentrated on extending Android's security architecture—for instance, enforcing developer-defined policies [26], establishing IPC provenance [27], inlined reference monitoring [28], [29], [30], securing user interfaces [31], defending against ad libs [32], [33], or integrating generic access control frameworks [34], [35], [36].

In this section, we focus on comparing our analysis method to closest related work in the areas of *dynamic* and *static* analysis of Android applications.

A. Dynamic Analysis

The pioneering *TaintDroid* [37] architecture leverages Android's runtime environment to provide realtime data flow analysis to detect privacy leaks in applications. Data originating from privacy-sensitive sources is tainted automatically by the system and tracked as it propagates through the program during execution. Each time tainted data reaches a data sink and leaves the phone, for example via network sockets or Bluetooth, the user is informed. Besides the concrete values that are transmitted, TaintDroid also logs information about the destination. The *AppFence* [38] architecture extends TaintDroid with access control mechanisms that do not only log information leakage, but also prevent it.

DroidScope [39] is an analysis platform which, based on the Android emulator, follows the tradition of virtualization-based malware analysis. It provides APIs that facilitate custom analysis at hardware, OS, and application level. It supports taint analysis to track information leakage through both Java and native code components of applications.

Both approaches, TaintDroid and DroidScope, have been successfully applied to monitor and analyze applications at runtime. However, as dynamic analysis approaches they require a large set of input values to be comprehensive. Moreover, they impose a non-negligible performance overhead. Thus, they are inappropriate for *large-scale* analysis of applications, e.g., during app vetting processes.

B. Static Analysis

Static analysis of Android apps has been applied for different purposes.

Detection of privacy leaks: For iOS, Egele *et al.* [40] propose a privacy leak detector called *PiOS*. Similar to Bati they apply a backwards slicing algorithm. However, they base their analysis on simple CFGs and therefore have to use an additional forward propagation analysis to verify that sensitive data indeed flows to a sink. Specifically for Android, a number of different static analysis approaches exist to detect privacy leaks in Android apps. *AndroidLeaks* [8] applies a combination of control-flow and data-flow analysis to verify whether information propagates from sensitive data sources to sinks that subsequently send the data off the device. However, their analysis is neither object- nor field-sensitive and thus it is not suitable for a precise application vetting. *AppIntent* [9] identifies sensitive data leaks and checks whether these leaks are user-intended. Their approach uses a guided symbolic execution approach to reduce the search space to a tractable size. For any user-intended leak they come up with a sequence of UI events that triggers this functionality. *Chex* [11] leverages customized SDGs to detect component hijacking vulnerabilities in apps via graph reachability analysis. Their model addresses the interleaved execution of multiple app fragments, each reachable from a single entry point of the app. Their application vetting restricts itself to the detection of connections between externally accessible interfaces and sensitive sources/sinks. While their evaluation shows a reasonable false-positive rate for their targeted use-case, their approach is also inherently limited, since it is neither path-sensitive nor is it capable of precisely reasoning about values flowing into sinks. Providing those features, as in Bati, would benefit their precision. *LeakMiner* [10] as well as *FlowDroid* [15] are based on the analysis framework Soot and employ a static taint tracking approach to discover privacy leaks. In terms of precision FlowDroid comes closest to our approach. They combine a precise on-demand alias analysis with an accurate model of the Android lifecycle. In contrast to Bati, their approach is tailored for detecting privacy leaks. Reverse engineering application behavior that requires a precise string analysis is out of their scope.

Malware detection: Static analysis has also been used successfully for detecting malicious apps. Zhou *et al.* [41] detect piggy-backed apps based on the observation that malicious payload is loosely coupled with the original app code. The approach separates apps into modules based on package names and extracts feature fingerprints for each module. By use of a code similarity measurement apps that only slightly differ in code can be flagged as malicious. *DroidRanger* [42] employs a light-weight, heuristic-based analysis to detect malicious behavior in apps. Behavioral footprints like inclusion of supplemental files, suspicious combinations of permissions, or API usages are extracted from known malware. The *RiskRanker* [43] framework uses several heuristics to detect malicious apps. To provide more evidence, a backwards slicing approach is used to check if potentially malicious method calls (like sending SMS) are rooted at some callback methods that can be triggered by the user. Yet it remains unclear how they model the Android lifecycle and which algorithms they use for generating precise data structures and points-to analysis.

Reverse engineering: The *Static Android Analysis Framework (SAAF)* [7] applies a similar analysis technique as Bati. They use backward slicing to resolve register values of predefined sinks. Their approach is also meant for reverse engineering applications, but several limitations preclude a precise analysis of real-world apps. For example, SAAF does not descend into methods found during backtracking and it remains unclear how the authors model the Android lifecycle.

Analyzing inter-app communication: SCanDroid [13] was one of the first static analysis tools for Android. Its main focus is to check whether inter-component (ICC) and inter-app (IPC) data flows are consistent according to a specification, which they extract from the application manifest file. SCanDroid is based on the established Wala framework and implements a constraint system to track values across instructions. To model ICC/IPC it is essential to have an accurate string analysis to identify receivers. They approximate the string values by constructing a subgraph that includes StringBuilder operations and use Wala's flow solver algorithms to compute prefixes of those strings that flow into methods related to ICC/IPC. Another tool that generates ICC mappings is *Epicc* [12]. Its authors employ simple inter-procedural CFG traversal algorithms to detect hard-coded strings. String assembly that goes beyond constant assignments as well as handling data URI's that are used to address receiving components are out of

scope. In case the algorithm cannot determine the receiver, it conservatively assumes that it can be any receiver. This implies a transitive closure when ICC receivers cannot unambiguously be determined in two or more components. Its ICC resolution would thus greatly benefit from a precise string analysis as implemented by Bati.

VII. Discussion

Abstracting from framework internals implies that our analysis cannot track flows within framework classes. An example for such a framework class is `SharedPreferences`, a general framework for saving and retrieving persistent key-value pairs or inter-component communication. However, including the framework code does usually not suffice to handle these cases as most of them rely on runtime values like strings. As a remedy, we propose to use domain knowledge (Section IV-D) to add tailored semantic knowledge about intra-framework flows. In combination with our value analysis, this enables automatic reasoning in such situations. For instance for `SharedPreferences`, Bati can resolve the key used to store and retrieve a value and hence store and retrieve operations using the same key can be connected as source-sink pair.

Bati is particularly built for large-scale application vetting. Detecting sensitive data leaks and discriminating premium SMS from normal SMS is fully automated for a single iteration of the analysis. In scenarios without domain knowledge to automatically assess the slice, the analysis requires user interaction. Especially in the complex area of malware analysis our system provides a semi-automated analysis. For instance, a human analyst has to manually invoke the analysis on a supplemental apk, whose location was revealed in a previous run. In many cases, the vetting process can be automated further by adding more expert knowledge.

An inherent limitation of Bati (as well as of related work based on static analysis) is that flows to and from native code cannot be tracked. Native code completely differs from Android bytecode and its analysis requires dedicated techniques that are beyond the scope of this work.

VIII. Conclusion

We presented Bati, a vetting framework for Android applications. It extends the established Java analysis frameworks Wala/Joana with a comprehensive Android-specific lifecycle model. By combining program analyses like path-tracking slicing, partial evaluation, and array access analysis we receive a novel, precise, path-sensitive value analysis. Besides automated data leak detection, this kind of analysis caters to unravel (statically) undecidable problems (see premium SMS example).

We demonstrated empirically the high precision of Bati through a community-provided benchmark suite. Moreover, we showed its practicability based on a large-scale analysis of 19,700 apps from Google's Play Store and based on a malware analysis, where Bati statically detected malicious behavior that has previously only been described after manual code reviews, or not at all. In contrast to the popular conjecture that path-sensitive analysis is generally impractical, our test results show that Bati is indeed practicable for large-scale analysis.

As an ongoing task, we are currently implementing further use-cases that require a precise value analysis as implemented by Bati, for instance, an automatic analysis of ICC. As future work, we consider a deeper investigation on how reflection and dynamic code loading can be handled automatically with our static value analysis.

References

[1] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)*. ACM, 2012.

[2] "WhatsApp took all my contacts and sent to their servers without asking me - BlackBerry Forums at CrackBerry.com." http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/.

[3] F-Secure Labs, "Mobile Threat Report Q1 2014," http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2014.pdf, 2014.

[4] McAfee Labs, "McAfee mobile security report: Who's is watching you?" http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf, Februray 2014.

[5] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)*. ACM, 2012.

[6] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. pp. 358–366, 1953.

[7] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, 2013.

[8] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *Proc. 5th international conference on Trust and Trustworthy Computing (TRUST '12)*. Springer-Verlag, 2012.

[9] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)*. ACM, 2013.

[10] Z. Yang and M. Yang, "Leakminer: Detect information leakage on Android with static taint analysis," in *Proc. 2012 Third World Congress on Software Engineering (WCSE '12)*. IEEE Computer Society, 2012.

[11] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. New York, NY, USA: ACM, 2012, pp. 229–240.

[12] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. 22Nd USENIX Conference on Security (SEC '13)*. USENIX Association, 2013.

[13] A. Chaudhuri, A. Fuchs, and J. Foster, "SCanDroid: Automated security certification of Android applications," University of Maryland, Tech. Rep. CS-TR-4991, 2009. [Online]. Available: http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf

[14] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, 2011.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[16] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, Dec. 2009.

[17] J. Graf, "Speeding up context-, object- and field-sensitive SDG generation," in *Proc. 9th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Sep. 2010.

[18] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, 2012.

[19] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proc. 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*. ACM, 1988.

[20] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proc. 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*. ACM, 1988.

[21] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, 1988.

[22] S. Arzt, E. Bodden, and S. Rasthofer, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS '14)*. The Internet Society, 2014.

[23] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)*. ACM, 2011.

[24] "IBM Rational AppScan," www.ibm.com/software/products/de/appscan, May 2014.

[25] "HP Fortify Static Code Analyzer," http://www8.hp.com/us/en/software-solutions/application-security/, May 2014.

[26] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *Proc. 25th Annual Computer Security Applications Conference (ACSAC '09)*. ACM, 2009.

[27] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. 20th USENIX Security Symposium (SEC '11)*. USENIX Association, 2011.

[28] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *Proc. 21st USENIX Security Symposium (SEC '12)*. USENIX Association, 2012.

[29] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - enforcing user requirements on Android apps," in *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, 2013.

[30] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android," in *Proc. 2nd ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '12)*. ACM, 2012.

[31] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proc. 22nd USENIX Security Symposium (SEC '13)*. USENIX, 2013.

[32] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Proc. 21st USENIX Security Symposium (SEC '12)*. USENIX Association, 2012.

[33] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *Proc. 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM, 2012.

[34] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS '13)*. The Internet Society, 2013.

[35] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. 22nd USENIX Security Symposium (SEC '13)*. USENIX Association, 2013.

[36] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending android security," Intel CRI-SC at TU Darmstadt, North Carolina State University, CASED / TU Darmstadt, Tech. Rep. TUD-CS-2014-0063, Mar. 2014, to appear at USENIX Security'14.

[37] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, 2010.

[38] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications," in *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)*. ACM, 2011.

[39] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Security symposium (SEC '12)*. USENIX Association, 2012.

[40] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proc. 18th Annual Network and Distributed System Security Symposium (NDSS '11)*. The Internet Society, 2011.

[41] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proc. 3rd ACM conference on Data and application security and privacy (CODASPY '13)*. ACM, 2013.

[42] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[43] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day Android malware detection," in *Proc. 10th international conference on Mobile systems, applications, and services (MobiSys '12)*. ACM, 2012.

Appendix A
Bati Trace for Premium SMS Example (Listing 1)

```
Path (1/2):
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: android.telephony.SmsManager->sendTextMessage{v23}(v34,
     "0", "95pAHD", "0", "0")
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v23 = android.telephony.SmsManager->getDefault()
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v34 = java.lang.StringBuilder->toString{v31}()
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v31 = java.lang.StringBuilder->append{v25}(v19)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: java.lang.StringBuilder-><init>{v25}(v27)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v25 = new java.lang.StringBuilder
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v27 = java.lang.String->valueOf(v18)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: PHI v18 = "10", "106618"
    ConstToken: "10"
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: PHI v19 = "66953930", v16
    ConstToken: "66953930"
Result:
    android.telephony.SmsManager->getDefault()->sendTextMessage("1066953930", null, "95pAHD", null, null)

Path (2/2):
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: android.telephony.SmsManager->sendTextMessage{v23}(v34,
     "0", "95pAHD", "0", "0")
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v23 = android.telephony.SmsManager->getDefault()
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v34 = java.lang.StringBuilder->toString{v31}()
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v31 = java.lang.StringBuilder->append{v25}(v19)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: java.lang.StringBuilder-><init>{v25}(v27)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v25 = new java.lang.StringBuilder
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: v27 = java.lang.String->valueOf(v18)
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: PHI v18 = "10", "106618"
    ConstToken: "106618"
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 1>  InstrToken: PHI v19 = "66953930", v16
    <com.example.MainActivity.onCreate(Landroid/os/Bundle;)V, 2>  InstrToken(FLAGGED): v16 = com.example.MainActivity->getFragment{this}()
    <com.example.MainActivity.getFragment()Ljava/lang/String;, 0>  InstrToken(FLAGGED): return "5829"
Result:
    android.telephony.SmsManager->getDefault()->sendTextMessage("1066185829", null, "95pAHD", null, null)
```

Appendix B
Bati Trace for Android Lifecycle Example (Listing 2)

```
INFO  AppAnalysis :     elapsed time: 424 ms
INFO  AppHandler  : === Source/Sink Manager ===
INFO  AppHandler  :     Found 1 sink entry nodes
INFO  AppHandler  :     Process sink # 1/1
INFO  AppHandler  :         call     : java.io.FileWriter->write{v6}(v16)
INFO  AppHandler  :         in method: com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V
INFO  Slicer      :     # nodes in backwards slice: 13/674
DEBUG Slicer      :     Slice:
DEBUG Slicer      :         java.io.FileWriter->write{v6}(v16)
DEBUG Slicer      :         java.io.FileWriter-><init>{v6}(v4)
DEBUG Slicer      :         v6 = new java.io.FileWriter
DEBUG Slicer      :         v4 = android.os.Environment->getExternalStorageDirectory()
DEBUG Slicer      :         v16 = java.lang.StringBuilder->toString{v13}()
DEBUG Slicer      :         v13 = java.lang.StringBuilder->append{v8}(v11)
DEBUG Slicer      :         java.lang.StringBuilder-><init>{v8}("Device ID: ")
DEBUG Slicer      :         v8 = new java.lang.StringBuilder
DEBUG Slicer      :         v11 = com.example.DataLeakage{this}.deviceId
DEBUG Slicer      :         com.example.DataLeakage{this}.deviceId = v12
DEBUG Slicer      :         v12 = android.telephony.TelephonyManager->getDeviceId{v8}()
DEBUG Slicer      :         v8 = com.example.DataLeakage->getSystemService{this}("phone")
DEBUG Slicer      :         com.example.DataLeakage{this}.deviceId = "abc"
DEBUG AppHandler  :     Recreate paths:
DEBUG AppHandler  :     Path (1/2):
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.io.FileWriter->write{v6}(v16)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.io.FileWriter-><init>{v6}(v4)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v6 = new java.io.FileWriter
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v4 =
     android.os.Environment->getExternalStorageDirectory()
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v16 =
     java.lang.StringBuilder->toString{v13}()
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v13 =
     java.lang.StringBuilder->append{v8}(v11)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.lang.StringBuilder-><init>{v8}("Device ID: ")
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v8 = new java.lang.StringBuilder
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v11 =
     com.example.DataLeakage{this}.deviceId
INFO  PathMerger  :         <com.example.DataLeakage.onCreate(Landroid/os/Bundle;)V, 0>  InstrToken(FLAGGED):
     com.example.DataLeakage{this}.deviceId = v12
```

```
INFO  PathMerger  :         <com.example.DataLeakage.onCreate(Landroid/os/Bundle;)V, 0>  InstrToken: v12 =
     android.telephony.TelephonyManager->getDeviceId{v8}()
INFO  PathMerger  :         <com.example.DataLeakage.onCreate(Landroid/os/Bundle;)V, 0>  InstrToken: v8 =
     com.example.DataLeakage->getSystemService{this}("phone")
DEBUG AppHandler  :     Result:
DEBUG AppHandler  :         java.io.FileWriter-><init>(android.os.Environment->getExternalStorageDirectory())->write("Device ID:
     "com.example.DataLeakage->getSystemService("phone")=>android.telephony.TelephonyManager->getDeviceId())
DEBUG AppHandler  :     Found Sources:
DEBUG AppHandler  :         UNIQUE_IDENTIFIER  android.telephony.TelephonyManager.getDeviceId()Ljava/lang/String;
DEBUG AppHandler  :
DEBUG AppHandler  :     Path (2/2):
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.io.FileWriter->write{v6}(v16)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.io.FileWriter-><init>{v6}(v4)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v6 = new java.io.FileWriter
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v4 =
     android.os.Environment->getExternalStorageDirectory()
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v16 =
     java.lang.StringBuilder->toString{v13}()
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v13 =
     java.lang.StringBuilder->append{v8}(v11)
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken:
     java.lang.StringBuilder-><init>{v8}("Device ID: ")
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v8 = new java.lang.StringBuilder
INFO  PathMerger  :         <com.example.DataLeakage.leakDeviceId(Landroid/view/View;)V, 0>  InstrToken: v11 =
     com.example.DataLeakage{this}.deviceId
INFO  PathMerger  :         <com.example.DataLeakage.onPause()V, 0>  InstrToken(FLAGGED): com.example.DataLeakage{this}.deviceId = "abc"
DEBUG AppHandler  :     Result:
DEBUG AppHandler  :         java.io.FileWriter-><init>(android.os.Environment->getExternalStorageDirectory())->write("Device ID: ""abc")
DEBUG AppHandler  :     Found Sources:
DEBUG AppHandler  :         - No source found -
```