R-Droid: Leveraging Android App Analysis with Static Slice Optimization

Invited Paper

Michael Backes
CISPA, Saarland University & MPI-SWS
backes@cs.uni-saarland.de

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

Abstract

Today's feature-rich smartphone apps intensively rely on access to highly sensitive (personal) data. This puts the user's privacy at risk of being violated by overly curious apps or libraries (like advertisements). Central app markets conceptually represent a first line of defense against such invasions of the user's privacy, but unfortunately we are still lacking full support for automatic analysis of apps' internal data flows and supporting analysts in statically assessing apps' behavior.

In this paper we present a novel slice-optimization approach to leverage static analysis of Android applications. Building on top of precise application lifecycle models, we employ a slicing-based analysis to generate data-dependent statements for arbitrary points of interest in an application. As a result of our optimization, the produced slices are, on average, 49% smaller than standard slices, thus facilitating code understanding and result validation by security analysts. Moreover, by re-targeting strings, our approach enables automatic assessments for a larger number of use-cases than prior work. We consolidate our improvements on statically analyzing Android apps into a tool called R-DROID and conducted a large-scale data-leak analysis on a set of 22,700 Android apps from Google Play. R-DROID managed to identify a significantly larger set of potential privacy-violating information flows than previous work, including 2,157 sensitive flows of password-flagged UI widgets in 256 distinct apps.

1. INTRODUCTION

Modern smartphone apps offer an abundance of features that request from users access to the users' highly sensitive, personal data. The wide proliferation of these apps has made them a prime target for malware developers, and the variety of reported privacy incidents has fueled the legitimate privacy concerns of end users that their sensitive data is stealthily collected, monetized, and disseminated [20, 1, 15, 30]. Centralized app markets have responded by trying to identify malicious and overly curious applications even before these apps are deployed on a user's smartphone. To this end, they strive for comprehensive application vetting to understand app internals and to thereby identify abnormal app behaviours.

Static analysis of apps is widely accepted as a well-suited, automated concept for application security vetting on a large scale. In the context of Android, prior work has already successfully identified particular security and privacy problems like (user-intended) privacy leak detection [23, 17, 44, 43, 42], component hijacking vulnerability detection [29], and misuses of (framework) features such as the crypto API [14] or dynamic code loading [36], to name a few.

All these approaches share the common goal to precisely capture which data flows into security- and privacy-sensitive method calls. Unfortunately, existing static analysis approaches are (fully or partially) agnostic to the *concrete* data values that arise during execution of an app (i.e., strings or primitive values) and that can make a crucial difference in assessing an app as being harmless or dangerous to the users' privacy. For instance, the concrete value for a receiver number in a text-message app tells apart a legitimate app from one that sends premium SMS messages. Some approaches have considered this problem of statically recovering concrete runtime values, but are currently limited to coarse-grained approximations for runtime strings [34, 23, 9, 10], which results in conservative approximations (*"could be any string"*, or *"any combination of these strings"*) in more evolved cases. As a consequence of this limitation, they face many *false positives* (i.e. false alarms) and are not suitable for assessing certain evolved cases at all (see Section 3.)

Moreover, an aspect that has received little attention so far is that any static analysis always requires a significant amount of manual investigation either to validate the results or to understand *how* data is processed within the application code. However, manually investigating the outputs of existing approaches even for simpler cases—typically a huge list of data-dependent statements and an involved kind of formal security assessments—typically constitutes an intricate task, since existing tools either work with the app's bytecode [23, 36] or transform it into an intermediate representation [7, 42, 29, 17] that is usually even less amenable to manual review than the original source code. As a consequence, the efforts for a human analyst in validating and assessing the results of current analysis solutions are significant.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

DOI: http://dx.doi.org/10.1145/2897845.2897927

Our contributions. To address the aforementioned challenges, we present a novel slice optimization approach to facilitate privacy- and security assessments on Android applications. Our approach relies on a standard slicing-based analysis to generate data-dependent statements for arbitrary points of interest in an application. While existing app analysis approaches already assess apps based on those early results, our analysis proceeds by further optimizing the slices statically. Our optimization offers the following benefits: 1. We provide a general purpose value analysis to precisely reconstruct values/strings to ease (semi-)automatic checks for more evolved security assessment tasks. 2. Our optimization pipeline statically transforms slices into semantically-equivalent, concise slices. This improves readability and reduces the number of false positives, a major benefit for an efficient reviewing process. The optimized slices can subsequently be further assessed by distinct *security modules* that create additional insights about sensitive data flows within the application and that facilitate manual app reviewing. Technically, we make the following three contributions:

1. Novel slice optimization approach. We start by leveraging a system dependency graph (SDG) that distinguishes different objects of the same type (object sensitivity), fields of the same object (field sensitivity), calling contexts of invoked methods (context sensitivity), and definitions of the same local variable on different paths through a method (flow sensitivity.) We add a comprehensive application lifecycle model to faithfully take Android's peculiarities into account. On top of a slicing-based dependency analysis, we propose a novel slice optimization approach that, on a high level, constitutes a use-def tracker to eliminate spurious dependences that the data-dependency analysis failed to resolve and a comprehensive value analysis to re-assemble strings and primitive values that are passed as parameters to security-relevant functions. To this end, we adopt optimization techniques from partial evaluation, domain knowledge and copy propagation to receive concise and semantically-equivalent slices with a low number of statements. As a result we receive optimized slices that are about 50% smaller than the original slices, making any subsequent manual reviewing task more efficient. Moreover, more evolved security problems such as premium number assessment (see Section 3) can be evaluated automatically due to our value analysis.

2. Complementary Analysis via Security Modules. Our approach supports the integration of further security modules to extend and amplify the analysis of apps' w.r.t. user's security and privacy. Security modules define their own data flow sources/sinks and receive the optimizated slices to perform security assessments. In this paper, we realize three such security modules—data leakage detection, user input propagation, and slice rendering for manual code review—that we used for our large-scale evaluation of Google Play apps.

3. R-Droid and Large-scale Evaluation on Google Play. We consolidate all aforementioned features into a tool called R-Droid. The evaluation of R-Droid on the widely accepted, open-source testsuite DroidBench excels over related work [7, 42] with nearly optimal results: 97% precision (one false alarm) and no missed violation. In a large-scale evaluation of 22,700 apps from Google Play, R-Droid managed to identify a significantly larger set of potential privacy-violating information flows than previous work, including 2,157 sensitive flows of password-flagged UI widgets in 256 distinct apps. Finally, we demonstrate the effectiveness of our approach on manual code reviewing on the common use-case of understanding malware behavior.

2. RELATED WORK

Improving Android's security has received a lot of attention in the security community. Over the last years, a larger number of analysis frameworks was published with a focus on information-flow aspects such sensitive data/user input leakage. Table 1 shows a high-level feature comparison of static analysis frameworks on Android sorted by publication year in ascending order. For the sake of simplicity we distinguish feature support in three categories: ✓ = comprehensive/sophisticated, ● = basic, and ✘ = no support.

Most related work [9, 26, 29, 42, 18] performs a data-dependency analysis on top of an application lifecycle model. While this is sufficient for binary assessments, e.g. is there a dependency between a source and a sink statement, more complex problems like reconstructing values/strings passed to an API call require a detailed list of dependent statements. Moreover, such data does not necessarily have to be derived from an API call (see premium SMS example in Section 3). Similar use-cases such as API (mis-)use [36, 35, 16, 14] in which the concrete (string) data needs to be assessed can not efficiently be processed by forward approaches, since *any* string has to be marked as source. In contrast, our backward slicing approach fulfills the requirements to conceptually handle all of these problems.

Table 1 shows an evolution in terms of supported features. The foundation of a static analysis is a comprehensive data model that precisely approximates Android's runtime behavior. *Chex* [29] was the first tool to take the different types of application entry points into account. *FlowDroid* [7] improved on this by introducing accurate per-component lifecycle models that were also adopted by *AmanDroid* [42]. R-Droid follows prior work and additionally adds models for frequently used classes to further refine the static lifecycle model. To keep the analysis complexity tractable almost all approaches manually model parts of the semantics of the framework API instead of adding the full framework code base to the analysis scope. *DroidSafe* [18] proposed a new technique to abstract from the framework complexity while still keeping all data-dependences. R-Droid adopts this model to not rely on incomplete knowledge about the framework API. A dedicated line of work focused on Android's inter-component communication (ICC) either as standalone approach [34, 33] or as part of an analysis framework [9, 27, 42]. While tracking data-flows across components clearly increases the precision of the overall analysis, our privacy leak evaluation (cf., Section 7) showed that such flows rarely occur in real-world apps.

String analysis. Most of the aforementioned work on API usage analysis as well as first works on ICC resolution [34] resort to rather limited constant propagation approaches tailored to one specific use-case. *SCanDroid* [9] resolves ICC receivers by implementing a constraint system to track values across instructions. String values are approximated by constructing a subgraph that includes StringBuilder operations. Flow solver algorithms then compute feasible string prefixes that flow into ICC-related calls. Christensen *et al.* [11] propose the Java String Analyzer (*JSA*) to statically check the syntax of dynamically generated expressions like SQL queries.

		SCanDroid [9]	Scandal [26]	AndroidLeaks [17]	LeakMiner [43]	Chex [29]	FlowDroid [7]	AmanDroid [42]	DroidSafe [18]	R-Droid
Approach	Framework used	WALA	custom	WALA	Soot	WALA	Soot	custom	Soot	JOANA
	Fwd/Bwd analysis	—	Fwd	Fwd	Fwd	—	Fwd	Fwd	Fwd	Bwd
	Analysis approach	Data-Dep (DD)	DD	Slicing	Tainting	DD	Tainting	DD	DD	Slicing
	String analysis	●	●	✘	✘	✘	✘	●	●	✓
Features	Apk parsing	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Lifecycle modelling	●	●	●	●	●	✓	✓	●	✓
	Framework modelling	●	●	●	●	●	●	●	✓	✓
	ICC support	✓	✘	✘	✘	✘	✓	✓	✓	✘
	Concurrency	●	✘	✘	✘	✓	✓	✘	✓	✓
	Reflection	✘	●	✘	✘	✘	●	✘	✘	✘
	Native code	✘	✘	✘	✘	✘	✘	✘	✘	✘

✓ = comprehensive/sophisticated support, ● = basic support, ✘ = no support

Table 1: High-level feature comparison of static analysis frameworks and extensions for Android. Frameworks sorted by publication year in ascending order.

Dedicated flow-graphs for string operations are translated into context-free grammars to infer possible string values. *DroidSafe* uses *JSA* to resolve ICC. The *IC3* project [33] is the closest related approach to our value analysis. They employ context-sensitive, inter-procedural composite constant propagation that can handle field correlations of complex objects. To describe the semantics of the Android API (we refer to this as expert knowledge) they devised a declarative language and use a constraint solver to output possible string values. Similarly, our value analysis uses inferred expert knowledge, but, in contrast, our approach is also object- and path-sensitive. Since our multi-stage value analysis is not restricted to strings, even complex propagation problems such as adding values to arrays/lists with subsequent retrieval via index calculations is possible.

Statically resolving string values is also highly relevant beyond Android. Automata-based approaches for string analysis exist to verify SQL expressions [39] or for finding string-related security vulnerabilities in PHP programs [45, 46]. Tateishi *et al.* [40] apply path- and index-sensitive string analysis to verify Cross-Site-Scripting (XSS) sanitizers for web applications based on monadic second-order logic (M2L). They verify that generated strings satisfy a given constraint rather than assembling concrete values like R-Droid does. String solvers have recently also been integrated into SMT solvers [41, 28, 47]. These solvers determine whether a certain string can be produced by a program (e.g. whether a XSS attack is possible). In contrast, R-Droid determines *which* concrete strings can be created in an app.

Manual analysis support. Dedicated support for manual analysis has been largely disregarded by related work so far. Static analyses always underlie manual reviewing to a certain extent, either to validate the results or to understand how data is processed. Eliminating the number of false positives, e.g. by refining the static runtime model, is a first but insufficient step towards an efficient reviewing process. While many frameworks only provide binary information about security assessments, others at least provide complementary information like code location or a list of tainted/sliced instructions. Approaches that disassemble [3] or decompile the application [32] provide readable high-level, but untargeted, output. Instead, R-Droid produces semantically-equivalent, small-sized output that can be fed into our code rendering module to provide similar readable, but targeted output.

```
1  public class MainActivity extends Activity {
2    protected void onCreate(Bundle savedInstanceState) {
3      String val1 = "10";
4      int val2 = 66953930;
5      if (Math.random() > 0.5d) {
6        val1 = "106618";
7        val2 = getValue();
8      }
9      SmsManager sm = SmsManager.getDefault();
10     sm.sendTextMessage(val1+val2, null, "95pAHD", null,
           null);
11   }
12   private int getValue() { return 5829; }
13 }
```

Listing 1: Premium SMS example

3. MOTIVATING EXAMPLE

We start with the illustrative example of *premium number classification*—a major monetization factor of malware—to demonstrate why current data leak detection and reachability analyses are insufficient for a faithful security analysis of Android apps. The example is depicted in Listing 1 as a code excerpt for sending premium SMS that many Android SMS malware variants rely on [48]. In line 10, an SMS with activation code *95pAHD* is sent (without user interaction) once the *MainActivity* is displayed. The premium receiver number is selected randomly and assembled via string concatenation, which constitutes a simplistic form of obfuscation.

Existing approaches based on static analysis do not resolve the concrete values for the receiver number of the text message (SMS), for different reasons. Forward analysis approaches [26, 43, 7, 17, 42, 18] rely on sensitive sources to execute their analysis, none of which are present in this example. Backward analyses approaches do not resolve the concrete numbers as well: approaches tailored to data leak detection [7, 42] do not flag this example as critical, due to the absence of sensitive information; basic string analyses [23, 9, 34] are either limited to constants and/or lack path-sensivity. Heuristic approaches simply retrieve string values from the bytecode and combine them in various ways to determine meaningful combinations. This will miss the implicit conversion of *val2*, an integer variable, to a string during concatenation; internally, this constitutes a `StringBuilder.append` call. Moreover, intra-procedural approaches [23] do not appropriately capture the number *5829*, which is returned by the method call `getValue` (it is a potential value as well, since it is implicitly converted to a string). Moreover, even if all values can be identified, existing analyses do not deter-

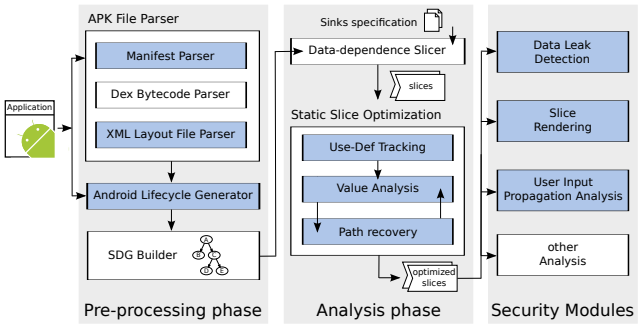

FIGURE 1: HIGH-LEVEL ARCHITECTURE OF R-DROID

mine which values are possible on which path. The common remedy is to enumerate all possible combinations of the identified strings, introducing false positives. In a nutshell, a path-sensitive value analysis is essential for determining the actual numbers *1066953930* and *1066185829*, which can in practice be compared against lists of known premium numbers.

4. ANALYSIS FRAMEWORK

The high-level architecture of our R-DROID is depicted in Figure 1. It comprises a pre-processing phase in which it collects application meta-data and generates a comprehensive application lifecycle model (cf. Section 4.2) that is subsequently used to create a system-dependence graph (SDG). For the analysis we leverage a standard (backward) data-dependence slicer. It takes an arbitrary list of sinks (in terms of method signatures) as input to capture data that may influence these statements. The output is post-processed by our optimization pipeline to obtain semantically equivalent slices with a low number of statements and a lower number of false positives. These optimized slices can then be fed into customized security modules, e.g. for privacy leak or input propagation analysis. The implementation of R-DROID (blue colored boxes) comprises approximately 11.5 kLOC, including code for the three security modules.

4.1 Pre-processing Phase

We start by extracting information included in the application package and analyze the manifest file that, among others, declares the apps' components and meta-data like the requested permissions, as well as layout files containing UI descriptions. To generate app data models, R-DROID leverages the information flow control (IFC) framework JOANA [22, 21]. Its frontend includes the analysis framework WALA [2], that offers an intermediate-representation (IR) in static-single assignment (SSA) form [4, 38]. Their Dalvik frontend, adopted from SCanDroid [9], transforms Android bytecode directly into WALA's IR. We generate bytecode using dexlib [3] to model a static application lifecycle model for each app that takes the Android peculiarities into account.[1] The lifecycle-enhanced app bytecode is subsequently used by JOANA to generate an object-, context- and field-sensitive SDG. The SDG contains intra- and inter-procedural data- and control-dependencies (also for exceptions) as well as object-sensitive points-to information to resolve dynamic dispatch. To keep

[1]WALA's immutable IR precludes direct altering

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

LISTING 2: ANDROID LIFECYCLE EXAMPLE

the complexity of the overall data structure tractable we do not include the full framework code. Instead, we use the light-weight framework model of the *DroidSafe* [18] project to capture data-dependencies within the framework.

4.2 Android Lifecycle Modeling

Android apps adhere to a complex event-driven application lifecycle that challenges static analysis approaches. Applications consist of multiple components that are asynchronously triggered by events, or launched (and stopped) by user interaction. Each of these components maintains its individual lifecycle with predefined callback methods that are implicitly invoked by the runtime environment. Developers override these callback methods such as `onCreate` or `onPause` (cf. Listing 2) to initialize data structures, to save an app's state before closing it or switching it into the background. Moreover, event-listeners can be registered for services (e.g. location events are triggered whenever the device's location changes) or UI-events (e.g. the button-click handler `leakId`.)

To build an accurate lifecycle model, R-DROID starts by adopting the entry point discovery algorithm from [29]. Using the entry points discovered in the app's manifest file and the overridden component callbacks, R-DROID builds a callgraph and performs a code reachability analysis in order to identify new entry points—registered event-listeners and overridden framework methods. This process is repeated until convergence, i.e., until the entry point set reaches a fixed point. To improve precision over permitting lifecycle callbacks to be called in arbitrary order, R-DROID follows prior work [7, 42] and models individual component lifecycle methods that exploit the partial callback ordering. The resulting per-component models contain fewer invalid paths, which amends the precision of our subsequent analysis.

Finally, we generate a synthetic main method to connect the individual lifecycle methods. This method serves as single entry point for the analysis and mimics the initialization routine of an app that is executed on a real device. Concretely, the static class initializers are invoked first, with `ContentProviders` being the first components created during application launch [12]. After that, custom application classes are invoked following the order provided by the class

hierarchy. Finally, all other components can be executed in any order.

During our experiments we found the resulting model can be further extended by adding accurate lifecycles for additional, frequently used, components: `Fragments` and Android's special threading class `AsyncTask` including parameter passing. These integral features either have not been considered by prior work or have been coarsely modeled, which results in false positives during analysis, i.e., invalid paths.

4.2.1 Fragment Lifecycle

Android 3.0 introduced fragments as a design pattern to support more dynamic and flexible UI designs, which became imperative with the increasing number of tablets. Fragments can be classified as reusable sub-components of activities with a dedicated code base and an optional user interface. They maintain an individual lifecycle, receive input events, and can dynamically be added to and removed from a running activity via a `FragmentManager` object.

Fragments require a host Activity (`FragmentActivity`) for their execution. Similar to callbacks, fragments can either be statically added to the UI of an activity (via layout descriptions), or added/removed dynamically via `FragmentTransactions`. While parsing the layout files for `Fragment` declarations is straightforward, determining the concrete types in transactions requires points-to information. R-Droid generates this information along with the callgraph for the basic lifecycle model. As output we receive a one-to-many mapping from `Activity` to `Fragments`. Event-handling for fragments is analogous to activities except for one case: Callbacks registered in a layout resource do not necessarily have to be declared in the fragment or in one of its inner classes, but can also be declared in its host activity. Identified callbacks can be discharged in arbitrary order while the fragment is running. For each identified `Fragment` R-Droid generates a lifecycle method in accordance to the official documentation [5] and subsequently adds it to the callback body of its host activity while active.

4.2.2 Modeling AsyncTask

Android's application model imposes very strict response times on 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`, the Android SDK provides a dedicated thread class (`AsyncTask`) for outsourcing such code into background tasks (e.g., to download a file from the Internet) and feeding results back to the originating thread (e.g., as a progress monitor). Similar to `Fragments`, `AsyncTasks` are increasingly used by app developers (in our large-scale app evaluation we found 62.7% of 22,700 apps to include at least one AsyncTask). Hence a correct model of this feature is mandatory for static analyses to avoid false positives in this component.

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. To precisely model this behavior (including data passing between these callbacks) we propose the `AsyncTask` lifecycle depicted in Figure 2.

An AsyncTask is specified by three generic types, i.e. `AsyncTask<Params,Progress,Result>` that are used as argument and return types of its callback methods. If a task

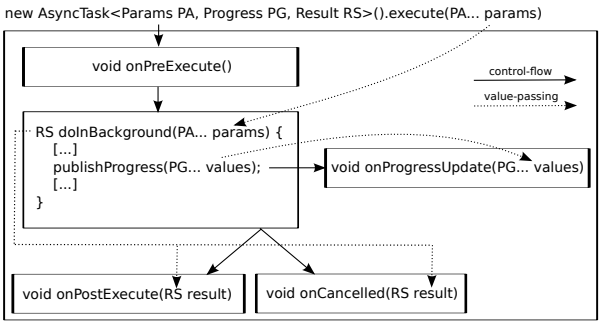

Figure 2: AsyncTask lifecycle

is triggered, the `onPreExecute` method is executed to set the task up. After that, the `doInBackground` method executes in a background thread to realize the main functionality. This method takes a *varargs* argument, basically corresponding to 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. While the `doInBackground` method is executed, the method `publishProgress` can be invoked with a *varargs* argument that then triggers the callback `onProgressUpdate` with the same argument. R-Droid automatically identifies the concrete types for the generic parameter types (`<Void,Void,String>` in Listing 2) and generates a tailored lifecycle method to reflect this behavior. For analysis purposes, this lifecycle method is used instead of the framework's `execute` method.

Array arguments in callbacks constitute another challenge for static analysis. Section 5.2 explains in detail how R-Droid resolves them as part of its value analysis. To the best of our knowledge, we are the first to generate a comprehensive model of multi-threading in Android, including the complex and prevalently used AsyncTask lifecycle with precise parameter passing.

5. SLICE OPTIMIZATION

Many related apporaches [9, 26, 17, 43, 7, 42, 18] are essentially tailored to kinds of privacy leak detection, i.e., detecting whether there is some flow/data-dependence between sensitive sources and sinks. Others, focus on specialized forms of API usage [36, 35, 16, 14] which usually requires a precise reconstruction of values/strings for (semi-)automatic assessment. All these app vetting approaches underlie manual reviewing for result validation due to the absence of a ground truth. None of these tools provides dedicated support to facilitate such a reviewing process.

Hence, we propose a new slice optimization with the following benefits: 1. We provide a general purpose value analysis to precisely reconstruct values/strings to ease (semi-) automatic security checks. 2. Our optimization pipeline statically transforms slices into semantically-equivalent, concise slices. This improves readability and reduces the number of false positives, a major requirement for an efficient reviewing process.

Our slice optimization pipeline is designed as multi-stage post-processing technique. Given a standard data-dependence slice generated by JOANA, R-Droid first applies def-use tracking to eliminate spurious dependences. Then, the value analysis is applied until convergence, i.e., no more optimiza-

```
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 = DataLeakage{this}.deviceId
10 DataLeakage{this}.deviceId = "fakeId"
11 DataLeakage{this}.deviceId = p1
12 Entry DataLeakage$ATask.onPostExecute(java.lang.String)V
13 DataLeakage$ATask.onPostExecute{this}(v3)
14 v3 = DataLeakage$ATask.doInBackground{this}()
15 return v12
16 v12 = ..telephony.TelephonyManager→getDeviceId{v8}()
17 v8 = DataLeakage→getSystemService{v2}("phone")
18 v2 = DataLeakage$ATask{this}.this$0
```

OUTPUT 1: CONTROL-FLOW ORDERED SLICE OF DATALEAKAGE EXAMPLE AFTER USE-DEF TRACKING

tions can be applied. To allow for more aggressive optimization and value retargeting, R-DROID tries to recover execution paths within the slice to re-apply the optimizations on each path on success.

5.1 Use-def Tracking

Traditional SDG-based slicing approaches [24] result in flow-*in*sensitive slices. However, as illustrated in Listing 1, flow and path information is essential to recreate exact values for automatic security assessments. Moreover, the resulting slices might contain spurious dependences, i.e., statements that do not directly influence arguments of the sink. This is because the slicer resolves data-dependencies for complete statements (such as method calls) and not for subsets of arguments. This usually leads to large slices and may additionally introduce false-positives.

We therefore conduct, as first optimization step, a use-def analysis on the resulting slices. Starting from the sink statement we iteratively backtrack register and field references and add defining and dependended statements to the result. To this end, we leverage WALA's SSA-based IR, in which use-def chains are explicit, i.e., for each use there is exactly one definition. We add flow-sensitivity by ordering the statements in reverse control-flow order. While backtracking explicitly adds flow-sensitivity for defining statements, statements such as multiple calls on the same object can be ordered using flow information obtained by control-flow graphs (CFGs) of the enclosing methods (cf., list operations in Output 3). The resulting slices are more concise and readable since non-relevant statements have been eliminated.

Output 1 shows the slice of the *DataLeakage* example in Listing 2 after use-def tracking. Beginning at the sink, i.e., the `write` method (line 1), data-dependent statements are iteratively added in reverse-control flow order. An `Entry` meta statement (line 12) is added as the second field assignment depends on the parameter `p1` of the `onPostExecute` method. 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 activity state: either the constant string *fakeId* (line 10) or the actual device id accessed (line 16) in a new instance of the `AsyncTask` class *ATask*.

5.2 Value Analysis

Our value analysis (generalized string analysis) statically simplifies complex expressions and re-assembles strings beyond constant values. It currently comprises four optimization steps that leverage techniques from partial evaluation, domain knowledge, and copy propagation. Execution path recovery is performed whenever possible to allow more aggressive optimization. The optimization pipeline is iteratively applied until convergence, i.e. until no more slice statements are modified. The outcome are semantically equivalent slices that contain fewer but more expressive instructions (due to retargeted strings and values). These optimized slices ease manual reviewing and allow a larger range of security assessments to be performed (semi-)automatically.

Copy propagation. We adopt WALA's copy propagation to eliminate assignment statements (that may occur as result of the other optimizations) and copy constants/registers directly to the statements in which they are used. This also applies to values stored in and later retrieved from class fields. Moreover, R-DROID eliminates function calls that return constant values/references, such as getter methods.

Evaluating unary/binary operations. This step statically evaluates unary and binary operations. Further optimizations like resolving indices for array and lists depend on this phase. R-DROID uses points-to information to determine the type of an operation such as `int` or `double`. We then statically calculate such operations iff the operand values are constants or can be iteratively resolved, e.g., an integer addition $x = 17 + 23$ is evaluated to $x = 40$. Static resolution fails if at least one operand is non-constant, e.g., the result of a framework call to `Math.round()`. In this case we cannot simplify the slice without expert knowledge.

Array access resolution. Related work on Android [23, 17, 43, 7, 42, 18] usually cannot precisely resolve array indices and thus over-approximates array modifications. This does not only reduce precision but also results in false alarms when sensitive data is written at position `x` but is later leaked from position `y`. Albeit challenging, resolving array access statically significantly improves the precision of the analysis. This particularly applies to the `AsyncTask` array parameter as described in Section 4.2.2.

R-DROID can accurately resolve array access: Based on its control-flow ordered list of array update instructions for every execution path, it statically reconstructs the content for each access and resolves the respective index. Three outcomes are possible:

1. If the access index i is statically computable and the data can be unambiguously determined at position i, then R-DROID can precise determine the accessed content: it discards the array instructions and replaces them with the value at position i.
2. If the access index is statically computable but this position can contain different data at the time of the access, R-DROID returns a list of possible values.[2]
3. If the index is not statically computable, R-DROID returns a template defining how the index is computed and a string representation of the reconstructed array. In case domain knowledge or a human expert *cannot* resolve this access further, all possible values must conservatively be considered in a subsequent security analysis.

[2]This may happen if the array is both updated with statically computable and non-computable indexes.

```
1  v27 = v8[v25]
2  v8["4"] = "no taint"
3  v8["5"] = v16
4  v16 = ..telephony.TelephonyManager→getDeviceId{v3}()
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, cf-ordered slice of `ArrayAccess2`

Output 2 shows the relevant part of the slice for the testcase `ArrayAccess2` of the benchmark suite DroidBench (see Section 7). This testcase evaluates if an analysis approximates array operations. An array is filled with sensitive (device Id, line 3-4) and non-sensitive data (*"no taint"*, line 2), of which the latter is finally leaked via SMS. Conservative algorithms will spuriously report a sensitive data leak. In contrast, R-Droid can statically resolve the array access. Given the array register `v8` and the array update instructions (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 index `v25` in line 1 is computed by a series of operations in the `calculateIndex` method (line 6-10). The expression evaluator starts with the return statement and iteratively assembles and solves the expression $(((1+1)*5)\%10)+4 = 4$. Thus, the non-sensitive value *"no taint"* is assigned to v8["4"] in the reconstructed array. With this semantic-preserving assessment we correctly classify this test as non-leaking.

Domain knowledge. Retargeting values and strings in presence of API methods requires an understanding of the API semantics. For example, the concrete string value of the SMS receiver number (*"1066953930"* in Listing 1) is internally constructed via calls to the `StringBuilder` constructor and the `append` method. To enable automatic reasoning for such API methods we manually model their semantics as domain knowledge, e.g. in this case the final string is the concatenation of the provided arguments. This knowledge is represented as rules specifying method signatures and semantic descriptions how method arguments are transformed. These rules are then applied to (control-flow ordered) sequences of API calls on the same object.

Besides modelling `StringBuilder` and `StringBuffer` operations we further add domain knowledge for commonly used Java collection classes including various kinds of `List`, `Map`, and `Set` implementations. Internally, they behave like arrays and provide convenience functions for the developer. We encode the getter/setter methods of these classes as domain knowledge and handle them analogously to arrays. Output 3 shows an excerpt of a slice that creates a list object, subsequently adds non-sensitive (`abc`) and sensitive data (`v20`) and finally retrieves the first element. Without domain knowledge, the slice would be incorrectly flagged as a sensitive data leak. Adding semantic rules for the `add`, `get` and `init` methods, we can reconstruct the list content and correctly output the string *"abc"* from the first position. This reduces the number of false positives, when sensitive and non-sensitive data is stored in the same collection. If the argument of the getter method is statically not computable,

```
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 = ..telephony.TelephonyManager→getDeviceId{v16}()
8  v16 = MyActivity→getSystemService{this}("phone")
```

Output 3: Slice containing list operations

or if there is no semantic rule for a framework method in the slice, we conservatively return the original instructions.

In total, we modelled 104 methods in 23 classes as domain knowledge. This allows retargeting more complex values/strings from low-level API calls and hence increases the number of security checks that can be performed automatically. Moreover, we further simplify the slice and increase the precision of the result in the aforementioned cases.

5.3 Path Recovery

Being able to discriminate execution paths within the slice allows a more aggressive optimization since register value ambiguity (e.g. due to branching) might be resolved. R-Droid detects three branching indicators within the slice: phi-instructions, callee-to-caller information and field value retrieval/update relationships. WALA's IR offers flow-sensitivity for local variables due to the SSA form. Phi statements, an essential building block of SSA form, are located at intra-procedural control-flow merge points and represent which variable modification arises from which path. Field setter/getter operations are not tracked by the intra-procedural phi instructions and have to be handled separately. For the latter two branching indicators, multiple execution paths exist if there is a one-to-many mapping, i.e. if there is one field retrieval and multiple associated field update instructions or a non-API function is called from multiple locations.

Depending on the complexity of the slice this step might introduce a large number of execution paths or might not be feasible at all. To maintain a good cost-to-benefit ratio, a threshold for the maximum number of paths and processing time is configurable. If one of these values is exceeded the optimization is stopped, otherwise the aforementioned optimization steps are re-applied to each execution path slice.

If R-Droid detects path information, it transforms the slice into a tree presentation (*path tree*). If this fails, e.g. due to recursive code within the slice, we terminate this optimization step. Tree nodes constitute basic blocks (BB)[3] from CFGs of the instructions' enclosing methods. Slice statements are subsequently mapped to their BB. Figure 3 shows the path tree generated for the receiver number in Listing 1 (two execution paths) with four nodes. Our tree model differs from CFGs in that instructions are connected rather than blocks, since the smallest unit of resolution are instruction arguments. If consecutive instructions of the slice reside in different basic blocks, we generate a waypoint. For phi-instructions at control-flow merge points, the waypoint has outgoing edges for each successor, e.g., `waypoint1` points to the assignment instructions of `valA` and `valB`.

During tree traversal, the path extraction algorithm traverses the same outgoing edges on waypoints of the same node (either A or B). This prevents the generation of invalid paths through impossible combinations, e.g., in our example

[3] A basic block is a code fragment with only one entry and one exit.

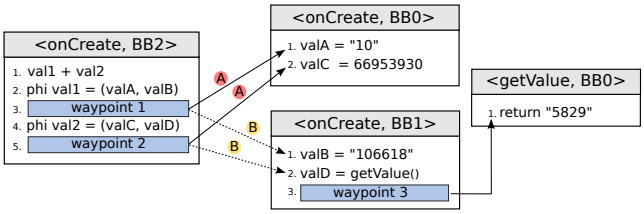

Figure 3: Path tree for receiver number in Listing 1

only two out of four distinct paths are feasible. Outgoing edges of waypoints are traversed before its immediate successor within the same node (if any). Within a node the algorithm stops if 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 (which in turn are flow-sensitive slices) that contain the instructions to reconstruct the correct receiver numbers.

6. SECURITY MODULES

R-Droid is a generic analysis framework tailored to the specifics of Android that can easily be complemented by further analysis modules for security and privacy analysis of applications. We demonstrate its effectiveness and relevance by defining three such security modules—data leakage detection, user input propagation, and slice rendering.

6.1 Data Leakage Detection

Declared app permissions only indicate what an app could potentially do, but do *not* adequately capture the *actual* behavior of the app. In particular, correlations between permissions are hidden, e.g., if address book data leaks to the Internet. Consequently, users cannot appropriately assess the risk entailed by installing an app [37, 8, 19]. Our data leakage detection module reports source-sink pairs identified between permission-clad APIs.

We leverage the sources and sinks of *SuSi* [6], and extend it as follows: Sinks from the Apache classes `HTTPClient` and their subclasses are added as they often constitute substantial parts of the app's network communication. We add sources that are (a) argument-dependent or (b) constructed via a series of method calls. A prominent example for (a) is the API to access the secure system settings, in which the sensitivity of the returned data depends on the argument. The device ID is frequently accessed (via argument `"android_id"`) and serves as unique identifier. Our module classifies the sensitivity of the result according to the parameter resolved during value analysis. The locale device settings are an example for (b), as `getLanguage()` and `getCountry()` only represent sensitive information if accessed via `java.util.Locale->getDefault()`. Finally, we include framework fields that provide sensitive data into our list of sensitive sources. `android.os.Build` and its subclasses provide numerous data about the installed Android build and version. To preclude false positives, we disregard sinks for which the app lacks the required permission (libraries frequently probe for permissions of the host app) using the permission map generated by *PScout* [8].

6.2 User Input Propagation Analysis

User input propagation is a specialization of the previous module that focuses on leaks of user data provided via UI widgets. It is configured with the same sinks, however we identify sources from UI input accessed via `findViewById` of `android.app.Activity`, `android.view.View`, and their subclasses. Input fields marked as passwords are of particular interest, hence we check for the respective view attributes. Slices that include user input may be forwarded to the slice-rendering module to improve readability and to facilitate manual assessment. Recent work [25, 31] presented an approach to cover a wider range of sensitive user inputs (like credit card inputs) by inferring input widget sensitivity via UI layout descriptions including labels and hints. Integrating such approaches is an interesting future extension.

6.3 Slice Rendering Module

Related work is usually limited to answering *whether* data may flow but has limited support for answering *how* as it does not account for individual execution paths between a source and a sink. Our evaluation shows that this is insufficient as traditional slices usually contain a significant fraction of the original program (we found on average 33 statements/slice and examples with up to 4k statements), which impedes manual inspection.

Our optimization pipeline efficiently reduces the number of instructions per slice. Still, reading bytecode or instructions in some intermediate representation is not as convenient as reading source code. To close this gap, this module renders the optimized slices in a human-readable format. To this end, we transform statements into series of call chains on the same object. Concretely, the algorithm starts at the sink and collects all invocations on the target object, reorders them according to runtime execution order and iteratively merges them. Class names are omitted when the class is constant for consecutive invocations. Non-constant arguments, i.e., return values from framework APIs, are iteratively replaced by variables. The result omits redundant information and improves readability (see Section 7.4).

7. EVALUATION

We evaluated R-Droid on the original DroidBench testsuite and received nearly optimal results. To demonstrate the scalability and utility of our approach we conducted a large-scale data/input leakage analysis on 22,700 apps from Google Play in which R-Droid identified a large number of privacy-critical data flows originating from sensitive APIs or from UI widgets. Finally, we elaborate on our slice optimization and the effects of our slice rendering module on manual reviewing efforts.

7.1 DroidBench Test Suite

DroidBench is an evolving open-source test suite [13] containing Android apps crafted to evaluate static and dynamic analysis approaches. Many of these synthetic test cases evaluate the recall of the performed analysis, hence there is a bias towards over-approximating approaches. There is only a small number of cases that explicitly check the precision, i.e., whether the analysis reports on the actual flows and does not generate false positives. We test R-Droid on the widely used original DroidBench (v1.0) to demonstrate the effectiveness of our lifecycle modelling and the benefits of our slice optimization on the few cases that pose such a challenge. We compare our results with FlowDroid [7], AmanDroid [42],

Sinks by category \ Sensitive sources by category	Account Info	Build Info	Bluetooth Info	Calendar	Contacts	Database	File Info	Locale Info	Location	Network Info	NFC info	SMS-MMS	Unclassified	Unique Identifier	Version Info
Code Loading	12	477	27	153	–	388	17	328	184	471	1	1	91	376	205
File	2	16	2	23	–	36	–	7	9	37	2	2	10	19	3
Log	146	3563	285	1516	2	3648	83	2501	1676 (**16.1%**)	4401 (**26.5%**)	31	29	1031	3148 (**42.1%**)	1570
ICC	1	10	2	34	–	42	–	11	5	110	–	1	7	12	7
Network	49	1017	121	505	–	1099	22	803	493 (**4.8%**)	1255 (**7.6%**)	6	12	302	897 (**12.2%**)	478
SMS-MMS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Unclassified	202	6335	389	2305	13	5034	127	4797	2715 (**26.1%**)	6137	49	38	1388	4618 (**61.7%**)	2921

Table 2: Information flows from sensitive sources to sinks grouped by category.

and DroidSafe [18] and exclude the four implicit test cases that none of the tools support (as did the original authors).

FlowDroid achieves a high precision (86%, 4FPs) and recall (93%, 2 missed flaws). Amandroid, that adapts their accurate lifecycle modelling, offers similar precision and recall as FlowDroid without being explicit about the failing testcases. DroidSafe used a novel framework abstraction to capture data-dependences within the API. As a result, they achieve an optimal accuracy (they detect all explicit flows). However, their approach also erroneously reports sensitive flows due to their flow-*in*sensitivity and missing lifecycle modelling. Instead, our approach yields a precision of 97% (1FP due to a missing inter-procedural must-alias analysis) and 100% recall. R-Droid particularly excels in the category *ArraysAndLists* that tests over-approximation for container classes, where all other tools fail.

7.2 Data Leakage Analysis

Even though benchmarks like DroidBench are a valuable tool to compare different approaches, they are hand-crafted and their coverage of functionality is rather limited, given the modest number of tests. Therefore, we conducted a large scale data leakage analysis on 22,700 apps from the Google Play Store. The apps were crawled between August 20-23, 2014, starting from the top 100 of each category and iteratively crawling recommended and similar apps. For this evaluation we configured R-Droid's path recovery with a 3 min timeout and a maximum number of 32 paths. During analysis, this affected 19% of all sinks. In these cases, the slices were assessed after the first value analysis run, which might cause imprecision. The experiments were conducted on a server with four Intel Xeon CPU E5-4650L @ 2.60GHz processors with 8 cores each and 768GB RAM on which we ran 64 *single-threaded* analyses in parallel. Despite its precision, R-Droid had a reasonable average processing time of 26min of which the graph builders consumed about 90%.

Our experiments emphasize the relevance of fragments in current apps (33% contain at least one). As an increasing fraction of code has been moved from activities to fragments, modelling their lifecycles accurately becomes imperative for precise static analyses. Our evaluation shows that apps, that were originally published prior to Android 3.0, often still do not adhere to the recommended fragment-based layout design. For all other apps, however, we detected 11 fragments on average, which corresponds to the average number of activities per app. Similarly, AsyncTasks are becoming standard for network communication. 14,244 apps (62.7%) include at least one AsyncTask.

Table 2 summarizes our findings as flows from sensitives sources to sinks (grouped by categories as provided by the SuSi project). We report the absolute number of flows between source and sink category for all analyzed apps. These numbers constitute an over-approximation since apps do not always have the respective permission(s) for a detected flow (e.g. if the flow is located in a third-party library). To eliminate such inaccuracy, we use publicly available API-to-permission information [8, 6] to compute the ratio of detected flows to the number of apps in the data set that hold all required permissions.

Our findings confirm the common belief that highly sensitive data like user location or unique IDs are frequently accessed and leaked via various channels. As an example, unique IDs are frequently written to logs (42.1%) and to the network (12.2%). There is a significant number of flows to unclassified sinks. The classification depends on the runtime type of the receiver and requires a dedicated analysis. For example, the `write` methods of an `OutputStream` may write data to memory, to a file or to the network depending on the concrete stream instance. Similarly, `ContentResolvers` are usually queried with a data URI. The concrete parameter value describes the data type to be accessed, e.g. contacts. Locale, version, and build information is widely ignored as sensitive source. Besides common cases in which such information is written to the system logs for debugging purposes, manual investigation revealed that a combination of this information is commonly used to generate a unique user-agent ID to identify and track the user. This clearly indicates that access to such low-sensitive data (that is not protected by permissions) may threaten the user's privacy and allows such apps to evade API-based privacy leak detection analyses. R-Droid did not find sensitive data flows to `SMS-MMS` sinks. One reason could be the rather small set of apps that declare the required permissions (2.6% in our test set). Another one, that SMS are not suitable to transmit large amounts of data and in fact most of these apps can be manually classified as phone finder/tracker that notify the owner via SMS about odd incidents. Although ICC modeling is out of scope for this work, we quantified how often sensitive data reaches API methods that initiate ICC, such as `startActivity`. The absolute numbers are relatively small. The fraction of cases that actually leak the received data is presumably even smaller, implying that complex ICC-related privacy leaks rarely appear in real-world apps.

7.3 Leakage of Sensitive User Input

Private data may not necessarily originate from API calls, but may also be inserted by the user via user interface widgets. Detecting such data flows requires a dedicated analysis as described in Section 6.2. Applied to the Google Play test set R-DROID reported a total number of 2,157 flows of password-flagged UI widgets in 256 distinct apps. We manually validated the flows in 150 apps. 9% of all flows were erroneously flagged as leaking. Some of these false positives could be eliminated by deriving more export knowledge. Although not confirmed by a user study, the effort necessary to manually investigate these cases was notably lower due to the smaller and more concise output. Similarly, sorting instructions by control-flow helped in reading and understanding the output (see also Section 7.4). We exemplify our findings in the following:

Case study: Logging of private user data. We identified that developers frequently log data that is supposed to be written to files or the network. Although this facilitates debugging during development, it may be considered a privacy breach by users if sensitive input is included in the log. The app `com.bitsontherun.android.dashboard` allows uploading videos to the *Bits on the Run* online video platform. Among others, the server response of a sign-up attempt, including password and email address from an UI widget in plain, is written to the system logs.

Case study: Plain user input transmission via insecure channels. User-supplied input is frequently sent to remote servers via HTTP. As part of its service registration `com.CG.checkgmov2` sends a POST message including the first and last name, email address, and password in plaintext. This data originates from UI widgets, where one is marked as password. `com.camilo.hkingorders` manages/tracks purchases made on *hobbyking.com*. Within the login routine, R-DROID reconstructed the authentication request during value analysis as follows: ***http****://www.hobbyking.com/hobbyking/store/uh_customerAuthenticateExec.asp?email=$EMAIL &password=$PW* (placeholders were inserted for the sake of brevity). This clearly violates the user's privacy, as we manually verified that the webshop supports HTTPS. We also found several cases in which user data is transmitted via the facebook graph library. `com.bokskya.books` integrates this API to publish user feeds. The widget used to enter status messages is marked with the password flag by the developer since such messages could include private information. However, R-DROID reported that these messages are sent to Facebook via HTTP. Manual investigation of the API documentation revealed that the library indeed does not support secure transmission via HTTPS (at that time).

7.4 Assessing Manual Reviewing Support

As shown in this section, additional manual investigation is often required to either validate findings or to understand app behavior in detail. R-DROID's optimization pipeline supports such efforts in multiple ways. During our Google Play evaluation the slice optimization generated semantically-equivalent slices that are 49% smaller than standard slices on average. The use-def tracking outputs slices that contain 34% less instructions on average. The output size is further reduced by 25% after the application of the value analysis. This is due to the fact that most optimizations like string assembly, binary operation calculation, or array access resolution involve multiple instructions that are, in the best case, optimized to a single value. In many cases the final slice contains less than 15 instructions. In addition, the control-flow ordering of instructions and the generic string/value assembly facilitate code understanding.

```
1  v10 = java.lang.Runtime→exec{v7}(p2)
2  v7 = java.lang.Runtime→getRuntime()
3  Entry hider.InstallService.execCommand1
       (android.content.Context,java.lang.String)Z
4  v27 = hider.InstallService→execCommand1(p1, v24)
5  v24 = java.lang.StringBuilder→toString{v21}()
6  v21 = java.lang.StringBuilder→append{v17}(p2)
7  java.lang.StringBuilder→<init>{v17}("pm uninstall ")
8  v17 = new java.lang.StringBuilder
9  Entry hider.InstallService.uninstallapp
       (android.content.Context,java.lang.String)Z
10 v24 = hider.InstallService→uninstallapp(this, "hider")
11 v36 = hider.InstallService→execCommand1(p1, v33)
12 v33 = java.lang.StringBuilder→toString{v30}()
13 v30 = java.lang.StringBuilder→append{v26}(v21)
14 java.lang.StringBuilder→<init>{v26}("pm install −r ")
15 v26 = new java.lang.StringBuilder
16 java.io.File→<init>{v21}(v23, p2)
17 v21 = new java.io.File
18 v23 = hider.InstallService→getFilesDir{this}()
19 Entry hider.InstallService.runRootCommand1
       (android.content.Context,java.lang.String)Z
20 v44 = hider.InstallService→runRootCommand1{this}(this, "newapp.apk")
21 v33 = hider.InstallService→runRootCommand1{this}(this, "testnew.apk")
```

OUTPUT 4: CF-ORDERED SLICE OF PACKAGE INSTALLATION AFTER USE-DEF TRACKING

In the following we elaborate on our slice rendering module, which transforms the optimized slices into readable and structured output to further ease manual investigation. To deduce malware functionality it is commonly necessary to manually analyze and understand its code. Our example (from the Malware Genome Project [48]) belongs to the `jSMSHider` family. This SMS malware targets Android users with a custom ROM. As these devices are already rooted the malware can install packages without the user's explicit consent. Output 4 shows the slice for a "command execution" sink after use-def tracking. (for the sake of brevity we replaced the original package name by *hider*). Although the partially-optimized slice is moderate in size, it shows that with an increasing number of instructions manual analysis becomes tedious. The slice contains a total of three execution paths. The argument `p2` (line 1) of the sink statement depends on the method argument of the `execCommand1` (line 3) which has two caller sites (line 4+11). The invocation on line 11 depends on the method argument of `runRootCommand1` in line 19 which again has two caller sites (line 20+21).

The rendering module receives slices for each execution path from the optimization pipeline and transforms them into structured code (see Output 5). The result precisely shows that the malware uses this code segment to both uninstall itself from the system (line 1) and to install its supplemental apk file, either named *newapp.apk* or *testnew.apk*. Manual analysis of samples of this family revealed that each sample only contains one of these supplemental apks (presumably to evade signature-based malware detection mechanisms).

8. CONCLUSION AND FUTURE WORK

We presented a novel slice optimization approach as post-processing to standard slicing techniques. As part of this approach we devised a comprehensive value analysis to retarget strings and values beyond constants. It is generally applicable and allows a larger number of security- and privacy-related

```
1 P1: java.lang.Runtime→getRuntime()→exec("pm uninstall hider")
2
3 P2: x1 = java.io.File→<init>(hider.InstallService→getFilesDir(),
        "newapp.apk")
4     java.lang.Runtime→getRuntime()→exec("pm install −r "+ x1)
5
6 P3: x1 = java.io.File→<init>(hider.InstallService→getFilesDir(),
        "testnew.apk")
7     java.lang.Runtime→getRuntime()→exec("pm install −r "+ x1)
```

Output 5: Rendered optimized code for Output 4

use-cases, such as various API (mis-) use analyses, to be assessed in an automatic way. Moreover, the concise output of R-Droid supports experts in understanding app functionality and in manually reviewing the results, a mandatory task for any static analysis. In ongoing work, we like to implement more API usage modules to further evaluate and extend the effectiveness of our approach. Automatically deriving expert knowledge from the Android API to improve the optimization rate would be an exciting future work.

Acknowledgments

This work was supported by the German Federal Ministry for Education and Research (BMBF) under project VFIT (16KIS0345) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the initiative for excellence of the German federal government.

9. REFERENCES

[1] 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/.

[2] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net, 2006.

[3] Dexlib Android bytecode library. https://code.google.com/p/smali/, 2009.

[4] 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.

[5] Android Documentation: Fragment. http://developer.android.com/guide/components/fragments.html.

[6] 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.

[7] 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.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[9] A. Chaudhuri, A. Fuchs, and J. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, 2009.

[10] 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.

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

[12] ContentProvider API documentation. http://developer.android.com/reference/android/content/ContentProvider.html.

[13] EC SPRIDE Secure Software Engineering Group. DroidBench. https://github.com/secure-software-engineering/DroidBench.

[14] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[15] F-Secure Labs. Mobile Threat Report Q1 2014. https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf, 2014.

[16] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)*. ACM, 2012.

[17] 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.

[18] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proc. 36th International Conference on Software Engineering (ICSE '14)*, pages 1025–1035, 2014.

[20] 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.

[21] 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, Sept. 2010.

[22] 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*, 8(6):399–422, Dec. 2009.

[23] 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.

[24] 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.

[25] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (SEC'15)*, pages 977–992, Washington, D.C., Aug. 2015. USENIX Association.

[26] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In *Mobile Security Technologies 2012 (MoST'12)*. IEEE, 2012.

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[28] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proc. 26th International Conference on Computer Aided Verification (CAV '14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of 19th ACM Conference on Computer and Communications Security (CCS '12)*, pages 229–240, New York, NY, USA, 2012. ACM.

[30] 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.

[31] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (SEC'15)*, pages 993–1008, Washington, D.C., Aug. 2015. USENIX Association.

[32] D. Octeau, W. Enck, and P. McDaniel. The ded Decompiler. Technical report nas-tr-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2010.

[33] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. 37th International Conference on Software Engineering (ICSE '15)*, 2015.

[34] 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.

[35] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin—helping app developers bullet proof their tls connections. In *24th USENIX Security Symposium (SEC'15)*, pages 239–254, Washington, D.C., Aug. 2015. USENIX Association.

[36] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[37] 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.

[38] 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.

[39] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 0:13–22, 2007.

[40] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[41] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proc. 21th ACM Conference on Computer and Communication Security (CCS '14)*, pages 1232–1243. ACM, 2014.

[42] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. 21th ACM Conference on Computer and Communication Security (CCS '14)*, 2014.

[43] 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.

[44] 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.

[45] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 154–157. Springer, 2010.

[46] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 306–324. Springer, 2008.

[47] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*, pages 114–124. ACM, 2013.

[48] 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.