# Practical Application-Level Dynamic Taint Analysis of Android Apps

Julian Schütte, Alexander Küchler, Dennis Titze

*Abstract*—**Dynamic taint analysis traces data flows in applications at runtime and allows detection and consequently prevention of flow-based vulnerabilities, such as data leaks or injection attacks. While dynamic taint analysis spanning all components of the stack is potentially more precise, it requires adaptations of components across the OS stack and thus does not allow to analyze applications in their real runtime environment. In this paper, we introduce a dynamic taint analysis framework for Android applications which injects a taint analysis directly into an application's bytecode and can thus operate on any stock Android platform. Our approach is more precise than previous ones, copes with flow-aware source and sink definitions, and propagates data flows across process boundaries, including propagation over file I/O and inter process communication. We explain how our framework performs with popular apps from the Google Play Store and show that it achieves a precision which is comparable to the most precise platform-level tainting framework.**

## 1. Introduction

Data flow vulnerabilities are a class of security flaws in programs which are based on unwanted flows of data across variables and registers. Typical data flow vulnerabilities are all kinds of missing input sanitations like SQL and code injections, but also data leaks threatening the privacy of the user. Dynamic taint analysis (DTA) is a well-known technique to track flows of data from sources to sinks during execution of a program in order to either detect or even prevent such data flow vulnerabilities at runtime. For platforms like Android, which are based on an application framework where execution of programs is an interplay of the actual application code and the underlying application framework and operating system, dynamic taint analysis can be implemented at different layers. Tracing data flows only at the operating system level is not a reasonable approach, as it would exclude semantics of the bytecode interpreter and thus make data flow tracing across memory regions a much more complex task. But when including both the operating system and application framework in the analysis, semantics of the application lifecycle and the framework API can be used in the taint analysis, which makes it much simpler to keep track of data flows at runtime. However, in practice there are major drawbacks of such platform level DTA. The platform itself must be modified at various places to support taint analysis. This affects the file system, virtual machines (i.e., ART or dalvik), as well as the Binder module for inter process communication (ICC). These dependencies on the platform make dynamic taint analysis less portable and bind it to a specific platform, whereas the applications are more or less independent of the specific platform version and may run on completely different platform configurations. As a consequence, average users cannot use dynamic taint analysis and profit from a respective data leak prevention on their phone, and security testers are limited to a specific platform version which would soon be outdated.

In contrast, taint analysis at application level only requires modification of the application and thus overcomes the requirement of modifying the Android system and its different platform components. The challenges here are however to reflect the effects of the Android framework (i.e., its API and its lifecycle management) and to capture flow-aware source and sink definitions to achieve the precision of platform-level taint analysis. Several solutions (e.g., [2], [14]) performing application-level dynamic taint analysis have been presented to the research community. These frameworks could not achieve a precision similar to platform-level tainting as they lack a precise handling of inter-process data flows, especially the use of files and ICC.

Our contribution is a framework for injecting dynamic taint analysis into Android applications which operates exclusively at application layer and achieves a precision comparable to the most popular platform-level dynamic taint solution TaintDroid [7]. This allows us to apply highly precise taint analysis to any Android application, independent of the Android platform itself, and to observe and prevent unwanted data flows their actual runtime environment, i.e. on any Android version. In summary, our contributions are the following:

1) A flow-aware multi-statement definition for data sources and sinks
2) Precise handling of file accesses at application level
3) Tracing data flows across ICC calls
4) Tracing data flows across Android system services such as SharedPreferences

After a discussion of related work in section 2 we point out the challenges of DTA, specifically with regard to application-level DTA and the Android platform in section 3. In section 4 we present our approach of injecting taint analysis at bytecode level and in section 5 we explain the details of our prototype implementation. Finally, in section 6 we evaluate our approach with respect to precision and performance.

## 2. Related Work

TaintDroid [7] is one of the most prominent dynamic taint analysis frameworks for Android and achieves high precision with an average runtime overhead of 7%. It mostly operates at the level of the dalvik virtual machine (VM) and thus covers taint propagation in both the Android SDK and the app likewise. Further, it tracks taint of files and ICC messages. We chose TaintDroid as an evaluation benchmark for our solution for its high precision, knowing that it is conceptually more complete compared to our approach. However, we aim to overcome the practical drawbacks of TaintDroid: as it operates at platform level, it requires a specifically adapted version of the Android system. For the average user, this makes TaintDroid not a viable option, as it voids guarantees of the phone and requires a non-supported fork of the OS. For security analysts, the dependency on a specific Android version supported by TaintDroid is a hindrance because it is not possible to analyze applications in their real environment, i.e., on a recent non-rooted stock Android. TaintDroid supports Android versions from 2.1 to 4.3[1] which amounts only to 18.6 % of devices in the field and excludes all apps requiring newer OS versions from being analyzed. This is not merely an issue of maintaining the system, but is rather conceptually founded in TaintDroid being integrated into the VM which changed from dalvik to ART and would thus require a rewrite of major parts.
XManDroid [5] extends the Android framework to monitor apps to detect escalation attacks based on inter-component communication (ICC).
Similar to our approach, Capper [14] implements application-level DTA to instrument the app. Capper inserts shadow registers holding the taint state of registers directly into the bytecode and thereby unnecessarily doubles the amount of registers needed by the application. In contrast, we use a global taint table to directly map objects to their taint state and thus achieve a memory overhead linearly to the number of created objects, instead of to the number of method calls. It uses static analysis to reduce the number of statements to instrument, which improves its runtime but reduces its precision by increasing its false negative rate to the limits of the static flow analysis. Further, it does not trace taints over ICC or files.
AppCaulk [12] combines static analysis to inject targeted DTA into Android applications. It uses heuristics to deal with data flows across process boundaries, which potentially leads to drastic overtainting for the actual traced data flows. It considers only files and sockets as communication channels between processes, while ICC is out of its scope.
Jadal [10] combines static and dynamic analysis for Java applications and has been extended to support Android [2]. It analyses control flows but does not dynamically trace taint states. Quire [6] and Scippa [3] construct an ICC call chains to detect insecure collaborations between applications. In contrast to our approach they require modifications to the Binder ICC and thus depend on the underlying platform.

1. At the time of this writing, Android 7 is the most current version

## 3. Specifics of Application-level DTA

We first introduce the basics of dynamic taint analysis and highlight the challenges when applying it at application-level only. For further reference on DTA, we refer to [13] and for foundations on program analysis to [11].

The goal of dynamic taint analysis is to trace how data read from a *source* is propagated during program execution until it is written to a *sink*. A source is usually a *definition* statement, i.e., the first assignment of critical data to a memory location in the context of the program. When data is read from a source, it is assigned a *taint flag*, which is a label indicating the type of data. A sink is respectively a *use* statement where tainted data is written into an unsafe method. The *taint propagation logic* defines how taint flags propagate across memory locations depending on the current statement and includes conditions that will *untaint* memory locations, i.e., remove a taint flag.

This general notion of dynamic taint analysis is independent from the target platform and universal to its implementation. When applying it to a real program, one has to take several design decisions which influence the performance and precision of the analysis:

**Taint level.** When tracing data flows at the platform level, memory locations refer to actual locations in virtual process memory and statements refer to CPU instructions. In contrast to that, we aim at DTA at application level, which is limited to the Android bytecode. In our case, memory locations are registers in the dalvik or ART virtual machine and statements refer to dex bytecode instructions.

**Management of taint labels.** While for platform level tainting, it is possible to run a separate global "taint engine", this is not possible in application-level DTA, because we need to restrict ourselves to the application's bytecode and its process. Rather, we have to modify the application itself in order to make it keep track of its taint labels at runtime and will inevitably loose control over taint flags at the process boundary, i.e. whenever code outside of the application's bytecode is called. Especially ICC calls, file I/O, JNI calls, and Android services may break taint propagation and must be covered by either transporting taint flags across these channels or reflecting their effects in the taint propagation logic.

**Flow-sensitivity of sources and sinks.** Platform level tainting typically refers to sources and sinks as single instructions (e.g., a `send` syscall on a socket file descriptor). In high-level languages, practical source and sink definitions do not refer to single statements, but rather to program slices. Consider the following code snippet in Listing 1.

Listing 1. Example of a multi-statement source

```
File f = new File("some_file.txt");
InputStream fis = new FileInputStream(f);
BufferedReader br = new BufferedReader(fis);
while ((String line = br.readLine()) != null) {
    // reading from file ...
}
```

When limiting source and sink definitions to individual statements, it remains unclear how to handle this snippet. If line 4 was defined as a source, all read operations from `BufferedReader` would be marked as a critical data source, which would lead to massive overtainting of `BufferedReader`, while still neglecting all other ways to read from a file. Treating the `File` constructor in line 1 as a source is also imprecise because on the one hand, we would miss the various other ways to construct `File` objects without explicitly passing the file name to the constructor. On the other hand, the call of the `File` constructor does not necessarily impose a data source, so we would increase both false negatives and false positives. Hence, application-level taint analysis must be able to cope with flow-sensitive source and sink definitions.

## 4. A Dynamic Taint Injection Approach

We inject DTA into Android applications by statically instrumenting their dex bytecode. As long as modification is legal in terms of passing the verification at installation and runtime of application, this is a valid operation. The cryptographic signature of the application will break, but resigning it with a different key will create a new valid Android application which runs on stock platform (except it cannot be installed as an update to the original app). The instrumentation preserves the original semantics and adds the ability to trace taint flags. As dex bytecode is register-based, there is no need to taint memory addresses, which allows for a very precise analysis.

### 4.1. Instrumentation flow

The basic flow of the instrumentation is shown in Figure 1. We use the Soot framework [8] for bytecode instrumentation, which disassembles bytecode into smali [4] and translates it into the Jimple intermediate representation (IR). Jimple is a three-address typed intermediate representation (IR) which is well suited for instrumentation because it allows to leave reorganizations of registers to the Soot compiler and is thus less error-prone than a direct modification of smali disassembly. We then inject a dynamic taint analysis into the Jimple IR, including source and sink tracing, a global taint table, read/write instructions for updating the table upon every operation on data registers, and the actual taint propagation logic. Finally, the instrumented app is packed and signed again to provide a new, valid apk file with the same semantics as before but now performing dynamic taint analysis. In the following subsections, we explain the details of the injected DTA.

### 4.2. Taint storage

A clear advantage of application-level taint analysis in bytecode over DTA at platform-level is that it is possible to taint objects held in registers, instead of blocks of memory locations, which allows a more precise analysis. We store

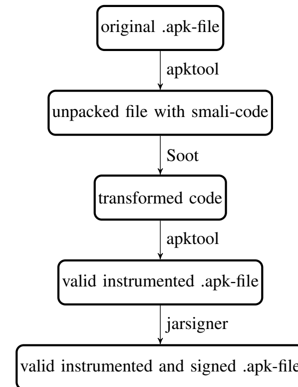


Figure 1. Instrumentation workflow

taint labels in a global taint table which maps from an object to its taint for every object in the application. Depending on what part of the object carries taint information, we consider the three possibilities

1) The whole object $o$ contains sensitive data
2) A field $o.x$ contains an object carrying sensitive data
3) A static field $C.X$ of a class $C$ contains an object carrying sensitive data

For each of these cases a different storage of taint is necessary. The first case requires a mapping from object $o$ to its taint. For the second case, an object maps to the values of its current attributes. The last case requires a mapping from class $C$ to its static attributes.

Listing 2 shows the motivation for this storage by an example for all the three possibilities of storing taint. Comments indicate the entries in the taint table after the line is executed. When the IMEI is read in line 1, the string containing it is tainted accordingly what complies to the first way of tainting data. In line 2, $o.a$ also contains the IMEI but as we can observe the field, we do not taint the whole object $o$ but only register that $s$ is assigned to $o.a$. When the value of $o.a$ is updated in line 4 of the code, the taint of $o$ does not have to be recalculated – i.e., the taint label IMEI would not be correct anymore – but only the attribute in the list is replaced. Also, when the static attribute $C.c$ is set in line 5, this only affects the list of static fields of class $C$ but not the taint of its outcomes e.g. $o$.

Listing 2. Example for the Taint Storage

```
String s = TelephonyManager.getDeviceId(); // s -> {"IMEI"}
o.a = s; // o -> {("a" -> s)}, s -> {"TAG_IMEI"}
o.b = y; // o -> {("a" -> s), ("b" -> y)}, ...
o.a = z; // o -> {("a" -> z), ("b" -> y)}, ...
C.c = x; // C -> {("c" -> x)}, ...
```

The approach benefits from the possibility to build the taint of an object at run time from its direct taint and the taint of all of its static and non-static attributes by calculating the disjunction of all the taint values. Whenever a new value of an attribute is set, the list of attributes is updated.

This taint storage allows us to have the most recent taint of an object with minimal additional computation to update the taint. The taint of objects that are an instance of a class

implemented in the app is precise as the taint of the object is always built from the current attributes. We use the objects as key for the taint value and compare them on their memory addresses. This allows us to differ between equal data that are derived in a different way.

For all library classes, we can only use the first possibility as we cannot observe the values of their fields. For these classes, we need to analyze how a method modifies the taint of an object. This is defined in the taint propagation logic of our dynamic taint framework.

## 4.3. Taint propagation

The taint analysis assigns taint labels to objects $o$. Taint labels can be regarded as bits in a bit vector $L$ of length $k$ where $L_i = 1$ indicates that the label at position $i$ is set. Each instruction of the code retrieves a set $IN$, reflecting the global taint state, and produces a new global taint state, determined by $OUT$. The taint propagation logic defines for each instruction how it affects the global taint state. It does so by specifying a $GEN$ set of newly introduced taint flags and a $KILL$ set of taint flags to remove. The effect of a single instruction $l$ can then be written as the set of input taint flags, minus all removed (killed) flags, joined with all newly introduced taint flags:

$$OUT(l) = GEN(l) \cup [IN(l) \setminus KILL(l)]$$

The definition of $GEN$ and $KILL$ is the core of the taint policy and depends on the operation of the instruction. Here lies one of the main differences of application-level tainting compared to platform-level-level tainting: while for the latter the definition of $GEN$ and $KILL$ only depends on the opcode of an instruction, application-level tainting must also take into account function names and thereby potentially specify a taint propagation policy for the full set of API functions, i.e., all functions which are not declared inside the analyzed application.

With more than 100.000 API methods [1], manually defining their tainting behaviour seems like an infeasible task. In practice, however, the tainting behavior of API methods can be inferred from their signature in most cases: we analyzed a large part of the Android API and summarize the tainting behavior of API calls by the following default rules. Assuming a statement consists of a call to a method $m$ of a base object $b$ (non-static methods) or of a base class $C$ (static methods) and passes arguments $v_1, .., v_n$ to the method, where $v_i$ is an object, we can define the default tainting behavior as follows:

If the statement is a *non-static method invocation*, the taint of any argument is propagated to the base object:

$$GEN_{invoke_virtual} = \{taint(b) = \bigcup_I taint(v_i)\}$$

As a consequence, the call of any setter of a Java object with a tainted argument will taint the object itself. As our taint analysis ends at the boundary of the application and does not inspect the API itself, tainting the base object is the only reasonable choice and in fact the correct one for non-static invocations. In theory, the heuristic is imprecise in two cases: propagating to the base object introduces overtainting when the method implementation does not have any side effects. Not propagating the union of all taint flags to all arguments leads to undertainting, if the method implementation copies values from one argument to another. The following example method demonstrates the worst case in which the heuristic fails:

Listing 3. Method implementation for which the heuristic fails

```
1 class Example {
    void copy(byte[] arg0, byte[] arg1) {
3     for (int i=0;i<arg0.length;i++) {
        if (i<arg1.length) {
5         arg1[i] = arg0[i];
  }}}}
```

As the `copy` method does not have side effects, tainting the `Example` object instance is incorrect. Further, as values are copied from `arg0` to `arg1`, it is incorrect to not copy the taint labels, too. In practice, however, we did not come across an API method matching this pattern, but rather determined that the taint propagation heuristics from above apply. Implementations like in Listing 3 are typically found in static methods (e.g. `System.arraycopy`, `Arrays.copyOfRange`, etc.).

Hence, for *static method invocations*, we propagate taint among all arguments of a method call:

$$GEN_{invoke_static} = \{taint(v_j) = \bigcup_I taint(v_i) \ \forall v_j \in I\}$$

This reflects the characteristic of static methods as being stateless utility functions, either returning constant values from getters or operating on arguments.

Finally, just like for any other method invocation, if the method call is contained in an *assign statement*, the taint status of the objects on the right hand side ($rhs$) is propagated to the object $o$ on the left hand side.

$$GEN_{assign} = \{taint(o) = \bigcup_{r \in rhs} taint(r)\}$$

We use the same heuristics to propagate taint over JNI calls of the app.

## 4.4. Handling intents

Intents transfer messages between components of applications – either within the same application or across process boundaries. An application constructs an intent, optionally adds payload to it and have the Android *binder* kernel module send it to one or more receivers. Hence, we need to treat the use of intents different from internal data flows within the app. Also, it is desirable to exchange taint information between different apps, given that they are both instrumented.

To address the challenge of taint tracking across ICC, it is necessary to attach the taint status of an intent to the intent. Hence, we make use of the intent's so called *extras* attribute, which is a set of key-/value pairs sent with the

intent. We use these extras to store the taint status of the intent and transfer it to the receiver(s). The aforementioned taint propagation logic will taint intent objects when tainted data is stored as a payload. Before an intent is sent, we get the intent's taint and add an additional extra containing the taint. Whenever an intent is received, we check if taint information in the extra is present, taint the intent with the taint information from the extra and delete the extra. Doing so, we can achieve message-level taint for intents with minimal effort. Also, this method is stable and applicable to every Android version since API version 1.

### 4.5. Persistence services

The Android API provides two ways for applications to persist data: shared preferences and files, the latter including sqlite databases, raw files, and sockets. Shared preferences are a typical representative of globally available Android services over which applications can share data using ICC. The `SharedPreferences` service stores key-value pairs persistently and either limits access to the respective application or makes them globally accessible. While treating such external services as data sinks is a conservative and viable choice, treating them as data sources introduces overtainting, as not all data read from them is necessarily critical. Hence, we strive for a way to precisely propagate taint over these channels, instead of regarding them as sources and sinks.

Simply tainting the service object (e.g., the `SharedPreferences` instance), as done by $GEN_{invoke_virtual}$ is not an option: first, this taint would not propagate across process boundaries and lifecycle stages of the app, i.e., it would be lost when terminating the app. Second, it would not be possible to distinguish taint labels of individual records.

To address these issues, we store an additional key-value pair whenever a pair is stored. The key is derived from the original key and the value represents the string-representation of the original value's taint. When data is read from the shared preferences service, the corresponding taint is also read and the return value is tainted with the taint information stored in the shared preferences. Care must be taken to avoid side effects, i.e., that artificial taint key conflict with existing keys or are considered in any operation on the key set, (e.g., when counting or iterating over entries). This is achieved by respective instrumentations of the `getAll`, `getX`, and `contains` methods. As a result, this allows precise and robust taint propagation over the external service which guarantees that taint states are persisted along with the actual contents.

### 4.6. Tainting files

File I/O would be an easy way to break taint analysis if it is not explicitly covered by the DTA. We thus store taint flags along with files using extended attributes of the file itself. Extended attributes (xattr) are not accessible through the Android API but can be used by native code if the file system supports them. Today, most Android devices use ext4 as a file systems as it is the recommended standard for versions since Android 2.3 and thus support the xattr library. Old versions until Android 2.3 used YAFFS2 and some older models featured f2fs. The drawback of this approach is that the xattr library cannot be used to store taint information in external storage in the default Android environment due to security restrictions. Nevertheless, this solution allows us to store file-level taint which is the best possible accuracy without modifying and parsing the whole file on every read- or write-operation.

### 4.7. Flow-sensitive sources and sinks

As pointed out in section 3, sources and sinks require flow-sensitive definitions spanning multiple statements, i.e. the problem of detecting a source and sink reduces again to a taint analysis.

One approach would be to include the source and sink definition into the actual DTA and treat "potential" source and sink tracing analog to the actual taint tracking. This leads to high precision but significantly increases the size of the global taint table, as multiple tentative taint flags would have to be maintained for a single register, depending on the number of potential sources and sinks it is a part of.

As typically source and sink definitions span several statements and only few methods, we identify sources and sinks by a simple static interprocedural data flow analysis which is applied during the instrumentation phase and therefore does not impact runtime performance. The specification of sources and sinks in our framework is thus either a regular expression identifying a single method call, or a program slice against which the control flow graph (CFG) of each method is evaluated before the actual instrumentation.

## 5. Prototype Implementation

With a prototype implementation of our approach we show that application-level DTA can be realized at a precision level which is competitive to TaintDroid.

### 5.1. Bytecode Modification

The instrumentation adds three new classes to an app: `TaintLabel` holds the representation of taint labels of an object. A `DynamicTaintLabelManager` class stores taint labels in a global table and provides methods to serialize taint labels for sending them across files and ICC channels. `TaintPropagationLogic` holds the representation of the configurable taint propagation logic. A `TaintPropagationWrapper` traces taints of registers and is called from code injected into the original app (Listing 4).

Listing 4. Taint tracking of a variable ($r1) written to a sink (sendTextMessage)

```
$r3 = SmsManager.getDefault();
// Injected: trace taint of message content and handle sink
TaintPropagationWrapper.handleTaint($r1);
// Actual sink
r3.sendTextMessage("1234567890", null, $r1, null, null);
```

## 5.2. Taint propagation logic

Before each statement in the original application, a call to the `TaintPropagationWrapper` is injected into the app, handling the taint effects of the original statement. It takes as input the registers which are used and defined in the statement, as well as the type of the statement, and applies the taint propagation logic. String objects are treated in a special way in that we represent them by their actual content, instead of their memory address. This complies with the Android (and Java) behavior of string interning which leads to the fact that `"x".equals("x") == true` although the memory address of the string objects might differ. Primitive types could be handled either accordingly by their content or boxing them into a complex object. However, in our prototype implementation, we deliberately chose to not trace taint states of primitive types as we found that in most cases private information is represented in complex objects and tracing primitive types adds unnecessary overhead and leads to false positives. Handling the generic $GEN$ and $KILL$ rules is straightforward: for assignment statements, the union of the taint flags in the right hand side is assigned to the left hand side register, whereas the taint of an object is the union of the taint flags of all its attributes. Control flow instructions do not affect the taint flag of the involved registers, i.e., we do not consider implicit leaks.

As for the different ways of handling method calls, the taint propagation logic operates by default according to the heuristics introduced in subsection 4.3:

- Taint flags of return values of method calls propagate to the registers they are assigned to
- If a non-static method is invoked and no return value is assigned, the base register of the method is tainted with the union of the taint flags of the base register and the arguments
- If a static method is invoked, the union of all arguments' taint flags is assigned to all the arguments

In addition to the heuristic, the user can define the taint propagation behavior for specific method calls in a separate XML file. Our prototype declares specific taint behavior for Collections, Objects, Intents, and StringBuilders. Listing 5 illustrates how `Collection.clear()` is handled.

Listing 5. Taint propagation of APIs specified in separate XML file

```
1 <TaintPropagationLogic>
   <class name="java.util.Collection">
3    <entry>
       <method matcher="equals" name="clear" args=""/>
5      <propagation type="REMOVE" target="base" args=""/>
     </entry>
7  </class>
   ...
9 </TaintPropagationLogic>
```

## 5.3. Intents

To send taint flags along with Intents, we use a string-serialized representation of taints, which we collect in a list and attach it to the Intent's `extra` data.

During instrumentation, we search for statements sending intents and inject one statement adding the taint-extra to the intent. To avoid side effects, it is ensured that existing keys are not overwritten and that at the receiving side, a statement is injected that checks if the intent contains the extra, reads it, taints the Intent accordingly and removes the extra data. This way, we transparently transfer taint flags between instrumented apps without the need to modify any underlying ICC mechanism. Taint flags are applied at Intent level granularity, i.e. the implementation does not allow to distinguish different taint flags of possibly complex data structures transferred with the Intent. This is however merely an implementation decision for the sake of performance and could be replaced by storing a precise mapping between taint flags and individual data fields in the Intent.

## 5.4. Shared preferences

Key-value pairs are stored in the `SharedPreferences` service by calling one of the `put` methods of the class `SharedPreferences.Editor`. Whenever such a method is called, we inject a statement that uses the key of the put-operation to derive an own unique key pointing to a value holding a serialized representation of the taint flags of the original value. To prevent side effects, we further ensure that these artificial key-value pairs do not influence the rest of the original application, especially when data is retrieved using the `get` or `getAll` methods.

Wherever the application `gets` a key, we insert a statement to retrieve taint flags from the corresponding artificial key and assign them to the register holding the original value. Where the application uses `getAll` to retrieve all key-value pairs from the SharedPreferences service, we ensure that artificial key-value pairs holding the taint flags do not become visible to the original application. Hence, we get all artificial pairs and assign the stored taint flags to the respective entries in the resulting map. This way, iterating over all keys is possible in the original application and the taint flag management remains hidden from the original code. Dealing with Android system services in such a way is an improvement over TaintDroid, which is not able to detect taint propagation through shared preferences [9].

## 5.5. Files

To store taint flags in extended attributes of files we use Android's native library `xattr`. The most important aspect is identifying the absolute path of the file which is read or written. This is not trivial as the filename is typically passed as an argument to the constructor of a File, FileDescriptor, Writer, or OutputStream object and is not necessarily close to the code location where data is actually written into the stream (or read from it). Thus, to keep track of file names, we treat them like taint labels and assign them to the respective object. This way, File objects can even be passed around across methods without loosing the information on the absolute path of the underlying file which is accessed.

Before each read and write operation, the instrumentation adds statements to assign the taint flags of the read file, or to write the taint flags of the registers being written into it to its xattrs. Performing this on every read and write operation reduces performance of the app but is the most precise way of tracing taint flags because it ensures to have the most recent taint value, even if the file is continuously read and written.

## 6. Evaluation

We evaluated our DTA with respect to precision and the impact on the runtime performance of the analyzed app.

### 6.1. Precision

A precise dynamic taint analysis is expected to be sound, i.e., not report false data flows, and complete, i.e., reliably report each critical data flow. The precision is mainly influenced by the granularity of the taint propagation logic and the capability to observe data propagation across process boundaries at runtime. In both aspects, application-level DTA is conceptually inferior to platform-level DTA, as it neither has the capability to trace data propagation beyond API calls, nor to make observations outside of the current process. We thus evaluated the precision of our DTA from two perspectives: first we tested against DroidBench[2] 2.0, the de facto standard test set of Android apps for benchmarking data flow analysis. At the time of testing, DroidBench comprises a set of 119 Android apps of which 3 are negative samples and 116 are expected to leak data (typically the IMEI or user input) to the log or a text message. DroidBench challenges data flow analysis to cope with various propagation channels and taint granularities, including ICC, reflection, and different stages in the Android application lifecycle. We instrumented all applications with our framework and manually tested them on an unrooted stock Nexus 5X smartphone. All apps could successfully be instrumented and started on the phone. It turns however out that some apps of DroidBench, being focused on static analysis, contain bugs which prevent even the uninstrumented app from performing the actual data leak. For instance, some apps run into out of band exceptions, while others do not allow triggering the leak because it is located in non-exported and unreachable components of the app which cannot be executed. Overall, 18 apps of DroidBench were not suitable for a dynamic analysis. 11 of them do not allow triggering the flow under normal conditions, 7 contained bugs which prevented them from proper execution. We thus ended up with 3 negative and 96 positive test samples. Unsurprisingly, all negative samples were correctly classified as such, i.e. we did not detect non-existing data flows. Further, also all positive samples were correctly detected, i.e. our framework performed with 100 % accuracy against the DroidBench 2.0 test set.

2. https://github.com/secure-software-engineering/DroidBench

| | | Expected | |
|---|---|---|---|
| | | Positive | Negative |
| Actual | Positive | 96 | 0 |
| | Negative | 0 | 3 |

TABLE 1. PRECISION AGAINST DROIDBENCH

While the strength of DroidBench is to challenge a flow analysis in terms of various propagation channels, it contains only small test applications. This does not allow to draw any conclusions on the practical effectiveness of a taint analysis framework from its precision against the DroidBench set. As our goal was not only to create a precise, but even more a practical applicable DTA, we further evaluate how our framework performs against a set real-life apps.

In contrast to platform-level tainting, an instrumentation-based approach like ours is affected by the complexity of an application. Thus, we take TaintDroid as a benchmark for our approach, knowing that it has a conceptual advantage which we aim to minimize as far as possible. To compare our solution with TaintDroid, we first statically analyzed the 10.000 most popular apps from Google Play to identify apps with potential data flows leaking the IMEI of the phone, i.e. apps which contain the relevant method calls. We reduced the set to apps which were still able to run on the latest platform supported by TaintDroid (i.e., Android 4.3). From this candidate set we randomly chose further apps and tried to confirm the data leak using TaintDroid. The result was a test set of 25 apps which reliably leaked data and were detected by TaintDroid. Then, we tested these apps against our solution by instrumenting them and interacting with the instrumented version in the same way as with the original one. The test results[3] are given in Table 2. One out of

| Package name | Flow detected | App runs stable |
|---|---|---|
| cn.menue.heart.activity | Yes | Yes |
| c.appfusion.funnyringtones | Only source | Yes |
| c.appredeem | Yes | Yes |
| c.boyahoy.android | Yes | Yes |
| c.chatroulette.snapchat | Yes | Yes |
| c.divum.MoneyControl | Yes | No |
| c.fry.promi | Yes | Yes |
| c.gameforge.xmobile.middleages | Yes | Yes |
| c.gameloft.android.ANMP.Glof.. | Yes | Yes |
| c.heyzap.android | Yes | Yes |
| c.honestwalker.kancart.DHgat.. | Yes | Yes |
| c.mobage.ww.a692.Bahamut_An.. | / | Instr. failed |
| c.movile.wp | Yes | Yes |
| c.nobilestyle.android | Yes | Yes |
| c.smartphonereligion.whatsap.. | Only source | Yes |
| c.softonic.moba | Yes | No |
| c.viaden.yogacom | Yes | Yes |
| de.visionmedia.maedchenvie.. | Yes | Yes |
| aditor2.aditor.de | Yes | No |
| avisador.de.radares.moviles.fijos.. | Yes | Yes |
| com.advertapp | Yes | Yes |
| com.baiwang.colorphoto | Yes | Yes |
| c.cootek.smartinputv5.skin.them.. | Yes | No |
| com.joycity.god | Yes | Yes |
| com.topmobileringtones.gangst.. | Yes | Yes |

TABLE 2. EVALUATION RESULTS

3. App names truncated for the sake of space

25 apps could not be instrumented successfully because it was lacking required classes – a situation probably caused by wrongly configured obfuscation/optimization which prevents the Soot framework from properly resolving the class hierarchy, but is also prone to runtime instabilities in the non-instrumented app. Out of the remaining 24 apps, we detected the critical data flow in 22 apps. For the remaining two, our solution successfully detected the source of the data flow but was not able to detect the sink. A manual inspection revealed that the cause for this false negative error is that these apps encode the tainted data in primitive types, which we deliberately do not track in our prototype implementation. For 4 of the 24 instrumented apps we could detect a leak, but the app did not run stable.

As for soundness, we argue that the injected taint propagation instructions from section 4.3 do not alter control flow and do not modify contents of existing registers, but rather track taint states in a global table. Taint propagation over external services actually adds persisted entries, but also ensures that any access to these services is free of side-effects by removing artificial taint entries before each operation. The same applies to taint propagation over files because meta data are not passed on to the application. A conceptual boundary of our approach would be reached for applications loading and verifying their own bytecode at runtime – a behavior we never experienced in the wild.

### 6.2. Performance

An upper bound of performance deterioration is doubling the number of executed statements in the worst case. To assess the actual performance impact of our DTA, we injected statements into a self-created small app consisting only of statement which are relevant to the data flow – this is the worst case in terms of performance impact. The time from start to data leak of the application increased from 299 ms for the original execution path to 429 ms of the instrumented application. Further, the instrumentation of the app massively influences the consumption of memory by the app as for every object, the taint needs to be stored. Again, we tested the DTA framework against two different apps. For this evaluation, our small test app is not representative and lead to only 3.5% of additional memory. For a randomly selected larger app (5.26 MB) from the Play store, the number of objects has almost doubled after some time of execution and the allocated memory has increased by 87%.

## 7. Conclusions

Our work is motivated by the search for a pure application-level dynamic taint analysis which is not limited to specific platform modifications and can still keep up with the precision of platform-level DTA. Conceptually, our DTA framework is applicable to all recent Android platforms, which is a clear improvement over TaintDroid, which covers only about 18.6% of the available platforms. As expected, our DTA does however not reach the same precision as TaintDroid. Nevertheless, the fact that we achieve a detection rate of 100 % against the artificial DroidBench set and are able to detect 91.7% of the flows found by TaintDroid in real applications is encouraging and shows that our approach is a viable solution for high-precision application-level DTA, which can be applied to large amounts of applications without limitations on the platform.

## References


[1] S. Arzt, S. Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. Technical Report TUD-CS-2013-0114, EC SPRIDE, May 2013.

[2] G. Ascia, V. Catania, R. D. Natale, A. Fornaia, M. Mongiovì, S. Monteleone, G. Pappalardo, and E. Tramontana. Making android apps data-leak-safe by data flow analysis and code injection. In *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 205–210, June 2016.

[3] M. Backes, S. Bugiel, and S. Gerling. Scippa: system-centric IPC provenance on Android. In *30th ACM Annual Computer Security Applications Conference (CCS)*, pages 36–45. ACM, 2014.

[4] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Ruhr-University Bochum, System Security Lab, Apr. 2011.

[6] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, volume 31, 2011.

[7] 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 real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, Mar. 2014.

[8] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS)*, volume 15, page 35, 2011.

[9] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC, pages 51–60. ACM, 2012.

[10] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. Combining static and dynamic data flow analysis: A hybrid approach for detecting data leaks in java applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC, pages 1573–1579. ACM, 2015.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[12] J. Schütte, D. Titze, and J. M. D. Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *International Conference on Trust, Security and Privacy in Computing and Communications*, TrustCom, pages 370–379, Sept 2014.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, S&P, pages 317–331, 2010.

[14] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS, pages 259–270. ACM, 2014.