AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps

Conference Paper · October 2014

DOI: 10.1109/TrustCom.2014.48

3 authors:

Julian Schütte
Fraunhofer Institute for Applied and Integrated Security
73 PUBLICATIONS **374** CITATIONS
SEE PROFILE

Dennis Titze
Fraunhofer Institute for Applied and Integrated Security
15 PUBLICATIONS **76** CITATIONS
SEE PROFILE

José María De Fuentes
University Carlos III de Madrid
76 PUBLICATIONS **501** CITATIONS
SEE PROFILE

AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps

Julian Schütte, Dennis Titze, and J. M. de Fuentes
{schuette,titze}@aisec.fraunhofer.de, jfuentes@inf.uc3m.es

Abstract

As Android is entering the business domain, leaks of business-critical and personal information through apps become major threats. Due to the context-insensitive nature of the Android permission model, information flow policies cannot be enforced by on-board mechanisms. We therefore propose AppCaulk, an approach to harden any existing Android app by injecting a targeted dynamic taint analysis, which tracks and blocks unwanted information flows at runtime. Critical data flows are first discovered using a static taint analysis and the relevant data propagation paths are instrumented by a taint tracking code at register level. At runtime the dynamic taint analysis woven into the app detects and blocks data leaks as they are about to occur. In contrast to existing taint analysis approaches like Taintdroid, AppCaulk does not require modification of the Android middleware and can thus be applied to any stock Android installation. In this paper, we explain the design of AppCaulk, describe the evaluation of its prototype, and compare its effectiveness with Taintdroid.

1. Introduction

With Android apps being increasingly used for business purposes, the threat of sensitive data leaking from a smartphone becomes prevalent. For example, we applied a dynamic taint analysis to the 10,000 most popular Android apps (taken from the Google Play Store, as of December 2013) and found out that every twentieth app sends out the phone's IMEI immediately at startup. However, the actual amount of apps leaking further personal information under some conditions at runtime is potentially much higher.

The Android permission model is not designed to cope with unwanted information flows. Though it allows fine-granular access control to APIs, it is context-insensitive and thus does not allow the user to specify which type of data may be sent to a specific API.

To cope with information leaks, several approaches have been proposed and some practically applicable solutions exist. Most of them refer to container-based approaches where either applications are wrapped in a "security container" or domains are isolated at kernel level (see [13], [4]). These approaches are however *context-free*, as they do not keep track of individual data flows but rather apply a perimeter security, either at API or OS level.
Dynamic taint analysis, in contrast, monitors how data is handled by an application and detects when an unwanted flow from a specific data source (e.g., the address book) to a specific sink (e.g., a socket) is about to occur. The most prominent representative of this class is Taintdroid [7], a system which applies a modified Android middleware to trace data flows during execution of an app. While Taintdroid is able to reliably detect data flows as they occur, its requirement to modify the Android middleware makes it unattractive for the average business user, as she would have to replace her original Android image with a custom Taintdroid-image and thereby void guarantees.

To overcome this limitation, we present an approach for tracing unwanted information flows in Android apps, which can be applied to any app and does not require modifications to the underlying middleware, thereby making it applicable to every stock Android installation. Our approach aims at weaving a targeted dynamic taint analysis into the bytecode of an app. The weaving is guided by a static data flow analysis which first identifies paths of potentially unwanted data propagation which will be subject to the instrumentation. In combination with a simple policy language, our approach becomes extensible and can be configured to detect different types of data leaks.

To summarize, our contributions are as follows:

- A policy language to specify illegitimate data flows, the taint propagation logic, and countermeasures to be taken once a data leak is detected at runtime.
- A policy-driven bi-directional static data flow

analysis of Dex bytecode, considering border cases like exception handling and data propagation over resources outside of the actual bytecode.
- A Dex bytecode instrumentation which injects a dynamic taint analysis along the previously identified data propagation paths.

To evaluate our approach AppCaulk, we applied AppCaulk to the most popular Android apps and assessed its effectiveness against a manual dynamic taint analysis using Taintdroid. As a result, AppCaulk was able to find and fix the majority of data leaks (approx. 80%) in a non-assisted manner.

The remainder of this paper is structured as follows. In Section 2 we first give an overview of our approach. In Section 3 we describe the data flow policy and in Section 4 how it controls the static data flow analysis, including taint propagation over resources outside of the actual bytecode. Its results guide the injection of a dynamic taint analysis, which is described in Section 5. Section 6 presents the results of the practical evaluation, Section 7 reviews work related to ours, and Section 8 concludes the paper.

2. Approach overview

The aim of this work is to discover and block data leaks in Android apps in hybrid way, i.e. by combination of static and dynamic techniques. In the following, we regard an Android application as a directed graph P of basic blocks, where each basic block is made up of $n \geq 1$ consecutive statements $s \in S$ (S denoting the set of statements). Each basic block ends with a statement where the control flow diverges, for instance a conditional branch. Statements relate to single lines of bytecode, starting with an *opcode* and followed by a number of opcode-specific registers or constants.

A potential data leak is imposed by a directed, noncyclic subgraph of P, leading from a *source* $i \in S$ of sensitive information (e.g., reading from a contact data provider) to an untrusted *sink* $o \in S$ (e.g., sending data to a server).

In order to discover data leaks at runtime, a naive approach of applying a dynamic taint analysis would be to wrap every single statement of the application with monitoring code to track the status of all variables. This would obviously result in a drastic performance deterioration, because the number of statements would be more than tripled and we would have to maintain the taint state of every single variable in the application. Luckily, in most cases only a fraction of all possible execution paths and variables will be relevant for the data leaks to control. We take advantage of this fact by first performing a static inspection of the application bytecode to choose only relevant code fractions and, afterwards, insert the minimum subset of instructions to counter data leakages.

The first task is to define what is the data leak to be detected. This can either be direct, if the respective information is sent out to an untrusted sink as a whole, or indirect, in case the information is for example sent to an intent which is received by a different part of the app. Since data leakages can happen in very diverse ways, we provide a configurable policy to characterize them. Therefore, the first step is to define the leak (and required countermeasures) using our policy. Such a policy will govern the next steps of our approach, particularly the data flow analysis and the instrumentation.

After the policy description, the application itself is analyzed. First, the application is disassembled and divided into the main execution blocks. Based on these blocks, the data flow analysis is performed. For this purpose, the analysis traverses the code twice – in a backward slicing step, we determine the execution paths leading to the sinks and eliminate statements not relevant for data propagation to the sink. In a second step, we apply a forward slicing to further eliminate all paths not starting at sources. To perform these actions, a propagation logic is defined. This logic is composed of rules that specify for each opcode how sensitive data is propagated from one register to another. Therefore, these rules are based on the well-known *taint propagation* concept [12], specifying how registers get *tainted* or *untainted*. To offer full flexibility, user-defined propagation rules can also be included into the aforementioned policy. They may also contain different *taint levels*, thus enabling classifying data by different protection levels such as “personal information” or “confidential”.

The next step is to instrument the program in a way that it tracks such flows and handles data leaks. Nevertheless, as this activity depends on the register type of the leaked data, a type inference task is carried out. Using this information, the instrumentation injects the appropriate action into the bytecode just before data from a tainted register is about the leak to a sink.

After the instrumentation step, the application is hardened and can be run on Android devices without the fear of data leakages. For this purpose, the modified application code is assembled, packed and signed.

3. Policy definition

The main degrees of freedom in data leak prevention are the definitions of “unwanted data flows” and the countermeasures to be taken. They cannot be set universally, but should rather be available in the form of profiles, a user can choose from. We thus apply a policy-based approach which allows to describe any use-case-specific information in the form of rules:

Sources. Sources of sensitive information, denoted by statements indicating either program entry points or method invocations. For method invocations, each return value is considered tainted using the configured tag. We allow any substring of the fully qualified method name and its parameters here. For example:

```
Landroid/telephony/TelephonyManager;->
    getDeviceId()Ljava/lang/String;[TAG_IMEI]
```

Sinks. Untrusted sinks to which sensitive data must not leak, denoted again by statements of the respective method invocations, and the index of parameters to which tainted data must not leak. For example:

```
Lorg/apache/http/client/HttpClient;->execute(
    Lorg/apache/http/client/methods/
    HttpUriRequest;)Lorg/apache/http/
    HttpResponse; [0]
```

Tags. A lattice of a set of tags to classify tainted data. For example:
`TAINT_IMEI`, `TAINT_GPS`, etc.

Propagate. A function defining for each statement how tags are propagated from input registers to output registers, denoted by the register's position. E.g.:
`aget: 0:L ← 1:L|2:L`
(`aget vx,vy,vz` gets the object at index `vz` from the array `vy` and puts it into `vx`. The propagation rule states that when either the index or the array is tainted with taint level `L`, the output will be tainted as well)

Untaint. A function asserting for each statement how tags are untainted. This is analogue to setting the target taint level to ∅ in a *propagate* rule.

APIPropagationRules. Not all functions can be completely examined when only operating on byte-code, since at some level, they can use native code. These rules specify how tainting is propagated for such functions. If an API function is not configured, but cannot be analyzed due to native code usage, our framework assumes all registers as tainted. Therefore, all potential data leakages through native code are considered.

Countermeasure. The action to be taken when a data leak has been detected at a sink, denoted by its fully qualified method name. The method's parameters are not given here as they are determined by the `CounterMeasureHandler` interface and filled with relevant data such as the current program counter, the tainted register, taint level, etc. For example:
`<sink>:L AskForUserConfirmation`
(When method `<sink>` is about to be called with a register having taint level `L` or higher, a method to open a user confirmation dialog is invoked. If the method returns `true`, execution continues, otherwise the program exits)

Such rules can be bundled in profiles, selectable by the user, such as “prevent personal information leakage to the Internet” or “require encryption for persistent storage” and allow us to adapt the propagation logic to capture even stricter variants, such as control- flow-based information leaks.

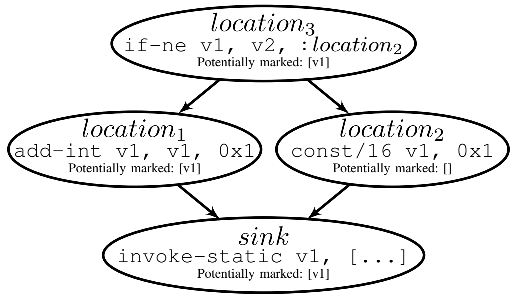


Figure 1. Potential taint propagation to a sink

4. Static data flow analysis

Given the information stated in the policies, we can now apply a static data flow analysis, whereas we have to consider some challenges specific to the Android OS.

We apply a bi-directional inter-procedural data flow analysis and slice the program code accordingly to extract only the statements which are relevant for data propagation from sources to sinks. During the analysis, we also consider propagation channels which lie outside of code execution, such as propagation over files or sockets. We refer to these situations as *external tunnels*. Section 4.2 describes how we deal with them.

4.1. Backward Slicing

The goal of the analysis is to annotate each location (i.e., line of code) in each function with a list of registers which could potentially leak their values to a sink, if the respective code path is executed.

Backward analysis. The analysis uses a worklist algorithm, similar to [6], to create a list of relevant registers at each location, starting at all sinks which are defined in the policy, such as

```
Lorg/apache/http/client/HttpClient;->execute(
    Lorg/apache/http/client/methods/
    HttpUriRequest;)Lorg/apache/http/
    HttpResponse;[0]
```

The policy also defines which sink parameters must be considered as leaks (parameter 0 in this example, referring to the `HttpUriRequest`) – they are initially marked as *relevant*. We now step back in the control flow graph (CFG), marking further registers with their taint status according to the propagation logic defined in the policy. For instance, for the instruction `move v1, v2`, the propagation logic assigns the taint state of the source register `v2` to the destination register `v1`.

Whenever the register is untainted on an execution path, we remove it from the list of relevant registers. Consider the simple control flow depicted by Figure 1: at the sink, register `v1` is marked as relevant, but as we step backwards in the CFG, $location_2$ assigns a constant value to it, thereby rendering it irrelevant for data leaks.

During the backwards analysis, we create the CFG on-the-fly and follow all function calls available in the classpath. Besides the application code, this includes all Android APIs and their implementations in the `framework.jar` file. However, some Android APIs (and in some cases, application methods, too) are implemented as native code and invoked over JNI. As these methods cannot be further analyzed, we cover them by fixed propagation rules defined in the policy.

As for Android 4.3, we identified 3600 natively implemented framework functions. The majority of them is however not relevant to data flow tracking, as they either take no parameter or do not return a value. Thus, 1339 native framework functions remained to be covered by propagation rules. They include many overloaded and thus redundant functions, differing only in their parameter types, such as `sqrt(D):D` and `sqrt(F):F`, thus making the definition of fixed propagation rules feasible.

Apart from straightforward taint propagation over local registers, some further cases have to be handled:

- *Member variables:* If a member variable (static or not static) is read at a location (e.g. `sget v1, L/[...]->MEMBER`), this member variable is considered relevant, and all locations where this variable is written are added to the worklist.
- *Wide operations:* Dex bytecode includes several 64 bit instructions operating on two successive 32 bit registers. These operations are only called with one part of the register pair, while the second one is implicitly used. For example `move-wide v1, v3`), applies to four registers from `v1` to `v4`. These instructions must thus be treated respectively in the propagation logic.
- *Coupled locations:* Some atomic operations comprise multiple consecutive statements. For example, method calls consist of one statement for the call and a second one for assigning the return value to a register. This has to be reflected when parsing statements during the analysis.

The backward analysis will terminate when no statement has been added to the worklist and no taint marking has changed in the previous iteration. At this point, we have a slicing of the program containing all control flows along which data may be propagated to a sink.

Forward analysis. To ensure that only *critical* data is considered, we now apply a forward analysis starting at all sources. Here, the source definition from the policy is used, which contains critical method calls and the taint level with which the return value of the call will be tagged. For example, the following line assigns the taint tag `IMEI` to return values of the method `getDeviceId()`.

```
Landroid/telephony/TelephonyManager;->
    getDeviceId()Ljava/lang/String;[TAG_IMEI]
```

Creation of annotations. Finally, we merge the results of forward and backward analysis and create *annotations* for each relevant line of code, which guide the subsequent instrumentation.

The merging algorithm for a simple case takes the result of the forward and backward analysis as input, and considers all locations which appear in both locations. The algorithm then states if the line can mark or unmark variable, and stores this information in an internal data structure, which is used by the instrumentation in the next step.

In reality, the merging algorithm has to deal with some more complex issues, which are only briefly explained here:

- *method calls:* tainted parameters need to be marked inside a method before the first statement
- *method returns:* taint states of return values need to be mapped to the registers to which the return values are assigned in the caller.
- *member variables:* tainted member variables have to be stored and and read on a per-instance basis.
- *exception:* taint states may propagate over exception handlers

4.2. Addressing external tunnels

The described forward and backward techniques rely on the logical path that may be traced for a given piece of information throughout the analyzed application. Apart from propagating register values, there may exist additional paths to forward critical data from sources to sinks, which we aim to capture. We refer to these paths as *external tunnels*.

External tunnels may rely on files, databases, content providers, SharedPreferences containers, intents and broadcast receivers (Figure 2). In order to build such a tunnel, it is necessary to first send the data to be leaked to the external instrument (e.g., a file). Afterwards, in any other part of the application, an information retrieval is performed. To the best of our knowledge, many of these data leaks have not been addressed previously.

In order to address external tunnels, we extend the CFGof the aforementioned analysis. In the case of files, whenever a file write with a tainted value is found in the forward analysis, all file reads in the whole application are added to the list of instructions to analyze and the result of the file reads are potentially tainted as well. Similarly, if a tainted file read is found in the backwards analysis, all file write operations are queued to be processed and their parameters are tainted accordingly. The same process is applied for databases, connecting $insert$ and $query$ operations. Also, the same methods are connected when they are related to content providers and a tainted $ContentValue$ is at stake. With respect to the shared preferences file ($SharedPreferences$ class), the methods involved in external tunnels are put and $getString$ (and any other get-* method defined for the class). Regarding intents, the methods $startActivity$ and $getIntent$ are linked to each other.

One special situation is that of broadcast receivers ($BroadcastReceiver$ class), where we consider $sendBroadcast$ as the insertion operation. As opposed to the previous cases, however, no predefined retrieval operation exists. Rather, once a broadcast receiver is gathered by the application, the $onReceive$ method is invoked and data is retrieved by methods which are specific to the implementing class. Therefore, we assume the first line of the $onReceive$ method as the retrieval operation and add it to the processing queue when a $sendBroadcast$ has been found before. With respect to the backwards analysis, whenever the process reaches the beginning of the $onReceive$ method, all $sendBroadcast$ operations are added to the list of instructions in which the analysis will continue.

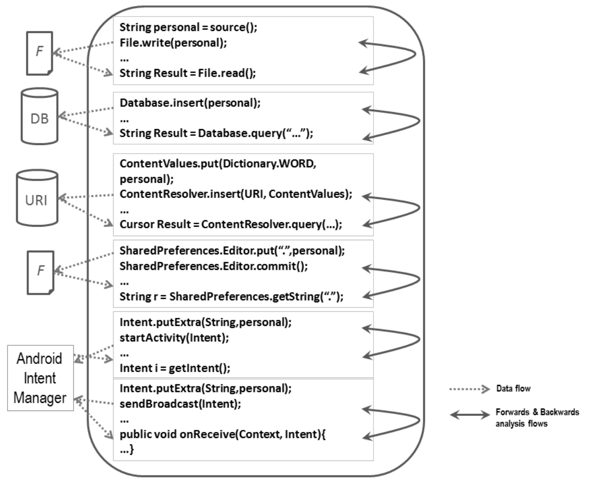

```
String personal = source();
File.write(personal);
...
String Result = File.read();
```
```
Database.insert(personal);
...
String Result = Database.query("...");
```
```
ContentValues.put(Dictionary.WORD,
personal);
ContentResolver.insert(URI, ContentValues);
...
Cursor Result = ContentResolver.query(...);
```
```
SharedPreferences.Editor.put(".",personal);
SharedPreferences.Editor.commit();
...
String r = SharedPreferences.getString(".");
```
```
Intent.putExtra(String,personal);
startActivity(Intent);
...
Intent i = getIntent();
```
```
Intent.putExtra(String,personal);
sendBroadcast(Intent);
...
public void onReceive(Context, Intent){
...}
```

Figure 2. External tunnels management

One limitation of the approach is its heuristic nature when addressing files. Precisely tracking file handlers is not possible at bytecode level, as files can be moved or modified outside of the application's bytecode, e.g. by native code or the OS itself. Therefore, we overapproximate by considering every file read after a file write has occurred, not differentiating between individual file names. Although this may lead to false positives, this decision ensures that no potential data leakages are missed.

5. Bytecode instrumentation for dynamic data flow tracking

As a result from the static analysis, we receive all statement locations and relevant registers, along with the registers' taint states that are on the paths from all sources $i \in S$ to each sink $o \in S$ and may have an effect on data propagation from source to sink. That is, we create a set $T \subseteq S$ of taintable statements, such that applying the propagation logic to each $t \in T$ results in a propagation path from i to o, no statement $t' \notin T$ exists on any propagation path. We are now ready to instrument the application so it includes a taint analysis along these paths, and only these paths.

In this section, we describe the procedure of Dex-bytecode instrumentation and the challenges to overcome.

5.1. Type Inference

To correctly interpret the registers during taint propagation, we have to infer their types to distinguish between *reference* and *non-reference* registers. This distinction is required in order to construct valid statements that will pass Android's bytecode verifier. Inferring the type of a register is trivial for a statement like the following, which reveals that v2 is an integer and v3 is a string.

```
invoke-static {v2, v3}, L/Testing;->test(I L/
    java/String;)V
```

Dex bytecode is not strictly typed and thus, type information must be inferred from a statement's context. For example, the following statement is type- sensitive at the time of execution, but does not reveal any typing information about v2 or v3 at bytecode level:

```
move v2, v3
```

To solve this problem, we infer types from use-definition chains created during the intraprocedural static analysis. For each method we first run from the return point to the start of the method and remember all register types. At the method start, we infer the remaining register types from the method header.

5.2. Runtime stack tracing

In order to achieve a precise context-sensitive data flow tracking, we need information about the instance of a method's caller at runtime.

Unfortunately, Dalvik does not include information about instances in the stack trace. Although it is possible to inspect a stack trace, references to the specific instances are not made available by the Dalvik VM and it is therefore not possible to determine whether a method has been invoked from an instance A or B of the same class.
As this information is required for the context-sensitive data flow analysis, we instrument the app's bytecode, so as to create a custom stack trace which includes all information we need. Along the critical data paths and for each thread, we create a global custom stack and push the caller's instance onto it just before a method call. After returning from the call, the caller's instance is removed again from the stack.

We then use this detailed custom stack in our tracing code and check whether the parameters of a method have been tainted in the caller's instance.

5.3. Register count

Dalvik is a purely register-based machine, so we have no choice of allocating additional memory for the dynamic tainting analysis, other than reserving further registers. This needs to be done at the time of instrumentation, because Dalvik does not allow to dynamically allocate registers at runtime. In practice, this leads to some difficulties, as many opcodes can only reference registers with 4-bit addresses, so when extending the number of registers in a method, we need to ensure that we comply to the opcode's restrictions. We therefore rewrite the respective statements and either replace the opcode with a semantically equivalent one, which supports registers beyond 4 bits (e.g., `invoke-direct` is replaced by `invoke-direct/range`), or swap the relevant registers with some of the lower 4-bit range.

5.4. Injecting dynamic taint analysis

In order to trace the taint state of each relevant register, we track them in a global tainting table. This is done by inserting a call to our taint tracking code before or after each statement in the *data flow graph* (DFG) and passing all relevant registers to it. The DFG is the result of the previous static analysis. Depending on the type of annotation for the location, the call to our code needs to be inserted before or after the statement.

In the taint tracking code, we then turn each register into a globally unique identifier and set its taint state as indicated by the static data flow analysis. This includes the tag that is given to a variable at each sink. Using this tag, we can e.g., display the type of data that is about to leak.

The taint tracking code also requires the results of the type inference: when passing registers to the taint tracking code, we need to know the type of the register in order to call the tracking method with the appropriate signature. While most non-matching types could be handled by proper casting in the taint tracking code, Dalvik distinguishes between ref and non-ref registers at bytecode level (i.e., references and primitive types). Correctness of these types is checked during the pre-verification in the bytecode verifier, which is invoked by the main ClassLoader already at the time of installation. So, it is not possible to handle mismatching types at run time in our code, as the application would not even install. Further, the alternative of switching off the verifier is not an option for our approach, since we aim at a solution that does not require any modification of the Android platform.

Since we have added the above-described type inference into the static analysis phase of our framework, we can now use this information to properly distinguish between ref and non-ref types.

5.5. Handling policy violations

Just before the call to a sink method, a handler is inserted that checks the current taint status of the parameters that are passed to the sink and takes countermeasures, if a tainted value is about to leak.

Of course, it is possible to simply prevent the sink from being called or to terminate the program. However, in order to increase user experience and stability of a hardened app, it is helpful to provide access to the GUI context of the application. This way, a policy handler can display dialogs and ask the user for confirmation. This requires the handler to access the application context object, which is not accessible from a static method. We therefore instrument the main activity of the app so it provides static access to `getApplicationContext()`.

6. Evaluation

AppCaulk has been implemented as a Python-based tool, which analyzes and modifies Android apps and runs on a standard Linux system. The input to AppCaulk is an Android APK file, which is statically analyzed and subsequently instrumented according to the findings from the data flow analysis, as detailed in Section 4 and Section 5.

In a first step of our evaluation, we implemented a simple app with some exemplary data leaks. This toy example includes reading the IMEI and telephone number and leaking them to an online server, under certain conditions. The data propagation path includes methods of the app itself, the Android framework, exception handlers, and file read/writes.

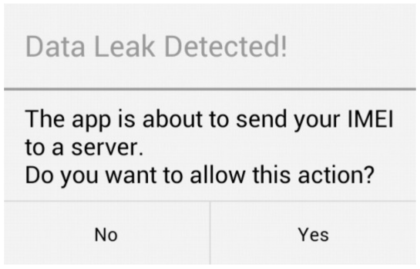
```
source {
      Landroid/telephony/TelephonyManager;->
          getDeviceId()Ljava/lang/String;
            [TAG_IMEI]
      Landroid/telephony/TelephonyManager;->
          getLine1Number()Ljava/lang/String;
            [TAG_PHONE]
      ...
}

sink {
      Lorg/apache/http/impl/client/
          DefaultHttpClient;-><init>[0]
      ...
}
```

Listing 1. Example policy

The policy has been set up to prevent such data leaks, and includes the exemplary sources and sinks shown in Listing 1.

The propagation rules are configured in the policy as described in Section 3. As a countermeasure, `AskUserConfirmation` is used, which checks at each sink if the specified parameter is tainted, and only if this is the case, displays a GUI dialog to ask the user for confirmation, as shown in Figure 3.


Data Leak Detected!

The app is about to send your IMEI to a server.
Do you want to allow this action?

Figure 3. Injected AskUserConfirmation Message

Since we use taint tags to track the type of data being leaked we know that the app is about to leak the IMEI and can inform the user accordingly.

6.1. Discussion of Performance Impact

One of our goals when combining dynamic taint analysis with a static data flow analysis was to limit the overhead caused by instrumented statements. Assuming that we had followed a naive approach, we had to instrument each statement with the taint tracking code, had to put every single method call onto our customized runtime stack, and had to rewrite all statements accessing registers which had moved beyond the 4 bit range during the instrumentation. Only adding the taint tracking code would have increased the bytecode by factor three (storing the current method, the current register, and invoking the tracking code). Pushing and popping caller objects from the stack would again add two statements per method call. Further, the rewriting of registers can take up to eight additional statements per instruction, when registers have to be swapped.
Thus, we would have ended up with an application which would be at least three to five times larger than the original one – obviously not a desired outcome.

Limiting the instrumentation to only those statements that have been identified as relevant during the static data flow analysis results in a drastic improvement, as illustrated by the example of our toy app. Although being very small, this app serves well as a benchmark, because it can be considered the worst case in terms of overhead, doing almost nothing besides leaking data. The app contains 371 statements, which all would be instrumented if no a-priori knowledge of possible data flows would exist. Since we perform such analysis, only 39 statements need to be instrumented. This means, we can leave more than 89% of the original statements untouched in our testing app, which results in no noticeable performance impact when it is executed. Since we only instrument paths that can potentially leak data, all other paths are not influenced.

Tested applications				AppCaulk	Taintdroid
Package name	Sources	Sinks	Methods	Data leakage?	Data leakage?
com.rhmsoft.fm	0	62	14514	No (no sources or sinks)	n/a
cn.wps.moffice_eng	1	29	39152	*	Not achieved
com.facebook.pages.app	1	4	36171	Yes	Yes
com.mobisystems.office	2	30	48308	Yes	Yes
com.allesklar.job	1	26	3492	No	Not achieved
de.arbeitsagentur.jobboerse	0	0	7587	No (no sources or sinks)	n/a
com.dataviz.docstogo	2	22	17748	Yes	Not achieved
com.estrongs.android.taskmanager	1	10	1647	No	Not achieved
com.netqin.ps	7	43	14355	Yes	Yes
com.indeed.android.jobsearch	0	8	2777	No (no sources or sinks)	n/a
at.tomtasche.reader	0	19	8260	No (no sources or sinks)	n/a
com.splashtop.remote.pad.v2	0	9	13560	No (no sources or sinks)	n/a
com.box.android	1	53	37023	No	Not runnable in emulator (crash)
com.dynamixsoftware.printershare	3	55	13170	Yes	Not achieved
com.allesklar.lehrstellen	1	34	6152	No	Not achieved
com.monster.android.Views	0	10	13101	No (no sources or sinks)	n/a
com.mobisystems.editor.office_with_reg	2	30	48709	*	Not achieved
mobi.infolife.smsbackup	0	3	1265	No (no sources or sinks)	n/a
de.joergjahnke.documentviewer.android.free	0	14	2436	No (no sources or sinks)	n/a
cn.wps.moffice_i18n	1	27	39649	*	Not achieved
com.olivephone.edit	1	43	48811	No	Not achieved
com.scout24.jobs	0	6	4925	No (no sources or sinks)	n/a
com.tux.client	0	7	2564	No (no sources or sinks)	n/a
com.rhythm.hexise.task	0	9	1128	No (no sources or sinks)	n/a
com.threebirds.wordreader	1	39	5148	No	Not runnable in emulator (crash)
com.stoik.mdscanlite	2	25	2508	Yes	Not runnable in emulator (not installed)
de.aok.gehaltsrechner	0	14	830	No (no sources or sinks)	n/a
im.ecloud.ecalendar	6	72	20327	Yes	Yes
com.intsig.BCRLite	7	39	11531	Yes	Not runnable in emulator (crash)
com.hi.applock	2	12	9353	No	Not achieved
com.netqin.mm	7	49	9615	Yes	Not runnable in emulator (crash)
com.threebirds.excelreader	1	39	5080	No	Not runnable in emulator (crash)
mobi.infolife.installer	0	30	9981	No (no sources or sinks)	n/a
com.sic.android.wuerth.wuerthapp	1	22	3742	Yes	Not achieved
de.interaid.heizoel24	0	7	4329	No (no sources or sinks)	n/a
com.citrix.Receiver	17	32	27562	Yes	Not achieved
de.sandnersoft.Arbeitskalender_Lite	0	9	3514	No (no sources or sinks)	n/a
com.samapp.excelcontacts.excelcontactslite	4	5	6913	No	Not runnable in emulator (crash)
com.tf.thinkdroid.amlite	1	35	31604	No	Not achieved
soma.de	0	12	3309	No (no sources or sinks)	n/a
com.nitrodesk.droid20.nitroid	5	47	35735	*	Yes
com.cisco.anyconnect.vpn.android.avf	2	2	4229	No	Not runnable in emulator (crash)
com.wyse.pocketcloudfree	0	35	13370	No (no sources or sinks)	n/a
com.zero8timetracking.timesheeter	0	27	16254	No (no sources or sinks)	n/a
com.IQBS.android.app2sd	0	11	1395	No (no sources or sinks)	n/a
com.ups.mobile.android	1	16	20125	No	Not achieved
com.infraware.polarisoffice.entbiz.gd.viewer	1	21	16493	No	Not runnable in emulator (requires PIN)
com.threebirds.officereader	1	43	5088	No	Not runnable in emulator (crash)

Table 1. Effectiveness comparison AppCaulk versus TaintDroid

Besides the actual modification of the app's bytecode, AppCaulk adds a fixed overhead of 1100 lines of code for the tracking and the AskUserConfirmation message.

Compared to a naive approach which would inflate apps to a triple to quintuple of their size, this shows how AppCaulk achieves a much lower overhead.

6.2. Effectiveness analysis

Applying AppCaulk just to our simple example app would obviously not allow us to make any statements about its practical applicability and its effectiveness. We thus applied AppCaulk to the 48 most popular Android applications in the German Google Play market as of July 2013. Given that TaintDroid can also identify data leaks in Android applications, we compared our prototype with this approach.

In order to compare both proposals, the AppCaulk prototype was configured to have $getDeviceId$ as source (which indicated the reading of the device's IMEI). To configure the sinks, the permissionmap [11] was consulted, and all API calls which need the INTERNET permission were considered as sink. In this way, all potential leakages of the IMEI to the Internet are considered. This type of leak is also considered by TaintDroid.

Table 1 summarizes the achieved results. The comparison procedure was composed of two steps. First, all applications were analysed by the AppCaulk prototype. The analysis took 7656,2 s. in mean and four applications did not finish their analysis (noted by * in Table 1). This figure highlights the need to further improve AppCaulk in the future.

The second step was to manually execute them in the TaintDroid environment. In case they did not have any source or sink, they were not considered. This issue happened in 19 out of 48 applications. The goal of this step was to achieve the same result as that of our prototype concerning data leakage detection – if our prototype detected a potential data leakage, it should also be detected by TaintDroid. From the 29 remaining apps only 19 were successfully installed. As 4 of them were not analysed by our approach, 15 cases can be effectively compared. According to the results, most of these applications (12 out of 15) got the same detection result using both approaches and only 3 were false positives. Apart from them, application $com.netqim.mm$ is also successful since AppCaulk detected a data leak which is already announced in its terms of use. It must be noted that no false negatives have been found. Therefore, results confirm the effectiveness of our prototype – no data leakages were missed and a reduced amount of false positives appeared.

7. Related Work

Security research on Android apps has been a popular area in the past few years and a number of publications related to ours have been published. Nevertheless, to the best of our knowledge we are the first to describe the combination of static analysis and injection of a dynamic taint analysis into existing applications.

AppGuard [1] and *Aurasium* [14] aim to bring more fine-granular and customizable security policies to apps. AppGuard revokes permissions of an app by rewriting its bytecode. Aurasium inserts monitoring code into the app and requires modifications of the Android middleware to trace and block the monitored code. *Mockdroid* [3] also modifies the Android OS, but aims at "mocking" data sources with dummy content, rather than blocking access to them.
Just like AppCaulk, *TaintDroid* [7] discovers flows of sensitive data at runtime. In contrast to our approach, it does so by weaving a dynamic taint analysis into the Dalvik VM and other parts of the OS. Consequently, this makes it difficult to apply TaintDroid to selected applications only or in scenarios where installation of a custom TaintDroid-image on a phone is not an option.

SCanDroid [9] builds upon the WALA code analysis framework to run a static data flow analysis and evaluate its results in the context of the permissions required by the application's manifest file. A modification of the bytecode was however not the goal of the authors. It should further be noted that the choice of Wala for static analysis raises some difficulties as Wala operates on Java source code or JVM bytecode only.
A more promising choice is *Soot* [10], which is a program analysis and optimization framework for Java. In combination with *Dexpler* [2], a bi-directional translation from Dex bytecode to the internal representation of Soot, the framework can also be applied to Android apps. *FlowDroid* [8] is a purely static data flow analysis tool including Android-specific propagation channels via callback functions. Our approach differs from FlowDroid in that we amend the static analysis with injection of a dynamic taint analysis so that we are not only able to discover potential data leaks, but also prevent them at runtime. Further, the dynamic part of our solution tracks data propagation even outside of the actual code, such as File I/O. Nevertheless, FlowDroid's callback-aware flow analysis could be combined with AppCaulk's static data flow part.
Though not dealing with Android apps, the work of Chang et al. [5] is the one closest related to ours. The authors propose a framework for statically analyzing C programs and instrumenting the program at locations where policy violations might occur. Chang et al. detail some specific problems and show that their system produces low overhead. In contrast to our framework, they operate at C code and not on the binary itself, thereby making the approach not suited for a fully automated blackbox analysis and instrumentation of third-party applications, as we seek to achieve.

8. Conclusion

In this paper we presented AppCaulk, a framework to harden Android apps by instrumenting them along critical execution paths with a dynamic taint analysis. AppCaulk is able to detect and prevent data leaks by tracing data flows from sources to sinks. Contrary to existing dynamic taint analysis approaches, AppCaulk operates directly on an app and thus does not require any modification of the underlying Android platform. Also, it can be applied to arbitrary third-party applications, as it only requires its bytecode.

Our practical evaluation has shown that AppCaulk reliably detects a high percentage of data leaks while the added overhead is kept low. Hence we conclude that the approach of AppCaulk can be turned into a reliable solution to effectively prevent data leakage along known paths.

As part of our future work, we will extend on soundness and completeness of data propagation paths outside of the actual bytecode, including further improved management of read and write operations on files and sockets, and dynamically loaded code.

References

[1] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky. Appguard - real-time policy enforcement for third-party applications. Technical Report

A/02/2012, Max-Planck-Institut für Softwaresysteme, 2012.

[2] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile, pages 49–54. ACM, 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 51–62, New York, NY, USA, 2011. ACM.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[6] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative data-flow analysis, revisited. Technical report, 2004.

[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 realtime privacy monitoring on smartphones. In *9th USENIX conference on Operating systems design and implementation*, 2010.

[8] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical Report Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[9] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications, 2009.

[10] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS Users and Compiler Infrastructure Workshop*, Oct. 2011.

[11] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. A static analysis tool and permission map for identifying permission use in android applications. http://www.android-permissions.org/permissionmap.html, accessed 10th Apr. 2013.

[12] 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 *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[13] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *Security Privacy, IEEE*, 8(3):36 –44, may-june 2010.

[14] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.