Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android

Vaibhav Rastogi[1], Zhengyang Qu[2], Jedidiah McClurg[3], Yinzhi Cao[4], and Yan Chen[2]

[1] University of Wisconsin and Pennsylvania State University
[2] Northwestern University
[3] University of Colorado Boulder
[4] Lehigh University
vrastogi@wisc.edu, zhengyangqu2017@u.northwestern.edu, jedidiah.mcclurg@colorado.edu, yinzhi.cao@lehigh.edu, ychen@northwestern.edu

Abstract. Mobile devices are becoming increasingly popular. One reason for their popularity is the availability of a wide range of third-party applications, which enrich the environment and increase usability. There are however privacy concerns centered around these applications – users do not know what private data is leaked by the applications. Previous works to detect privacy leakages are either not accurate enough or require operating system changes, which may not be possible due to users' lack of skills or locked devices. We present Uranine[1], a system that instruments Android applications to detect privacy leakages in real-time. Uranine does not require any platform modification nor does it need the application source code. We designed several mechanisms to overcome the challenges of tracking information flow across framework code, handling callback functions, and expressing all information-flow tracking at the bytecode level. Our evaluation of Uranine shows that it is accurate at detecting privacy leaks and has acceptable performance overhead.

1 Introduction

Privacy encompasses an individual's or a party's control of information concerning themselves. With the advent of smartphones and tablets, third party applications play an important role in the lives of individual consumers and in enterprise businesses by providing enriched functionality and enhanced user experience, but have simultaneously also led to privacy concerns. On the consumers' side, how third-party applications deal with the wealth of private data stored on mobile devices is not quite clear. Enterprises, on the other hand, need to protect sensitive business data. With the implementation of Bring Your Own Device (BYOD) policies to better accommodate the needs of employees, the issue is further aggravated, as the business data is stored on devices that are not completely trusted. Leakage of business data to the Internet or from business applications to personal applications is an important concern. Some leakage of private data may be legitimate and even intended; yet, other leakages may be questionable. We therefore believe that information about the privacy leaks

[1] Uranine is a dye, which finds applications as a flow tracer in medicine and environmental studies.

Table 1: Uranine compared with dynamic approaches. + is better, − is worse.

	TaintDroid [13]	Phosphor [4]	Uranine
Real Time	Yes (+)	Yes (+)	Yes (+)
System Modification	Yes (−)	Yes (−)	No (+)
Configurability	Little (−)	Little (−)	High (+)
Accuracy	Good (+)	Good(+)	Good (+)
Performance (runtime)	Good (+)	Good(+)	Good (+)
Portable	No (−)	Yes(+)	Yes (+)

should be completely transparent and accessible to the user (or the IT administrator in case of enterprises). The user may then choose to allow or disallow the leaks either through real-time interaction with an on-device controller or through preset policies. In particular, apart from good accuracy and performance, the detection of privacy leaks should have the following requirements.

- *Real-time.* Real-time detection, or detection immediately when leaks happen, enables situationally-aware decision making. The situation (condition) under which a leak happens is important—a privacy leak may be user-intended, and in that case legitimate. For example, upload of a user's address book to a social network under user's consent is legitimate. Offline detection of leaks may be helpful, but does not usually identify the complete situation under which a leak happens.
- *No system modification.* Mobile devices typically come locked, and it is difficult for an average user to root or unlock them to install a custom firmware.
- *Easily configurable.* The user should be able to enable the privacy leak detection just for the apps she is concerned about. Other parts of the device such as system server processes and trusted apps from the device vendor should be able to run without overhead.
- *Portability.* The framework should work across different devices with potentially different architectures, e.g. ARM and x86, and with different runtimes, e.g. Dalvik and ART (a recently introduced Android runtime[2]), with little or no code modification.

There have been many earlier systems targeted at detecting privacy leaks, but all have some drawbacks with regards to the above characteristics. TaintDroid [13] detects privacy leaks in real-time, but requires the installation of a custom Android firmware, which possibly limits its accessibility to expert users. Furthermore, TaintDroid's firmware code modifications must be adapted to different architectures and even different Android versions. Phosphor [4] is a dynamic taint tracking system for Java which can work on Android. It instruments the application and library code to detect privacy leaks in real-time. However, it requires modification of the bytecode of platform libraries, which again requires custom firmware and hinders wide-scale deployment. Static analysis systems [14, 2] fail on the real-time requirement—inputs from the user or from the remote server may affect what is sent out of the device and thus the leak may or may not be considered legitimate.

In this paper we propose *Uranine*, a real-time system for monitoring privacy leaks in Android applications without the need for platform modification. The major challenge comes from the requirement of *no platform modification*, includ-

[2] https://source.android.com/devices/tech/dalvik/art.html

ing being unable to instrument framework code:[3] we need to approximate flow through the framework code and for *callbacks*, i.e. application code called by the framework code. This is further complicated by the existence of heap objects, which often point to other heap objects and whose effects may easily lead to missing leaks if not handled carefully.

Uranine provides a framework for instrumenting stock Android applications without the need for the application source code. It begins by converting the application bytecode to an intermediate representation (IR), which it instruments employing the techniques presented in this paper. The instrumented IR is assembled back into a new application installable on an Android device. As the instrumented application runs, privacy leakages are automatically tracked.

Apart from being real-time and requiring no system modification, our approach also brings the added benefit of instrumenting only apps that the user is concerned about; the rest of the system, including the middleware and other apps, run without overhead. Finally, since we do not touch the Android middleware and the virtual machine runtime, our approach ensures portability. Table 1 summarizes the comparison between Uranine and other similar systems.

This paper makes the following contributions.

- We solve the problem of tracking private information through platform APIs and libraries without modifying the platform, by developing appropriate data structures and algorithms in Section 3.1.
- The Java language and especially the Android platform relies heavily on callbacks, i.e. functions in app code that are called by the platform libraries. We discuss the challenges of handling callbacks for real-time information flow tracking, and propose the first solution for this problem in Section 3.1.
- Aspects of Java, including reference semantics for objects and garbage collection, pose a problem with regards to developing a clean solution that does not interfere with the Java model. Our solution, centered on hashtables with weak references to hold necessary data-structures for different objects, solves this problem (Section 3.1).
- We have developed a system prototype called Uranine for real-time detection of privacy leakages in Android apps without system modification.

We evaluated (Section 5) a working prototype of Uranine on real-world applications from Google Play. The evaluation shows that Uranine is accurate in tracking information flows. Our evaluation of performance overhead shows that Uranine has acceptable overhead on real-world applications. We also note that it is possible to further reduce the performance overhead of Uranine by performing static analysis and instrumenting only paths along which private information flows can take place.

The rest of this paper is organized as follows. Section 2 gives the background and states our approach together with the challenges involved. A detailed description of the Uranine framework is covered in Section 3, while Section 4 covers the implementation aspects. Section 5 gives our evaluation of Uranine. We then

[3] Throughout the paper, *app code* refers to the code contained in the app; *framework code* refers to the code defined in the Android platform and may be called through Android APIs.

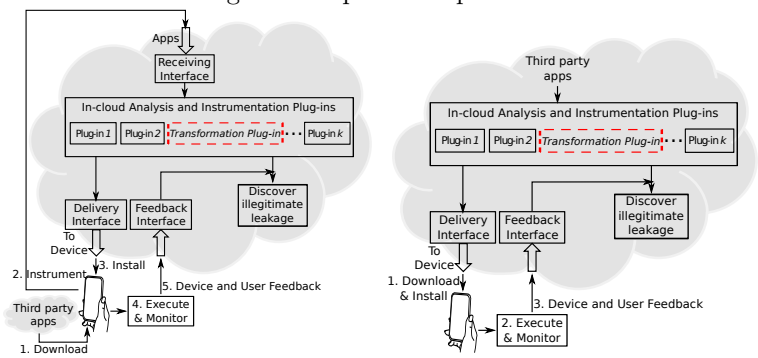
present some relevant discussion in Section 6 and related work in Section 7. We finally conclude in Section 8.

2 Background and Problem Statement

2.1 Android Background

Android is an operating system for mobile devices such as smartphones and tablets. It is based on the Linux kernel and implements middleware for telephony, application management, window management, etc. Applications are typically written in Java and compiled to Dalvik bytecode, which can run on Android. The bytecode and virtual machine mostly comply with the Java Virtual Machine Specification.

Unlike the JVM, The Dalvik Virtual Machine is a register-based VM. Each method has its own set of registers (not overlapping with other methods). Instructions address these registers to perform operations on them.



(a) Deployment by Vendor or Third-party Service (b) Deployment by Market

Fig. 1: Deployment by Vendor or Third-party Service

2.2 Problem Statement

Static analysis has its own advantages for information flow tracking, but a dynamic information flow tracking solution may also be desirable for the following reasons: (a) Static analysis may only tell what may happen but cannot tell what actually happens. Runtime conditions, including inputs from the user and the server may influence what actually happens, meaning that any privacy leaks may be classified as legitimate or illegitimate. Even if a static analysis can detect user interaction, what exactly a user confirms is very difficult for it to capture. (b) Private sources in Android, which are based on URIs, such as contacts, cannot be soundly tracked by static analysis (unless it marks all database queries as possible private sources). Databases such as contacts are accessed through corresponding URIs, which are simply wrapped strings and may be obfuscated or inaccessible statically. Lastly, (c) static analysis is often conservative due to scalability reasons, and thus may have false positives. In the light of all these points, we focus on dynamic information flow tracking in this paper.

Previous dynamic analysis approaches on Android for tracking information flow have modified the Dalvik VM or the library code to propagate taints [13, 4]. As this requires platform modification and thus limits the usability, it is rea-

sonable to ask whether dynamic information flow tracking is possible without platform modification by rewriting the apps alone. Uranine answers this question positively. It accepts stock apps from the user, and returns a ready-to-run instrumented app enabled with information flow tracking.

Deployment Models Figure 1 shows the two possible deployment models of our approach. The first model is suitable when there is no control on the source of apps. It is suitable for enterprise, third-party subscription services, individual users, and smartphone vendors and carriers. As the user downloads a third-party app, the downloaded app is passed to our system for instrumentation. Such a system would typically reside in the cloud as a service supported by the vendor or a third-party. It is also possible to place this service on the users' own personal computers or enterprise's servers. Once the app has been analyzed and instrumented by the system, the app is installed on the user's device. The app is then constantly monitored on-device as it runs. We note that the whole process may be completely automated with the use of an on-device app so the user needs to only confirm the removal of the original app and installation of the instrumented app. Furthermore, entry-level users may be provided with preset information flow tracking and enforcement policies. The second deployment model, which is more suitable for app markets and enterprise app stores, is slightly different in that the apps are instrumented before the user downloads and installs the app. We note that other existing real-time taint-tracking systems do not have similar deployment models.

Android apps are digitally signed by their developers, so instrumenting an app would require an application to be re-signed. The current app update system at Google Play (and possibly other Android markets) depends on apps' signatures. Deployment by third party services will therefore need to provide out-of-band mechanisms to notify users of available updates. This is however not much of a concern: mobile app management and app wrapping products such as Good [17] and MobileIron [22] already provide similar deployment models to enterprises in the context of API interposition similar to [35, 9].

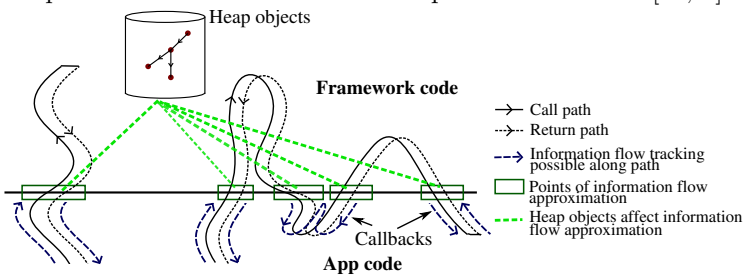


Fig. 2: A depiction of challenges C1 and C2 met in Uranine. There are paths between app code and framework code depicted as meandering function call paths and return paths, together with callbacks (the app code that is called by framework code). The left path results from ordinary calls while the right path includes callbacks. Information flow tracking can only be done for app code, requiring approximations for framework code. Callbacks must be handled soundly. Objects on the heap point to each other and their effect on information flow should be properly accounted for during approximations.

Challenges Following are the challenges that we solve in creating Uranine.

C1 Framework code should not be modified, i.e. we cannot instrument framework code. We summarize the effect of framework APIs according to a custom policy, combined with manual summarization for a few special cases. Previous works on static or dynamic binary instrumentation [24, 41, 36] have needed to summarize system calls or very simple functions in low-level libraries like `libc`, which are much simpler. Static analysis works also typically use summarization [21, 15] to achieve scalability. However, we show by example that in our context of dynamic analysis and complex framework with Java data structures in Android, summarization alone is not sufficient. Heap objects can be particularly challenging to handle, and we need additional techniques for effective taint propagation.

C2 The effect of callbacks should be accounted for. Callbacks are functions in app code that may be invoked by the framework code. Since framework code cannot be instrumented, we cannot do taint propagation when callbacks are invoked. We propose a technique which uses over-tainting to avoid false negatives.

C3 In the Java language model, objects follow reference semantics, so we must have a way to taint the locations referenced. Furthermore, objects are deallocated automatically by garbage collection, so our taint-tracking data structures should not interfere with garbage collection.

As noted above, there are trade-offs between system modification and detection accuracy. However, we note that even though we resort to over-tainting to solve some of the above challenges, our results demonstrate that a carefully conceived design may still have a low false positive rate in practice. We discuss our solutions in detail in the next section.

3 Uranine Design

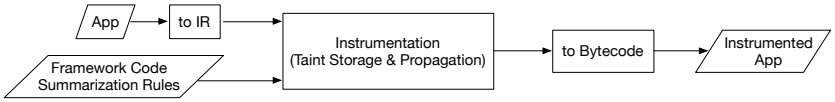


Fig. 3: Instrumentation flow in Uranine

Uranine offers a general framework for instrumenting applications statically and for providing information flow tracking, which may be used in a number of applications, including tracking privacy leaks and hardening applications against vulnerabilities. Figure 3 depicts the architecture of Uranine. When an app is given to Uranine, the app code is first converted to a custom intermediate representation (IR) that can be instrumented for taint propagation to happen at runtime. The instrumented IR is then converted back to bytecode and a new app is prepared. Since the framework code cannot be instrumented, we approximate the effects of framework code through a few general but customizable summarization rules. The rest of this section first describes our techniques for taint storage/propagation and the instrumentation details. The latter part of the section then describes our static analysis.

3.1 Taint Storage and Propagation

The techniques for taint storage and propagation influence the accuracy and runtime performance of privacy leakage detection. Our techniques focus on pro-

viding privacy leakage detection without false negatives under the constraint that the platform not be modified. Much of the design for taint tracking here is fairly routine and may be found in previous work [13, 30, 4]. We describe the routine or obvious aspects very briefly and then discuss in detail the specific challenges and corresponding solutions in our work.

Each entity that may be tainted is associated with a taint tag, which identifies what kind of private information may be carried by the entity. In the Uranine model, taints are stored and propagated for local variables (i.e. method registers), fields, method parameters/returns, and objects. Different bytecode instructions handle different storage types (i.e. local variables, fields, etc.) and accordingly have different taint propagation rules. Additionally, in a complete system, IPC (inter-process communication) taints and file taints may be handled at a coarser granularity. For IPC, the entire message carries the same taint. Similarly, an entire file is assigned a single taint tag. In our design, tracking IPC and file taints requires communication with an on-phone Uranine app, which keeps track of all file taints and IPC taints from instrumented applications. This paper focuses on taint tracking within Java code (more specifically, Dalvik bytecode) and further discussion on IPC and file taints is out of the scope of this paper.

We next describe the taint propagation rules for the different situations. We begin our discussion by assuming that we can instrument all the code (including the framework) and then introduce changes that would be required to leave the framework code intact.

Method-local registers. For each register that may possibly be tainted, we introduce a shadow register that stores the taint for this register. Any move operations simply also move the shadow registers. The same also happens for unary operations, while for binary operations, we combine the taints of the operands, assigning this to the shadow register of the result. Instructions assigning constants or new object instances cause the taint of the registers to be zeroed.

Heap objects. Heap objects include class objects containing fields, and arrays. For each field that may possibly be tainted, we insert an additional shadow taint field in the corresponding class. The load and store instructions for instance fields and static fields are instrumented to assign to or load from these taint fields to the local registers. We note that we may not insert additional fields into framework classes. In this case, we taint the entire object. How this is done and the effects of this will be discussed shortly.

In the case of arrays, each array is associated with only a single taint tag. If anything tainted is inserted into an array, the entire array becomes tainted. This policy is used for efficiency reasons, and has been also adopted by other works such as TaintDroid. We also support index-based tainting so that if there is an `array-get` (i.e. a load operation) with a tainted index, the retrieved value is tainted. We will discuss shortly how we associate taint with Array objects.

Method parameters and returns. Methods may be called with tainted parameters. In this case, we need to pass on the tainted information from the caller to the callee. We take a straightforward approach to achieve this—for each method parameter that may be tainted, we add an additional shadow parameter that carries the taint of the parameter. These shadow parameters may then convey the tainted information to the local registers. Method returns are trickier. Since we can return only one value, we instead introduce an additional

parameter to carry the taint of the return value. In Java, we have call-by-value semantics only, so that making assignments to the parameter inside the callee will not be visible to the caller. We therefore pass an object as the parameter, which is intended to wrap the return taint. The caller can then use this object to see the return taint set by the callee.

Our next part of discussion relates to specific challenges discussed in Section 2 and mostly relates to the requirement of not changing the framework code.

Calls into the framework (Challenge C1) Whereas the application code may be instrumented for taint propagation, we may only approximate the effects of calls into the framework code on taint propagation. We use a worst-case taint policy to propagate taints in this case:

- Static methods. For static methods with void return, we combine the taints of all the parameters and assign this to all the parameter taints. For static methods with non-void returns, the taints of all the parameters are combined and assigned to the taint of the register holding the return value.
- Non-static methods. Non-static methods often modify the receiver object (the object on which the method is invoked) in some way. Therefore, we combine the taints of all the non-receiver parameters; apart from its original taint, the receiver object is now additionally tainted with this combined taint. In case the method returns a value, the return taint is defined as the receiver taint.

Note that these rules are not enough to summarize the effects of framework code. Non-static methods often have arguments that are stored into some field of the receiver. Consider the following piece of code.

```
1  List list = new ArrayList();
2  StringBuffer sb = new StringBuffer();
3  list.add(sb);
4  sb.append(taintedString);
5  String ret = list.toString();
```

In this case, `sb` and `list` are untainted until line 4. Thereafter, `sb` is tainted and `ret` should be tainted because it will include the contents of `taintedString`. Our general solution is that when an object becomes tainted, any objects containing that should also become tainted. For every object o_1 that may be contained in another object o_2, we maintain a set of the containing objects. If the taint of o_1 ever changes, we propagate this taint to all the containing objects. The set of containing objects is updated whenever we have a framework method call $o_2.meth(.., o_1, ..)$, where $meth$ is a method on o_2 and possibly belongs to the framework code. This is a worst case solution; in certain cases, such a method would not lead o_1 to be contained in o_2. The update operation may be recursive, so that an update to taint of o_2 may lead to updating the taint of the objects containing o_2, and so on. Objects may point to (contain) each other and hence there may be cycles; the update operation will however achieve a fixed point eventually and then terminate.

Handling callbacks (Challenge C2) A callback is a piece of code that is passed onto another code to be executed later on. In Java, these are represented as methods of objects that are passed as arguments to some code, and the code may later invoke methods on that object. These objects typically implement an

interface (or extend a known class) so that code is aware which methods are available on the object.

Android makes an extensive use of callbacks, which often serve as entry points to the application. Examples of such callbacks are `Activity.onCreate()` and `View.onClick()` when overridden by subclasses. Apart from these, callbacks may be found at other places as well. For example, `toString()` and `equals()` methods on objects are callbacks. Identifying callback methods correctly may be done using class hierarchy analysis. A class hierarchy analysis analyzes the inheritance relationships between different classes and, based on these results, the overriding relationships between different methods. The class hierarchy analysis acts as a guide to the rest of the instrumentation by defining how different methods are dealt with during instrumentation.

Since callback methods override methods in the framework code, their method signatures may not be changed to accommodate shadow taint parameters and returns, lest the overriding relationships are disturbed. For example, consider the following class.

```
1  class DeviceIdKeeper {
2    private String id;
3    public DeviceIdKeeper(TelephonyManager m) {
4      id = m.getDeviceId();
5    }
6    public toString() { return id; }
7  }
```

The app code may call `toString()` on a `DeviceIdKeeper` instance. Since the return here may not be instrumented to propagate taint, we may lose the taint here. Furthermore, it is also possible that this method is called at some point by the framework code.

Our solution. In order to not lose taint in this case, our solution is to lift the return taints of all callback methods to the receiver objects. That is, in the instrumented callback method, the return taint is propagated to the receiver object taint. In case a possible callback method is called by app code with tainted parameters, we taint the receiver object with the taint of the parameters and then inside the method definition taint the parameter registers with the taint of the receiver. Since heap objects can carry taints in our model, such over-tainting needs to be done only in case of parameters of primitive types. With the parameter and return tainting in place, we may use the techniques described for calls into the framework (Section3.1) to summarize the effect of this call. The key to note here is that the receiver object of the callback serves as a convenient taint carrier and thus taint is not lost in both the cases: when the callback is called by an app method, and when it is called by the framework.

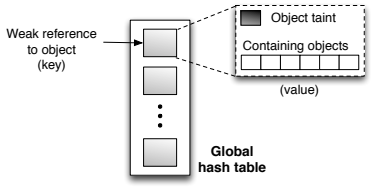


Fig. 4: Associating taint data-structures with objects

Taint data-structures (Challenge C3) From the above, it is quite clear that we need a way to taint objects. Java uses reference semantics to address

objects. That is, object variables are pointers to object values on the heap and assignment for objects is only a pointer copy. Thus, we may have two types of tainting, either tainting the pointer, or tainting the object. Storing pointer taints is simple and has been discussed as storing taints for method-locals and fields. In addition, we also need to associate a set of containing objects with each object (Section 3.1).

Our solution. In our solution, we use a global hashtable, in which the keys are objects and the values are records containing their taints and the set of containing objects. Any time the taints or containing objects needs to be accessed or updated, we access these records through the hashtable. Our hashtable uses weak references for keys to prevent interference with garbage collection. In Java, heap memory management is automatic; so we cannot know when an object gets garbage-collected. Weak references are references that do not prevent collection of objects and so are ideally suited for our applications. We further note that these data-structures should allow concurrent access as the instrumented app may have multiple threads running simultaneously. A schematic of our global hashtable is presented in Figure 4.

We considered but rejected an alternative method of keeping these data structures. With every object, we can possibly keep a shadow record, which is an object that stores the object taint and the set of containing objects in its fields. The instrumentation may then move this shadow record together with the main object through method-local moves, function calls and returns, and heap loads and stores. This technique however does not work well with the way we handle calls into the framework. Consider the following code fragment.

```
1  // list is a List
2  // obj is an object
3  list.add(obj);
4  obj2 = list.get(0);
```

In the above code, `obj` and `obj2` could be the same objects. However, since the loads/stores and moves inside the `List` methods are not visible to us, we cannot track the shadow record of `obj` there. The shadow record of `obj2` may at most depend on the record of `list`. Thus, there is no way to make the shadow records of `obj` and `obj2` the same, something that we achieve easily with our approach of weak hashtables.

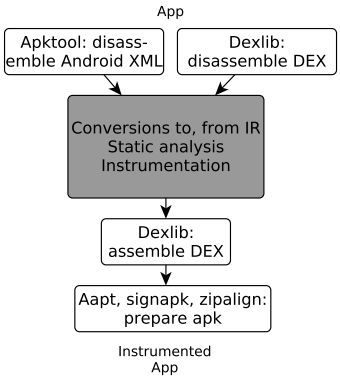


Fig. 5: Uranine implementation depicting the use of existing code (white boxes) and the features we implemented (gray, discussed in detail in Section 3).

4 Implementation

We have implemented a working prototype of Uranine. We use a library called `dexlib` [32] to disassemble and assemble Dalvik bytecode. The disassembled representation is converted to an intermediate representation (IR). In addition, we also use `apktool` [1] to disassemble the binary Android XML format (needed to discover entry points for static analysis) and other tools from the Android SDK and elsewhere to prepare an instrumented app. Figure 5 provides these details graphically.

We choose to work on an IR very close to the bytecode, and do not require decompilation to either Java bytecode or the source code as some previous works have required. Since decompilation is not always successful, this approach improves the robustness of our system. The IR enables us to simplify the bytecode instruction set to a smaller instruction set containing only the details relevant to the rest of the analysis and instrumentation. Disregarding details like register widths, the Dalvik bytecode instructions[4] generally have a direct correspondence with the instructions in the IR. Similar instructions (such as all binary operations or all kinds of field accesses) are represented as variants of the same IR instruction. Range instructions (`invoke-*/range` and `filled-new-array-*/range`) access a variable number of registers; these are converted to the simple representations of `invoke-*` and `filled-new-array-*` instructions with a variable number of register arguments in the IR. Even though we use this IR for instrumentation, it is also suitable for performing static control flow and data flow analysis. In fact, the same IR is used as input to our class hierarchy analysis, the results of which then guide the instrumentation. The instrumented IR is then finally assembled back to Dalvik bytecode.

Most of our instrumentation code is written in Scala, with about a hundred lines of Python code. The taint-tracking data structures and related code is written in Java. The instrumentation adds a compiled version of this code to every app for runtime execution. The total Uranine codebase sizes to over 6,000 lines of code. We note that Scala allows for writing terse code; the equivalent Java or C++ code is usually two to three times as long.

5 Evaluation

We evaluate Uranine on two aspects: accuracy and performance overhead. To perform accuracy evaluation, we configured Uranine to detect the leakage of location, phone identifiers (like IMEI and phone number), and contacts (address book). Our sinks include all APIs that send data to the network, write to the file system, or send SMS messages. We note that even though we restrict to a few relevant sources and sinks, we can easily extend the privacy leakage tracking by adding other private information sources and sinks as well.

Our app dataset consists of 1,490 apps randomly selected from Google Play. Apps are instrumented automatically and run with random inputs (fuzz testing) provided by the Android Monkey tool[5]. For understanding privacy leakage results, we also conducted manual tests for a smaller set of apps.

[4] http://s.android.com/devices/tech/dalvik/dalvik-bytecode.html
[5] http://developer.android.com/tools/help/monkey.html

Table 2: Accuracy evaluation of Uranine and comparison with TaintDroid

App	Uranine	TaintDroid	App	Uranine	TaintDroid
mobi.android-cloud.app.ptt.client	Contact	Contact	com.ama.lovetest.calculator	IMEI, Phone#	IMEI
com.enlightened.Androidskyjewelsfree	IMEI	None	com.flashlight.trefilm.coins	IMEI	IMEI
com.magmamobile.game.Slots	IMEI	None	com.silkenmermaid.gau.dldic	IMEI	IMEI
me.zed_0xff.android.alchemy	IMEI	None	com.gamevolution.MarbleMadnessPro	IMEI	IMEI
com.magmamobile.game.BubbleBlast2	IMEI	None	com.reverie.game.toilet-paper	IMEI	IMEI
com.rhs.wordhero	Loc	Loc	com.red.white.blue.free	IMEI	IMEI
com.rferl.almalence.staringcat	IMEI	IMEI	com.gameloft.android.ANMP.GloftGTFM	IMEI	IMEI
app.win.conforl1	ICCID, IMEI, Phone#	None	com.alloright.trib	IMEI, Loc, Phone#	IMEI, Loc
com.anbgames.openthedoor.hoola2	IMEI	IMEI	com.euro2012.geekbeach.acquariusoft	IMEI	IMEI
com.aceviral.toptruckfree	IMEI	IMEI	com.fjj24512014.korea	IMEI	None
com.flirtalike.android	IMEI, ICCID	IMEI, ICCID	net.aaronsoft.poker.eva	IMEI	IMEI
com.keithe.lwp.aquarium	IMEI	IMEI	com.mobizi.scratchers	IMEI	None
com.androiminigsm.fscifree	Contact, IMEI	Contact, IMEI	sg.vinay.FourpicsOnewordcheatsanswers	IMEI, Phone#	None
mobi.jackd.android	Loc	Loc	com.electricpocket.ringo	Contact	Contact
com.topface.topface	IMEI	IMEI	com.keek	IMEI, ICCID	IMEI
com.pilotfishmediainc.happyfish	IMEI, Loc, Phone#	Loc	com.phantomefx.reeldeal	IMEI	IMEI

5.1 Accuracy

In this section we evaluate how Uranine performs in detecting privacy leaks. We use our dataset real-world applications from Google Play for the evaluation. We use TaintDroid results to compare with our results. Our methodology involves running Uranine-instrumented applications on a TaintDroid build, allowing us to generate both TaintDroid's and Uranine's results together in one run, and thus eliminating any differences that may arise because of random inputs or non-determinism in multiple runs.

Manual tests. We conducted manual tests on a physical device (Samsung Nexus S) over a small random subset of apps. These results enable us to carefully study the differences between TaintDroid and Uranine. The results are depicted in Table 2. The results, where neither TaintDroid nor Uranine detected any leakage, are not shown in the table.

Our results show some disagreement with TaintDroid. We see that TaintDroid does not detect any phone number leaks that we detect; a look into TaintDroid code then revealed to us that TaintDroid has disabled tracking of phone numbers with the comment "causes overflow in logcat, disable for now" in source code. In all other cases of disagreement between Uranine and TaintDroid, we manually confirmed the correctness of Uranine. It turns out that in the cases where Uranine does detects an IMEI (or ICCID) leak while TaintDroid does not, there is some kind of hashing of the identifier involved, such as the calculation of MD5 or SHA1 digests. It appears that TaintDroid does not propagate taint across the functions that calculate these digests. This is also confirmed in AppsPlayground [26]. In conclusion, our results are generally consistent with TaintDroid. Any apparent inconsistencies result from implementation artifacts of taint tracking. It is worth

emphasizing here that our contribution is not to show an improvement over other systems in terms of detecting more privacy leaks, but to do the detection without system modification.

Automatic tests. We further conducted automatic, random testing on a bigger dataset of 1,490 apps. The tests were conducted on the Android emulator (provided with the Android SDK) running a TaintDroid image. Since the emulator does not provide most of the device identifiers (such as IMEI and phone number), we further added some code to our emulator image to provide real-looking identifiers on the respective APIs for accessing these identifiers. Because of these modifications, our emulator's TaintDroid can also detect phone number and IMSI leaks.

Our runs detected privacy leaks in a total of 360 apps; in the rest of the apps, no leak was detected either by TaintDroid or Uranine. The results for TaintDroid and Uranine differed for 177 apps. We have manually analyzed each of these cases, and have found that Uranine was accurate in most cases. Below, we detail our findings and bring out relevant insights.

For 92 apps where Uranine detected privacy leaks but TaintDroid did not—we confirmed that these were TaintDroid's false negatives. In all these cases, the apps leak the device identifiers after hashing (with, for example, MD5). In most cases, we were able to see the MD5 checksum of the device identifier being leaked (IMEI leaks were most frequent) in plaintext. Further, in other cases, these leaks were in ad libraries that are known to have the leaks flagged by Uranine. For example, our analysis of an older version of the Admob library shows that it leaks the MD5 of a string derived from the phone's IMEI number.

Uranine's detection of leakage in 4 apps is likely to be a false positive. In two apps, our logs reveal Uranine flags leakage when an empty string is being written to a file. In the other two cases, Uranine detects IMEI leakages on writing strings that look like base64 codes. Decoding those codes however does not reveal the IMEI number nor anything that looks like a hash of that. False positives are actually expected in Uranine, due to overtainting as part of our design. Considering this, 0.2% false positives are insignificant.

Table 3: Leaks detected in automatic tests

Leak type	Apps leaking	Leak type	Apps leaking
IMEI	310	IMSI	18
ICCID	16	Phone #	79
Location	107	Contacts	5

There was another set of 13 apps where Uranine flagged leakage but TaintDroid did not. In all these cases, we can see strings looking like MD5 or SHA hashes being leaked, but were unable to derive them from known identifiers (perhaps they were mixed with some salt before hashing). Though we could not classify these cases, we believe them to be TaintDroid false negatives. Finally, we detected 14 cases that were false negatives for Uranine—we could however correct them by adding additional sinks that we missed earlier.

In summary, we found Uranine to be fairly accurate in detecting privacy leaks with few errors. Table 3 shows the privacy leaks detected by Uranine.

5.2 Performance

Measuring the runtime overhead of applications instrumented by Uranine is not trivial. First, there are no popular macrobenchmarks for Android. The DaCapo benchmarks [6], which are popular Java benchmarks, are not easily ported to Android (due to their use of Java-specific libraries and GUI) and moreover, may also differ from real-world application workloads on Android. Second, conventional microbenchmark suites for evaluating virtual machine performance may also give skewed results as we are instrumenting applications here rather than the virtual machine. A lot of the code for real applications runs in the framework, is not instrumented, and runs without overhead—a microbenchmark will thus misrepresent this situation.

We measure performance overhead using real-world Android applications. However, most applications are GUI-intensive and interactive in nature. Thus, one cannot simply run the benchmark application and obtain the results. We devise our own methodology of evaluating performance of Android applications in response to certain events. For our benchmarks, we select a total of six events from three very popular applications: BBC News, Last.fm (a music application with social networking features), and the stock Android application for managing contacts. For each application, we evaluate the time to launch the main activity of the application and the time to complete a click of a pre-selected feature on the application. The time to launch the main activity is as reported by the ActivityManager (part of the Android middleware). The time to complete a click is measured by instrumenting the click handler function to report the interval from its beginning to the point it returns.

Table 4: Macrobenchmark performance. The reported times (Original/Instrumented columns) are medians over five independent runs.

Benchmark	Event	Original (ms)	Instrumented (ms)	Overhead
BBC News (version 2.5.2 WW)	Launch	953	1418	49%
BBC News (version 2.5.2 WW)	Click ("Live BBC World Service")	450	434	-
Last.fm (version 1.9.9.2)	Launch	523	567	10%
Last.fm (version 1.9.9.2)	Click ("Sign up")	132	140	6%
Contacts (from AOSP 4.0.4)	Launch	580	645	11%
Contacts (from AOSP 4.0.4)	Click ("Done" after contact creation)	23	59	156%

Table 4 presents the comparison of the original applications and those instrumented for information flow-tracking. As can be seen from the table performance overhead is usually low, almost always within 50% and often around 10%. We attribute this to the fact that the Android framework does most of the heavy-lifting during runtime, from creating the UI to managing the data structures and data stores. Thus, even though we may expect a huge performance overhead because each instruction is instrumented, real-world application overhead appears quite low in comparison. Anecdotally, in our runs, we have seen noticeable performance overheads, but the overheads have never been intolerable. Furthermore, the performance of Uranine compares favorably with the reported performance of TaintDroid (15-30% overhead) and Phosphor (50% overhead).

Finally, we would like to reiterate that our approach is highly amenable to static analysis. We expect that in production, a tool such as Uranine will be guided by a static analysis, which will be able to identify that most paths cannot propagate the relevant information and thus do not need instrumentation.

6 Discussion

6.1 Static Analysis and Optimizations

We believe that Uranine has great potential for optimizations so that runtime overhead can be minimized. First, it is possible to tune the instrumentation, and perform constant propagation passes to reduce the instrumentation overhead. Second and more importantly, it is possible to perform a static information flow analysis that identifies the paths along which the relevant information flow could take place. Such paths are usually small in number, and if Uranine instruments those paths only, applications may run with negligible overhead. In fact, the implementation of Uranine already includes hooks to attach a static analysis, which can then guide the instrumentation. We have performed preliminary studies testing the use of static analysis to guide the instrumentation and a complete study is part of future work. Note that the use of static analysis does not obviate the need for a dynamic analysis system (Section 2.2).

We note that the opportunity for static analysis is present in our approach only, involving no platform modification. Previous works such as Phosphor [4] modify the platform libraries to track information flow and will therefore not benefit much from optimizing instrumentation by static analysis.

6.2 Limitations

We discuss here our limitations and avenues of future work. While Uranine is good for detecting privacy leaks in legitimate applications, a truly malicious app may be able to evade the system through some of these limitations.

Implicit flows. A fundamental limitation of dynamic taint tracking is the inability to track implicit information flows via control flow [30]. Our work shares this limitation. Static analysis may be used to track control flow. However, this leads to the risk of severe over-tainting. Research is underway to make implicit flow tracking practical [19].

Native code. We currently do not support taint tracking through native code, which some Android applications include in addition to bytecode. Previous works such as Phosphor and TaintDroid, as well as static analysis works on Android which only analyze bytecodes, all have this limitation.

Dynamic aspects of Java. As a limitation of static instrumentation, the dynamic aspects of JVM, such as reflection and dynamic class loading (using `DexClassLoader` or similar features in Android) do not cleanly fit in. These may however be supported by our approach in the future. We may apply worst-case tainting for all method calls made by reflection as we do for other methods. Furthermore, we can instrument calls by reflection and alert the user if they do not pass certain security policies (such as restricting reflective calls to only certain APIs in the Android platform). Code loaded by dynamic class loading may also not be available during static instrumentation. In a deployment, it may be possible to prompt the user to allow re-analysis whenever dynamic code loading is detected, so that an instrumented version of the code being loaded can be created.

Incorrect summarization. Policy-based summarization of framework code, as used in our work, not only has the problem of over-tainting, but could also result in under-tainting of data passing through APIs that do not fit within

those policies. For example, some classes may update a global state when their methods are called. We are not aware of such a situation but such cases could be used to bypass the system. Manual summarization of known cases is obviously one solution. Automatic method summarization is an open research problem in static analysis, and any progress there will benefit our cause as well.

7 Related Work

Information flow tracking. The closest to our work are TaintDroid [13] and Phosphor [4]. The key advantage of our technique is that we do not require modification of the Android platform as these do.

Dynamic taint analysis has been employed in a variety of applications from vulnerability detection and preventing software attacks [24, 25, 33] and malware analysis [31, 37] to preventing privacy exposures [41, 10]. We present a general technique for taint tracking in this paper without modifying the Android platform. Our technique may be used for the above applications, especially when there is a constraint to run applications on an unmodified platform. There are also works doing taint tracking by bytecode instrumentation. Haldar et al. [18] implement taint tracking by instrumenting the Java String class. Chandra and Franz [8] instrument the Java bytecode for taint tracking. These works share the same limitation of Phosphor discussed earlier.

There are also a number of related works using static analysis. PiOS [11] uses it to detect privacy leaks on iOS apps. Enck et al. [14] and Gibler et al. [16] decompile Dalvik to Java bytecode and perform static analysis on that using existing tools for Java. FlowDroid [2] also converts Dalvik back to Java bytecode and builds on top of Soot[6] while adding in Android-specific requirements to the analysis. Chex converts Dalvik bytecode to the WALA[7] IR and then employs WALA for static analysis [21]. Cao et al. [7] automatically collected implicit control flow transitions through the Android framework code to assist static analysis tools. As discussed earlier, there are limitations of static analysis over real-time dynamic analysis. Xia et al. [34] eliminate some limitations by performing offline partial executions of apps after static analysis. However, they are still unable to handle situations with external input from users or servers, which is quite common.

Static instrumentation. Static instrumentation has been used earlier for Android applications [35, 9]. These works have focused on API interposition rather than tracking information flow; the latter is more challenging because of the need to instrument many instructions and to encode the semantics of information-flow tracking. AppSealer [38] statically instruments Android applications to repair component hijacking vulnerabilities. Capper [39] is a follow-up work that detects privacy leakages without platform modification. Both these works are similar to Uranine; however, their taint tracking will have false negatives: they try to address C1 but do not solve it adequately and do not even discuss C2 and C3. Instrumentation has been used in other applications as well, some of which even use static analysis to optimize it. Saxena et al. use static analysis to make

[6] http://www.sable.mcgill.ca/soot/
[7] http://wala.sourceforge.net

their binary instrumentation efficient [28]. Xu et al. [36] instrument C sources for taint tracking and further optimize it using static analysis.

Other related work in mobile device security. Kirin [12] defines security policies based on Android permissions. A number of works additionally prevent access of private information or supply fake data to apps [5, 40, 23]. The above works enable access control while we provide information flow control. Another line of works [20, 3] investigates user perceptions as related to mobile privacy. They conclude that users are often not aware of privacy leakages, and that proper awareness and usable controls can mitigate users' concerns about privacy. Rosen et al. [27] perform static analysis of Android applications and provide end-users with information about privacy-related behaviors of these applications. Our tool could easily supplement such works by providing real-time insights about the behaviors of these applications to the users. Finally, researchers have developed proof-of-concept malware utilizing side channels that cannot be detected by a traditional information-flow analysis such as ours [29].

8 Conclusion

This paper describes Uranine, a framework for dynamic privacy-leakage detection in Android applications without modifying the Android platform. To achieve this, Uranine statically instruments Android apps only, and does not need support for information flow tracking from the platform. We present a design and implementation of Uranine and evaluate its performance and accuracy. Our results show that Uranine provides good accuracy and incurs acceptable performance overhead compared to other approaches.

9 Acknowledgments

We thank our reviewers for their valuable comments. We also thank Peng Xu and Weiwu Zhu, who helped in the system implementation and evaluation. This paper was made possible by NPRP grant 6-1014-2-414 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] Apktool. Android-apktool: A tool for reengineering Android apk files. `http://code.google.com/p/android-apktool/`.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM PLDI*, 2014.

[3] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. Little brothers watching you: Raising awareness of data leaks on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 12. ACM, 2013.

[4] J. Bell and G. E. Kaiser. Phosphor: Illuminating dynamic data flow in the jvm. In *OOPSLA*, 2014.

[5] A. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[7] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[8] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC*, 2007.

[9] B. Davis and H. Chen. "retroskeleton: Retrofitting android apps". In *Mobisys*, 2013.

[10] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix ATC*, 2007.

[11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[12] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *ACSAC*, 2009.

[13] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.

[15] A. Fuchs, A. Chaudhuri, and J. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland*, `http://www.cs.umd.edu/~avik/projects/scandroidascaa`, 2009.

[16] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing*, 2012.

[17] Good. Mobile app containerization. `http://www1.good.com/secure-mobility-solution/mobile-application-containerization`.

[18] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2005.

[19] M. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. *Proc. of NDSS*, 2011.

[20] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *ACM CCS*, 2012.

[22] MobileIron. Appconnect. `http://www.mobileiron.com/en/products/appconnect`.

[23] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, 2010.

[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[25] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *IEEE MICRO*, 2006.

[26] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of ACM CODASPY*, 2013.

[27] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[28] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *IEEE/ACM CGO*, 2008.

[29] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[30] E. 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 SP*, 2010.

[31] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, 2009.

[32] smali. Smali: An assembler/disassembler for Android's dex format . `http://code.google.com/p/smali/`.

[33] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGPLAN Notices*, volume 39, pages 85–96, 2004.

[34] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *IEEE S&P*, 2015.

[35] R. Xu, H. Saıdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security*, 2012.

[36] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, 2007.

[38] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[39] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *ASIACCS*, 2014.

[40] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.

[41] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1): 142–154, 2011.