POSTER: Towards Compiler-Assisted Taint Tracking on the Android Runtime (ART)

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

Oliver Schranz
CISPA, Saarland University
schranz@cs.uni-saarland.de

Philipp von Styp-Rekowsky
CISPA, Saarland University
styp-rekowsky@cs.uni-saarland.de

ABSTRACT

Dynamic analysis and taint tracking on Android was typically implemented by instrumenting the Dalvik Virtual Machine. However, the new Android Runtime (ART) introduced in Android 5 replaces the interpreter with an on-device compiler suite. Therefore as of Android 5, the applicability of interpreter instrumentation-based approaches like TaintDroid [1] is limited to Android versions up to 4.4 Kitkat. In this poster, we present ongoing work on re-enabling taint tracking for apps by instrumenting the *Optimizing* backend, used by the new ART compiler suite for code generation. As Android now compiles apps ahead-of-time from dex bytecode to platform specific native code on the device itself, an instrumented compiler provides the opportunity to emit additional instructions that enable the actual taint tracking. The result is a custom compiler that takes arbitrary app APKs and transforms them into self-taint tracking native code, executable by the Android Runtime.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection–Information flow controls

Keywords

Information Flow Control, Taint Tracking, Android, ART, Compiler

1. MOTIVATION

In modern mobile operating systems, potentially privacy-critical user data and hardware capabilities are often exposed to apps through permission-guarded APIs. For Android versions up to 5.1, those permissions are requested and granted only once at installation time, while the approaching Android M release will feature dynamic permission revocation for a selected subset of permissions. However, the actual usage of those privacy-sensitive resources remains completely opaque to the user. In order to take a closer look at the data usage of apps, taint tracking has proven valuable in the Android setting.

Taint Tracking on Android. In order to track the flow of information through an application, the app's code needs to be instrumented. Each piece of data returned from a privacy-critical *source*, like a permission protected API, is tainted immediately in the app process itself. In case such data is used in computations, the taint carries over to the results. As soon as a data *sink* (e.g. network transaction, I/O call) is reached, the involved variables' taint values are checked and a predefined set of actions is triggered. For example, a dynamic system could interrupt an app in case it tries to send data derived from a user's contact list to a remote server. In the general case, the definition of *sinks*, *sources* and the actions to take upon leakage are specified by policies.

State of the Art. We distinguish static and dynamic approaches for taint tracking. Static techniques analyze apps offline [2][3] to provide risk assessment before the app is actually run or installed. The downside is their lack of runtime information, raising coverage problems if code is dynamically loaded at runtime. Dynamic systems [1] monitor the app at runtime and can actively prevent data leakage but suffer from missing structural information to detect implicit information flow. Even though hybrid approaches exist to provide better results, soundness and completeness are still open challenges in many contexts. Despite those downsides, taint tracking has proven to be a powerful tool to detect data leakage in practice.

While offline analysis remains unaffected by the new Android Runtime (ART), dynamic approaches which instrument the Dalvik Virtual Machine suffer from the fact that the interpreter is replaced by an ahead-of-time compiler suite. Therefore, systems like TaintDroid [1] lack their instrumentation target and thus are not applicable anymore as of Android 5.

Our Ongoing Work. The new runtime provides a novel opportunity for Android solutions to place the instrumentation point in the compiler itself. At this point, we have full control over how the app is compiled and what is produced as its native representation, without the need to rewrite the original app or to modify the concrete runtime environment. Using this approach, additional code to handle the taint tracking is inserted as a part of the compilation process. From the system's point of view, a regular app is executed since the code is not injected dynamically but already present in the compiled app.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
DOI: http://dx.doi.org/10.1145/2810103.2810129.

2. DESIGN DECISIONS

2.1 Target Compiler

The introduction of a dedicated compiler suite changed the way apps are installed on Android. In the past, the bytecode was optimized before execution and just-in-time (JIT) compiled at runtime by the Dalvik Virtual Machine. Even though the new compiler suite operates on dex files for compatibility reasons, it comprises a flexible toolset instead of a simple replacement for dex optimization. The *dex2oat* executable serves as an entry point to the compiler suite and takes advantage of several possible compiler backends for the actual code generation. *Quick*, the current standard, is derived from the JIT compiler and thus not suited for more complex ahead-of-time optimizations such as Linear Scan Register Allocation. In order to allow for state of the art optimization techniques during the compilation phase, Google designed a new compiler backend from scratch, namely the *Optimizing* backend. Figure 1 gives an overview on how the compiler suite compiles an application using the *Optimizing* backend. The intermediate representation (IR) is a control flow graph enriched with meta information like post-domination and def-use-pairs, created to ease complex optimizations. Given its clean and powerful IR and the indication that Google is planning to make it default in the long term, it yields a promising instrumentation target.

2.2 Instrumentation Points

For the *Optimizing* backend, there are three canonical locations to place the taint tracking code generation at.

1. **Dex to IR Transformation.** The logic to insert new nodes into the IR and interlink them with the current graph is already present at this stage and can be reused to generate additional tracking code. Inserting new code is possible here in a non-invasive way that leaves the IR intact and therefore does not interfere with the subsequent optimization passes.
2. **IR Graph.** Adding nodes here is not as straightforward, but gives the opportunity to decide which optimization passes may ease the process of code injection and may run beforehand, and which ones have to follow afterwards. This allows for flexible placement of the tracking code generation phase. Additionally, the built and interlinked IR graph provides the necessary infrastructure to create own passes.
3. **IR to Native Code Transformation.** Directly adding code here is comparable to instrumenting the resulting binary, even though helper functions are present to ease the process of generating native code. However, the compiler's abstraction from registers and stack slots is not applicable to code not present in the IR, so this logic has to be replicated. Furthermore, optimization passes over the taint tracking code are not easily possible at this late phase because they rely on a proper IR graph that lacks information about our additional code in this case.

We reuse the IR generation routines of the dex to IR transformation phase to add taint tracking code, but for the future we plan to combine it with optimization passes that operate on the created IR graph to improve the runtime performance of the resulting code.

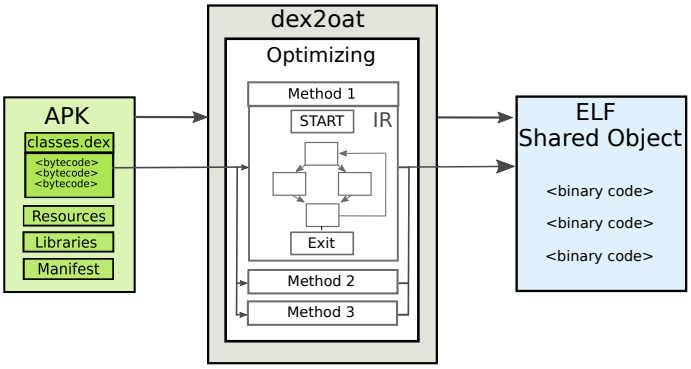

Figure 1: Sequence diagram of the *dex2oat* suite compiling an application. The *Optimizing* backend is used to first transform each method to an IR graph and finally compile them to native code.

2.3 Performance

Typically, dynamic taint tracking requires additional code to be added to take care of the taint generation, propagation, and checking. This usually imposes significant performance overhead. The injected code requires additional computation time and the memory footprint increases due to the storage of taints. There is always a trade-off between performance overhead and accuracy, raising the challenge on how to get reasonable results without sacrificing app performance to an extent that affects user-experience. One of the reasons to instrument the compiler instead of modifying the dex file directly is the advantage of benefiting from the compiler's optimization infrastructure. The IR graph is tailored in a way that greatly facilitates adding new optimizations, like passes that utilize static information to optimize the taint tracking code. For instance, the tainting of local variables that again have only local variables in their forward slice can be omitted.

2.4 Accuracy

In most cases, taint tracking is neither complete nor sound, still recent results [1][2] prove its value in finding critical data leakage. In our setting, tainting is done for intra-process computation as only the app and not the whole system is seen by the compiler, so if tainted data leaves our controlled environment, the taint may get lost. In general, there are several points in Android's application framework that introduce certain kinds of haziness concerning taint tracking. For some of those challenges, the amount of accuracy loss can be decreased by extending the scope of the taint tracking or by defining proper policies.

Framework and Boot Image. Android provides a rich set of framework libraries, like *framework.jar*, that applications can use for recurring tasks. ART precompiles such libraries into a boot image consisting of *boot.oat* and *boot.art*. It is loaded into each application's process for optimization purposes as it constitutes the classes that most apps make use of. Creation of the boot image happens once during startup after an Android upgrade and is performed by the *dex2oat* compiler suite. This gives us the opportunity to also include taint tracking code in the framework classes to increase the accuracy by using our instrumentation for the image compilation, too.

I/O and Serialization. When tainted data is written to file storage, it is not straightforward to decide how to handle its taint. First problem is to keep the tainting information intact. Simply dropping the taint results in a loss of accuracy, but keeping the taint may raise compatibility issues with other non-taint tracking apps. The sweet spot is to actually store the taint along the actual data in a backwards-compatible way. A naïve solution would store files with additional meta files carrying the taint information. If the file is read by code that supports taint tracking, then the taint is also read, otherwise it is simply ignored. TaintDroid [1] utilizes the file system's extended attributes to store the taint. However, this requires system modifications. Second problem is to define a proper granularity. If tainted data is appended to a file, either the whole file or only the specific subtext is tainted, depending on how fine-grained the tainting is defined.

Native Libraries. Native libraries are already shared objects and therefore there is no need for the backend to compile them again. The downside is that the compiler has no information about the actual native code that is called from within the Java world. Simply ignoring taint information for native calls may miss leakage, but tainting every variable returned from the native world can result in massive over-tainting which leads to false-positives. While *TaintDroid* completely refuses to load native libraries that are not part of the firmware [1], we plan to support them. Tolerating native calls may decrease the accuracy but also greatly enhance compatibility and acceptance since games for instance often utilize native libraries for performance reasons.

Inter Process Communication. Android makes heavy use of the *Binder* module for inter-process communication (IPC), which poses challenges similar to those for *I/O and Serialization*. For the app-to-app communication, the policy depends on whether both apps support taint tracking. This topic was recently addressed by a system called IccTA [4]. Beside inter-app communication, *Binder* IPC is frequently used for communication with the system as many manager classes are *Binder* proxies connected to the middleware. For those cases, taint information is lost unless the system is modified to also include taint-aware code.

2.5 Generalization

The approach described in this work is suited towards injecting taint tracking code into applications in a non-invasive and stable way. The idea however can be generalized to support arbitrary instrumentation at the compiler level. We plan to extend our compiler additions to an instrumentation framework, comparable to dexlib [5]. Approaches like inline reference monitoring [6] that often rely on rewriting or hooking may for instance benefit from the flexibility and robustness of a compiler-based code injection framework that can potentially also expose parts of the backend's optimization infrastructure.

2.6 Deployment

Currently, our implementation solely instruments the compiler suite residing in *libart-compiler.so* while the runtime environment *libart.so* remains unmodified. This allows us to ship our instrumented *dex2oat* binary and its dependencies to the device using a regular application. Given an arbitrary third-party application that we want to make self-taint tracking, we first install it, which will result in Android compiling it using its unmodified on-device compiler. Second, we run our own custom compiler on the app's APK file to create the native version, which now includes the taint tracking code. Third, we replace the native version generated by Android with our recompiled one. Fourth, we start the app using an Intent. The (unmodified) Android runtime will now load our version of the compiled app. All we need is the app APK, which for installed apps is stored on the device or otherwise can be obtained from an app market, and root rights to successfully execute steps one and three. Our prototype runs on a rooted Android compiled from AOSP master branch (checked out June 2015) and successfully creates and installs self-taint tracking apps without involving the developer and without changing the system. In addition, a virtualization technique like Boxify [7] could allow for achieving app instrumentation and execution without root privileges.

3. CONCLUSIONS

We presented our current work-in-progress for app taint tracking on Android, moving towards overcoming the caveats ART introduced in this field. Our system instruments the *Optimizing* backend, which is designed for supporting state-of-the-art compiler optimizations and will be default in future versions of Android. Our prototype comes as a regular Android application and can recompile arbitrary third-party apps to be self-taint tracking on rooted devices. Apps compiled with our approach are executed as usual with Android being completely agnostic about their taint tracking abilities. The approach can be generalized to an instrumentation framework that may ease further research which depends on analysis, rewriting or hooking techniques.

4. REFERENCES

[1] W. Enck, P. Gilbert, B. 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 *USENIX OSDI 2010*.

[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 SIGPLAN PLDI 2014*.

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

[4] L. Li, A. Bartel, T.F.D.A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: detecting inter-component privacy leaks in android apps. In *IEEE/ACM ICSE 2015*.

[5] Gruver B. Smali: A assembler/disassembler for android's dex format. `http://code.google.com/p/smali`. [Online; accessed 8-July-2015].

[6] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, 2004.

[7] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security 2015*.