# Poster: ARTist: The Android Runtime Instrumentation and Security Toolkit

Michael Backes*, Sven Bugiel†, Oliver Schranz†, Philipp von Styp-Rekowsky† and Sebastian Weisgerber†
*CISPA, Saarland University, MPI-SWS
backes@cs.uni-saarland.de
†CISPA, Saarland University
{bugiel,schranz,styp-rekowsky,weisgerber}@cs.uni-saarland.de

***Abstract*—We present *ARTist*, a compiler-based application instrumentation solution for Android. *ARTist* is based on the new ART runtime and the on-device *dex2oat* compiler of Android, which replaced the interpreter-based managed runtime (DVM) from Android version 5 onwards. Since *dex2oat* is yet uncharted, our approach required first and foremost a thorough study of the compiler suite's internals and in particular of the new default compiler backend *Optimizing*. Moreover, we exemplify the viability of *ARTist* by re-instantiating intra-application taint tracking solutions, which hitherto depend on the now abandoned DVM, on Android 6. Our results in particular provide compelling arguments for prefering compiler-based instrumentation over alternative bytecode or binary rewriting approaches.**

## I. Introduction

Google's Android OS has become a popular subject of the security research community over the last few years. Among the different directions of research on improving Android's security, a dedicated line of work has successfully investigated how instrumentation of the interpreter (i.e., Dalvik virtual machine) can be leveraged for security purposes. This line of work comprises seminal works such as TaintDroid [1] for analyzing privacy relevant data flows within applications, AppFence [2] for protecting the end-users' privacy, or Moses [3] for domain isolation, just to name a few.

However, with the release of Android 5 Lollipop, Google made a large technological leap by replacing the interpreter-based runtime with an on-device, ahead-of-time compilation of apps to platform specific bytecode that is executed in the new Android runtime (short ART). While this leap did not affect the app developers, it broke legacy compliance of all of the previously mentioned security solutions that rely on instrumentation of the DVM and it restricts them to Android versions prior to Lollipop. In fact, it has left the security research community with two choices for carrying on work that relies on instrumented runtimes: resorting to binary or bytecode rewriting techniques [4][5] or adapting to the novel but uncharted on-device compiler infrastructure.

***Our contributions.*** In this work, we present a compiler-based solution that is not only able to re-instantiate previous solutions such as dynamic, intra-application taint tracking [1], but, moreover, to provide a more robust, reliable, and integrated application-layer instrumentation approach than previously possible. Concretely, we make the following contributions.

*Study of the ART compiler suite.* Since the novel ART compiler suite, *dex2oat*, is still uncharted, our solution required first and foremost a thorough study of the newly introduced *dex2oat* compiler. We provide, to the best of our knowledge, the first in-depth, comprehensive study of the internals of ART's compiler suite in the form of a companioning technical report [**?**]. In particular, we deep-dive into its most recent backend called *Optimizing* that became the default with Android 6 Marshmallow. Those new insights not only allow us to implement a compiler-based solution, but also form expert knowledge that facilitates independent research on the topic.

*Compiler-based app instrumentation.* We design and implement a novel approach, called *ARTist* (ART Instrumentation and Security Toolbox), for application instrumentation based on an extended version of ART's compiler frontend *dex2oat*. Our system leverages the compiler's rich optimization framework to safely optimize the newly instrumented application code. The instrumentation process is guided by static analysis that utilizes the compiler's intermediate representation of the app's code as well as its static program information in order to efficiently determine instrumentation targets. A particular benefit of our solution, in contrast to alternative application layer solutions (i.e., bytecode or binary rewriting), is that the application signature is unchanged and therefore Android's signature-based same origin model and its central update utility remain intact. To demonstrate the benefits of a solution such as our *ARTist*, we conduct a case study by instantiating a TaintDroid-inspired [1] dynamic intra-application taint tracking solution using *ARTist*. Our results provide compelling arguments for preferring compiler-based instrumentation over alternative bytecode or binary rewriting approaches.

## II. The Android Runtime Instrumentation and Security Toolkit

The architecture of *ARTist* consists of two major components: a security-instrumented compiler (*sec-compiler*) and an app to deploy the compiler (*deployment app*). The *sec-compiler* is our implementation of a compile-time instrumentation tool that is based on the *dex2oat* compiler. The latter is a regular Android application that ships, deploys, and manages the *sec-compiler*.

### A. Security-Instrumented Compiler

The general concept of security-instrumented compilers is not restricted in its modifications of the compiler. Given *dex2oat*'s modular design, there are immediately multiple possibilities apparent where app modifying code could be placed. For instance, *dex2oat*'s design would easily allow porting bytecode and binary rewriting approaches ($\mathsf{Instr}_{DEX}$ & $\mathsf{Instr}_{BIN}$) into the compiler infrastructure (cf. Figure 1). Of the different choices, *ARTist*'s *sec-compiler* is concretely designed to operate on the intermediate representation of

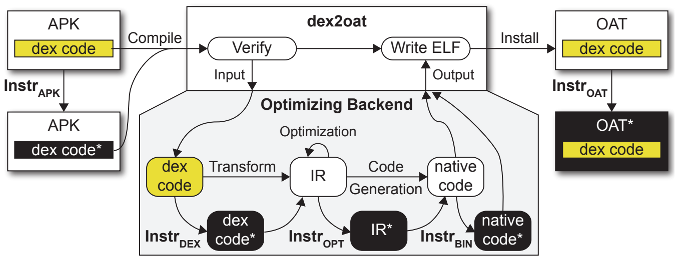


Fig. 1. The code instrumentation points before, during, and after the compilation for different representations of the app code. Instrumented code is depicted in black boxes.

*dex2oat*'s *Optimizing* backend ($\mathsf{Instr}_{OPT}$), where the existing optimization infrastructure and static code information in the *Optimizing* IR benefit an efficient and precise code modification. More precisely, our app instrumentation code is realized as an *Optimization* and therefore modularly integrated into the optimization workflow. Consequently, our security instrumentation logic has full control over the ordering and execution of optimizations, which opens up the opportunity to optimize the already instrumented code by creating or applying compatible optimizations that improve the performance of the security code. Generally, using the *Optimization* interface one can extend the compiler with custom functionality (*Modules*) that is decoupled from *dex2oat*'s code base. In addition, our integrated solution implicitly takes advantage of the robustness of *Optimizing*'s code generators, which are well-tested, constantly improved, and in productive use on every stock Android phone running version 6+.

## *B. Compiler Deployment App*

Responsibility of the *deployment app* is to deploy the *sec-compiler* at application layer in addition to the system's *dex2oat* binary. Using *deployment app*, one can create security-instrumented versions of installed applications by re-compiling the apps' bytecode with *sec-compiler* and replacing the oat files stored on filesystem. To make the Android runtime agnostic to this instrumentation, two particular challenges had to be overcome. First, Android has mechanisms in place to verify that oat files correspond to their respective apps and that the paths of the oat files are correct. Our implementation solves this challenge by rewriting paths and checksums to match those that the system *dex2oat* would have generated. Second, the oat files are by default stored at and loaded from a protected location to which $3^{rd}$ party apps have no access. A naïve solution to this problem would be to require extended privileges for our *deployment app* (e.g., a dedicated SELinux type or root on security-relaxed after-market ROMs). Alternatively, app virtualization solutions such as Boxify[6] and NJAS[7] can be applied to solve this problem without requiring extended privileges. In either case, the Android default runtime will load the instrumented oat file while remaining agnostic to the fact that it was replaced by our customized version.

While the instrumentation with *ARTist* already provides powerful tools to modify the application, most security solutions require additional custom code within the app (e.g., a policy decision point). To facilitate adding custom code to an instrumented app, *deployment app* has a preprocessing step that combines the app's original bytecode with the additional code before the compilation. During compilation, connections between original and new code are built in form of invocations of the added code's methods.

For the concrete deployment, we opted for utilizing the default AOSP *dex2oat* binary and leveraging its modularity to ship our extensions as separate libraries to the compiler suite instead of shipping *deployment app* with a statically linked *dex2oat* binary that includes our *ARTist* extensions. We use the LD_LIBRARY_PATH environment variable to ensure that our *dex2oat* loads and dynamically links our *ARTist* libraries, such as libart-compiler.so, from the assets directory of the *deployment app*.

# III. Intra-App Taint Tracking Case Study

We study the viability of our *ARTist* approach through a case study by re-instantiating intra-application taint tracking, as demonstrated by the seminal TaintDroid [1] work, in the form of a new *ARTist Module*, called *TaintARTist*. In contrast, to the original TaintDroid work, which was based on instrumenting the now abandoned Dalvik virtual machine (DVM), our approach abstains from instrumenting the dex execution environment and instead builds on inlining taint tracking logic into the application code base at compilation time.

We first exploit the processing features of the *dex2oat* compiler to detect the data flow sources and sinks and afterwards use its static analysis features to overapproximate the relevant data flows that have to be instrumented. Second, we make use of our data flow analysis to only inline taint tracking code where critical flows *can* happen at runtime. The result is a intra-application taint tracking prototype based on *ARTist* that runs solely on the application layer for Android versions 6+.

# References

[1] 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 *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[2] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proc. 18th ACM Conference on Computer and Communication Security (CCS'11)*. ACM, 2011.

[3] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: supporting operation modes on smartphones," in *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12)*. ACM, 2012.

[4] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," in *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE Computer Society, 2012.

[5] H. Hao, V. Singh, and W. Du, "On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android," in *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)*. ACM, 2013.

[6] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *Proc. 24th USENIX Security Symposium (SEC'15)*. USENIX Association, 2015.

[7] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proc. 5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'15)*. ACM, 2015.