ARTist - A Novel Instrumentation Framework for Reversing and Analyzing Android Apps and the Middleware

Oliver Schranz - CISPA Helmholtz Center i.G.

July 30, 2018

1 Introduction

With the introduction of Android 5 Lollipop, the novel Android Runtime (ART) superseded the Dalvik Virtual machine (DVM) as the new default runtime. One of the major changes is the new on-device compiler suite called dex2oat that provides ahead-of-time compilation of applications in contrast to the former bytecode optimization (dex-opt) and just-in-time compilation. While ART fundamentally changes the the way applications are optimized and executed, not much research has been done in this area. While there are some noteworthy exceptions [1][2], the area is still mostly uncharted. In this whitepaper, we explore the possibility to utilize the new runtime and in particular the dex2oat compiler to create a fully-featured instrumentation framework. The contribution of this work is two-fold. First, we want to tighten this gap by providing information on the compiler's (undocumented) internal workings. Second, we introduce ARTist (the Android RunTime instrumentation and security toolkit), a compiler-based instrumentation solution for Android that does not depend on operating system modifications and solely operates on the application layer. In contrast to existing instrumentation frameworks, it avoids invasive system-modifications and is meant for researchers, developers and end users alike. ARTist comes with a rich ecosystem including a toolchain to build own instrumentation modules, an application wrapping ARTist with an easy-to-use GUI that can be installed from an APK file, a framework for large-scale evaluation and example modules. Applications of ARTist range from app & system modding and analysis to more security-related topics such as inline reference monitoring, taint tracking, method hooking and library isolation, just to name a few.

1.1 Outline

In section 2, we provide the required background knowledge about Android, ART and in particular dex2oat. Related work is discussed in 3 and our system's

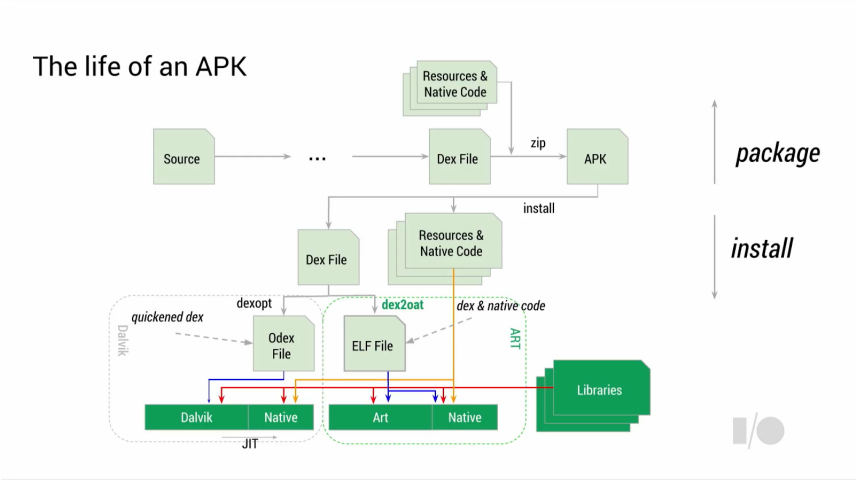

Figure 1: The lifetime of an APK (from Google I/O).

design and implementation are described in 4. In the end, we discuss the current project state and future in section 5 and conclude the paper in 6.

2 Background

Here we provide some background information on core parts of Android that we utilize in our work. Basic knowledge of Android is required.

2.1 App installation process

The way Android apps are built, delivered and installed includes numerous tools and involved parties on the way from the developer to the end user. We will have a brief look at each side to cover the basics of this process. Figure 1 gives an overview on the process.

2.1.1 The developer side

Developers mostly write their app code in Java & Kotlin plus optionally some pre-compiled native libraries (often written in C/C++). The Java and Kotlin code is first transformed to .class files, yielding classical Java bytecode. However, since Android used to run the specialized Dalvik Virtual Machine, the dx tool transforms Java bytecode to Dalvik bytecode and saves it in as few classes.dex files as possible. The resulting classes.dex files combined with pre-compiled native libraries and other resources form the APK file that is typically distributed through an app store.

2.1.2 The device side

Once downloaded to a device, installation happens in multiple steps. After resources are extracted, the dex code is optimized in different ways depending on the Android version. Pre-ART, the code was optimized using dex-to-dex transformations to create optimized dex files (odex). With the introduction of the new runtime, there are multiple possibilities now depending on the current state of the device. In full compilation mode, the code is now ahead-of-time compiled to native code that is stored in an ELF-based format called OAT. In recent Android versions, this happens when the device is currently charged and enough space is available. In other cases, however, a hybrid solution is applied by either compiling just some parts of the code ahead-of-time or even skipping the process as a whole and resorting back to interpretation and just-in-time compilation. In the latter case, the runtime will analyze the execution of apps and remember *hot code paths* so that the next time the device is deemed to be ready (charging, enough space, etc), profile-guided compilation translates those parts of the code that were found to be performance-critical. In general, the complexity of the dex code optimization increased for each version aiming at striking a balance between quick app start and execution times, delay at install time due to the compilation process and the additional memory consumption.

2.2 The Runtime

The runtime components of ART take care of loading and executing the OAT files compiled from bytecode of apps and system components. Pre-ART the Zygote process preloaded a set of often used classes (Android framework, crypto library, etc) into memory so that they are available in every fork, which is essentially all apps and many system components. Post-ART, the dex2oat on-device compiler compiles this set of classes into two files: boot.oat and boot.art. The former is the straightforward compiled version of the preloaded classes, the latter represents a heap of already allocated and preloaded objects to avoid costly allocation of them in each single forked process later. Essentially, Zygote provides a *warmed-up* but generic base process that is forked but not exec'd. The way that applications are executed is that after the fork, the app's compiled OAT file is loaded into the memory as a shared library, which fits Android's design of having event-based callbacks instead of a dedicated main function is apps.

2.3 dex2oat

ART's dex2oat is a fully-featured compiler suite with multiple compilation backends, code generators for all supported hardware platforms and a multitude of different options to fine-tune compilation. It is responsible to compile not only apps but also system components such as the systemserver (ActivityManagerService, PackageManagerService, etc) and the boot.oat/boot.art images.

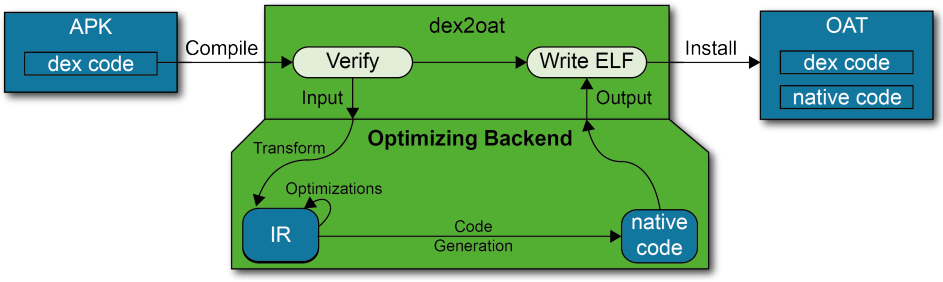

Figure 2: Compilation of an APK file using the Optimizing backend.

2.3.1 The compiler framework

While dex2oat provides the framework that loads and verifies the bytecode and later writes the output, the actual compilation is left for one of the available backends. Lollipop introduced three backends: Quick, Optimizing and Portable, each featuring distinct intermediate representations (IRs) and optimization strategies.

2.3.2 Quick

Meant as a quick transition from Dalvik to ART, the Quick backend is a straightforward translation of the just-in-time compiler to the ahead-of-time paradigm. While it was the default backend for apps in Lollipop, and for boot.oat/boot.art in Lollipop and Marshmallow, it was eventually superseded by the Optimizing backend because its design was not suitable enough for state-of-the-art optimizations[3].

2.3.3 Portable

Since its introduction, portable was never the default backend and was dropped eventually from the code base altogether. The idea was to utilize LLVM and its bitcode as a backend, but judging from the numerous blog posts, forum threads and questions on the mailing list, it seems it never reached the required robustness and performance goals.

2.3.4 Optimizing backend

In this work, we mainly focus on the Optimizing backend, which is the default for apps since Marshmallow and for other Java-based system components since Nougat. Figure 2 provides an overview of the whole process. After bytecode verification, the backend translates the code into its IR method control flow graps (CFGs) where optimization passes are executed. Eventually, the optimized IR CFGs are compiled to native code using the code generator for the architecture the device is running on. Multi-threaded compilation is possible since

methods are (mostly) compiled independent of each other and optimizations are designed as passes over the method CFGs. The intermediate representation resembles a per-method CFG made from nodes similar to dex instructions in a single static assignment form with explicitly computed def-use pairs. IR nodes are object-oriented representations of dalvik instructions (e.g., invoke-virtual, If) and additional *meta* instructions that are added to the code to uphold the Java semantics. The reason for the latter category of nodes is that the programmer writes her code in Java/Kotlin and hence expects the program to stick to the Java semantics. With the translation to native code however, faulty code might trigger native crashes such as segmentation violations instead of nicely-readable NullPointerExceptions. Meta instructions fill this gap by explicitly adding the java semantics into the code (e.g., check for array bounds, throw NullPointerExceptions by hand) to make it transparent to the developer whether the code runs in the Dalvik VM or under ART. More information on the IR can be found in my Master Thesis [4].

3 Related Work

While the fields is mostly uncharted, there are a few noteworthy works on different topics related to the Android Runtime. We will have a look at an excerpt that is close to our approach.

3.1 ART

In his Black Hat Asia whitepaper from 2015 *Hiding behind ART*[1], Paul Sabanal explains how to make use of the new runtime and OAT file format to create multiple user-mode rootkits that hide their presence on the device. The whitepaper provides a rich source of information about the OAT file format including its headers, checksums and general structure. In *Fuzzing Objects dART*[5], Anestis Bechtsoudis fuzz-tests the new compiler with automatically generated dex input to search for vulnerabilities and gives further information on the internal workings of dex2oat. Debuggability and the Quick backend are further detailed in a slide deck[6] from Linaro.

The CCS 2016 paper *TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime* by Mingshen Sun et al describes how to build a taint tracking system into the code generators of dex2oat, thereby providing valuable information on how apps are compiled and executed. This work is closest to our ARTist project because it also directly utilizes a fork of dex2oat but adds own code on a different layer and does not generalize to a full instrumentation framework.

3.2 Android Instrumentation

There are multiple systems providing instrumentation capabilities for developers and researchers seeking to analyze and modify Android apps and system

components, so we have a look at the most popular and common ones to be able to compare them to ARTist later.

Xposed is the most well-known hooking framework for Android and the go-to solution for modders[1]. Given root access and an unlocked bootloader, flashing Xposed means overwriting the app_process, which is the base process used by Zygote, to be able to load hooks for arbitrary methods. At the time of this writing, there is a huge community (in particular on the popular XDA community) behind the project and a long list of available modules created by modders. Assuming a power user that is capable of rooting a device and flashing customizations, then Xposed can also be used by end users without programming experience by simply downloading and installing ready-made modules from an online source.

Frida is a popular tool among analysts and reversers that allows for method hooking in multiple targets including binaries, Android and iOS apps[2]. Frida injects Chrome's V8 engine into target processes so that hooks can be written in Javascript. There are bindings available for multiple languages (e.g., python). While it is widely popular in the security community, it is mostly geared towards developers and researchers, and *not* suited for end users since Frida is mostly controlled through an ADB connection and the Javascript bridge is too costly for productive usage.

4 System Design

In this section, we discuss the various aspects of the ARTist projects, from the underlying goals and design decisions to the ecosystem built around it.

4.1 Goals

The ARTist instrumentation framework is designed around three major goals that are derived from what existing tools can and cannot provide. We will first discuss the three goals and in the remainder of the design section explain why and how ARTist is meeting them.

Deployability: Ease of deployment is key for adoption and a driving factor for reaching a critical mass and form a community. Often, Android security solutions create custom versions of the Android Open Source Project (AOSP), which can provide better guarantees because they can utilize elevate privileges, but they are hard to deploy since they either require manufacturers to implement their solution in their own Androids or end users to flash those custom ROMs to their devices. There is a whole line of research that focuses on application-layer only solutions for Android that aim at striking a balance between achieving

[1] `http://repo.xposed.info`
[2] `https://frida.re/`

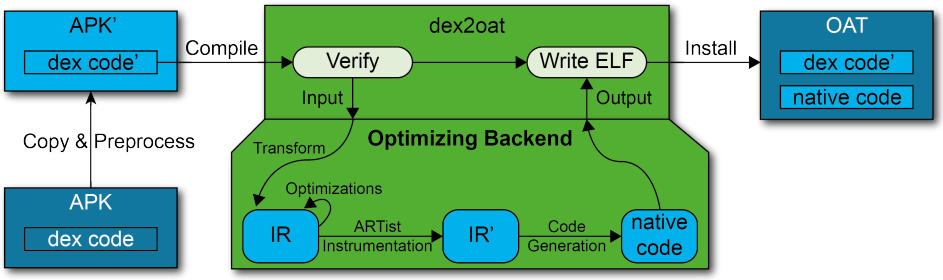

Figure 3: Instrumentation of an application using ARTist.

results similar to system-centric solutions and still providing better deployability and hence usability. On the one hand, for example, Xposed is very powerful since it can hook arbitrary methods in apps and other system components, however setting up Xposed is not an easy task for the laymen user. Frida on the other hand can additionally hook native methods but is only deployable by experts and not suitable for users at all.

Invasiveness: Existing approaches differ a lot in their invasiveness, i.e. the amount to which they are changing the operating system and ecosystem to achieve their goal. In general, the more invasive an approach the less robust it might become since modified or replaced system components increase the likelihood of incompatibilities or failed assumptions. Xposed, for example, is more invasive than Frida since it replaces the app_process, which is an integral part of the Android runtime environment in contrast to the in-memory changes inflicted by Frida. From a design perspective, it is preferable to achieve the required functionality while being the least invasive.

Granularity: The instrumentation granularity is one of the subtle differences between hooking and instrumentation frameworks. While most approaches focus on function hooking, a finer-grained solution aims at instructions instead of functions. While, strictly speaking, one is not more powerful than the other, it definitely makes a difference for developers since allowing for instrumentation on a finer granularity makes it easier to implement subtle modifications. Xposed and Frida both operate on function granularity, meaning it is possible to execute code before, after and instead of a given function. Changing single instructions, however, is not easily possible.

4.2 Overview

Figure 3 gives an overview of the instrumentation and re-compilation of an application using ARTist. First, in a preprocessing phase an arbitrary Java/Kotlin library is merged into the target code. This allows for shipping complex implementations such as reference monitors. Second, the modified target is then

provided as an input to our extended compiler. Third, after the translation to IR CFGs and the applications of the backend's own optimizations, ARTist applies its instrumentations that are disguised as optimization passes over the code. Since optimizations are expected to change the code, the compiler stays oblivious to the fact that we are actually execution instrumentation passes at this point. Fourth, after the resulting modified IR is compiled to native code and written to an OAT file, we replace the OAT file originally created by the system compiler at install time with our newly created version. From this moment, whenever the target is launched again, the new OAT file is loaded and hence the instrumented version of the code is executed.

4.3 Deployment options

Given that we introduced deployability as one of the major design decisions underlying our system, we will have a look at the two different deployment mechanisms for ARTist and the different trade-offs they are making.

4.3.1 App based

The most common and least invasive deployment strategy for ARTist is the app-based installation. It is based on the fact that our extended compiler consists of a binary and some companion shared libraries, which we can simply ship as assets within an Android application. Our app called ArtistGui essentially wraps the binary ARTist compiler with an easy-to-use graphical user interface to make it accessible and usable for developers and end users alike. It further provides convenience functionality, such as a management utility for importing and selectively applying instrumentation modules, keeping apps instrumented upon app updates and removing instrumentations to restore the original, unchanged functionality of a target. Reconsidering the process from Figure 3, this approach is non-invasive since we ship our own compiler instead of replacing the existing one and we only replace the compiled OAT file of an app. This requires root but no further operating system changes or deeper modifications, and hence constitutes a superior deployment model compared to existing solutions. Currently, this approach is used to instrument applications installed on the device. However, we are currently working on also instrumenting other parts of the OS such as the systemserver or the `boot.oat` from within the app.

4.3.2 Custom ROM

In order to not only instrument apps but also system components, there is an alternative deployment path that is more intrusive than the app-based strategy. When compiling a custom ROM, replacing the ART project with our custom ARTist version allows for compiling and hence instrumenting all system services and `boot.oat` as well because it essentially replaces the system compiler. Due to its increased invasiveness, this path is only eligible for system developers and

researchers, and eventually we want to make this possible without requiring to compile a custom ROM.

4.4 Ecosystem

In its current state, ARTist is not only an extended version of the dex2oat compiler but a full ecosystem revolving around compiler-based instrumentation on Android. In this section, we will highlight some of the major parts and their responsibility in the bigger picture.

4.4.1 ARTist

As depicted in figure 3, ARTist essentially is an extension to dex2oat's optimizing backend that is capable of executing custom instrumentations disguised as optimization passes. It dynamically loads developer-provided instrumentation modules and applies them according to the given instrumentation filters and policies. It consists of two repositories: art and artist. The art project provides a specialization for each supported Android version that can interface with artist, i.e. load instrumentation modules from provided command line arguments and patche OAT checksums. The artist repository contains all the boilerplate code for modules to interact with the method IR CFGs, such as our simple *injection framework* where you can simply declare the position and the target function of a new method call to be injected into the target. By getting access to the whole method CFG, modules can operate on the instruction level instead of the common method granularity of existing approaches.

4.4.2 Modules

Modules represent the actual instrumentation logic applied by ARTist. They consist of three major parts:

1. The first part is an optional and developer-provided library that is merged into the target during preprocessing. It is best practice to implement the actual business logic here and connect it to the target later. Basically, this library that we call *Codelib* is regular Android app project that can be created using the default tools like Android Studio.

2. The second part consists of the instrumentation passes that will connect the target to the merged library. These are written in C/C++ and utilize the boilerplate code the artist repository provides to interact with the IR code. It is compiled with our ARTist Module SDK that contains the AOSP's compiler toolchain and required headers and libraries from art and artist. While it would be possible to use this for assembling new functionality within existing target methods, it is recommended to use the instrumentation passes only to glue together the target and your library and keep all the complex functionality within the *Codelib*.

```
vector<shared_ptr<const Injection>>
    HTraceArtist::ProvideInjections() const {
  vector<shared_ptr<const Parameter>> params;

  vector<shared_ptr<const Target>> targets;
  auto target = make_shared<const Target>(
                      Target::GENERIC_TARGET,
                      InjectionTarget::METHOD_END);
  targets.push_back(target);

  auto injection = make_shared<const Injection>(
                      TraceCodeLib::TRACELOG,
                      params,
                      targets);

  vector<shared_ptr<const Injection>> results;
  results.push_back(injection);
  return results;
}
```

Figure 4: The code of our trace module's instrumentation pass.

3. The third part consists of some additional meta information, such as the module maintainer, the compatibility version for the ARTist ecosystem, title and description texts, and the module's current version. All three parts are assembled into a zip file that constitutes the final ARTist module. It can be used by pushing it to a device where it is loaded by our ARTist wrapper called *ArtistGui* (cf. below for details) and provided as an argument to the wrapped ARTist compiler version.

Our module design covers a broad range of use cases, from lightweight static analysis to complex in-depth instrumentations. To this end, we showcase some example modules to provide an intuition what is possible with ARTist:

Method Tracing One of the most simple use cases is the trace module that we use as our go-to pick for debugging and testing. All it does is injecting method calls into (almost) all methods of a target where the invoked method uses stack inspection to find and print the currently active method name to the log. Figure 4 depicts the code of the described instrumentation pass. In a nutshell, we utilize our *injection framework* to declaratively describe what we want to inject (calls to the *tracelog* method in our *Codelib*) and where we want the injected method invocation to be placed (at the end of each target method). Combined with the filter definition in Figure 5 that tells ARTist to apply this to all methods but the blacklisted ones and the (straightforward) *Codelib* implementation of the *tracelog* method, this module generates logcat

```
// skip android support lib ui methods
// since they bloat up the log
unique_ptr<Filter> TraceModule::getMethodFilter() const {
  const vector<const string> blackListDefinition = {
    "android.support.",
  };
  return unique_ptr<Filter>(
      new MethodNameBlacklist(blackListDefinition));
}
```

Figure 5: The filter definition of the trace module that removes support library methods from the instrumentation scope.

Figure 6: Logcat dump of the *heise online* app instrumented with the trace module.

outputs similar to those in Figure 6. While it is not the most useful module we created, it serves as an easy example for how fast one can get up and running with ARTist.

Inline Reference Monitoring As a part of our initial ARTist EuroS&P paper [7], we created a bare-bones implementation of an inline reference monitor that showcases dynamic permission monitoring. The basic idea is to utilize a permission mapping that lists permissions required for using public Android API methods so that we can inject a call into our reference monitor implemented in the *Codelib* immediately *before* such an API is invoked by the target. In the *Codelib*, an arbitrary policy can then be enforced, e.g., ask the user for confirmation or take a predefined policy, and abort the target if a violation is detected. This approach can refine the dynamic permission system introduced in Marshmallow by providing the user with more flexibility and also allowing to restrain internet access and other permissions that are not supported for official revocation.

Stetho One of the more interesting modules utilizes facebook's open source stetho library[3] that is meant to be included in the debug builds of applications. It connects the app to the chrome developer tools so that it can be debugged

[3] `https://facebook.github.io/stetho/`

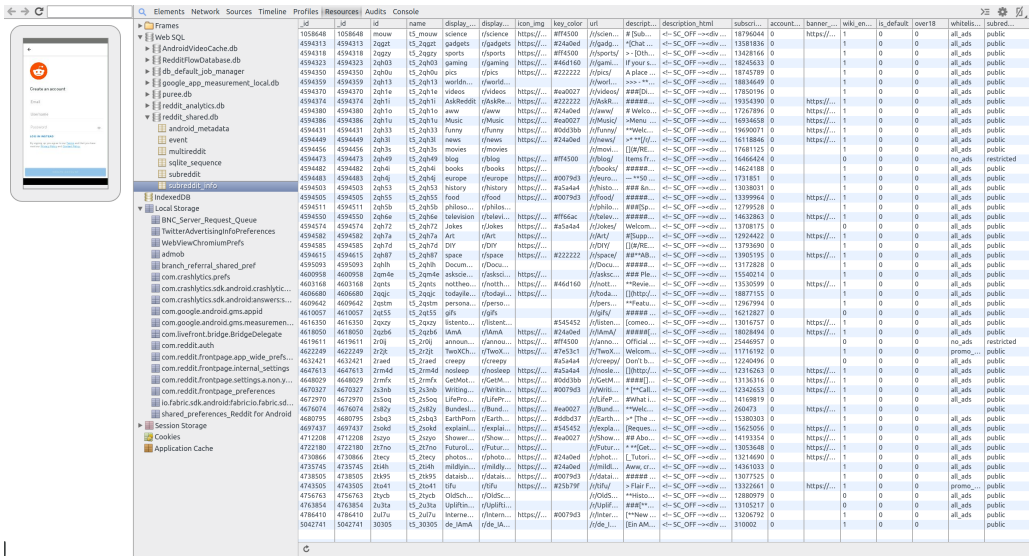

Figure 7: Stetho providing access to the reddit app's on-device files and databases.

like a web page, including traffic interception, file and database inspection and modification, and invocation of Javascript in the app context. We created a module that injects this into third-party apps from the play store, effectively allowing to debug arbitrary applications. Figure 7 shows the chrome developer tools connected to the reddit app.

Rootkit In his black hat asia talk [1], Paul Sabanal described how to create rootkits by utilizing ART's OAT files and their (almost) independence of the input they are generated from. The same measures mentioned in the talk and whitepaper, for example changing the list of active processes by modifying the smali code and re-compiling it to dex bytecode, can be automated completely using an ARTist module.

Traint Tracking Taint tracking is an information flow mechanism that marks (*taints*) data within the target and observes the flow of information to detect what other data is influenced by it. It is often used to detect multiple forms of leakage of sensitive data. We outlined the prototype of an intra-app taint tracking system for Android apps as a part of the initial ARTist paper [7]. Figure 8 describes how we define and instrument method-local and global sources and sinks to be able to trace the flow of information across multiple app methods. More in-depth information can be found in the paper.

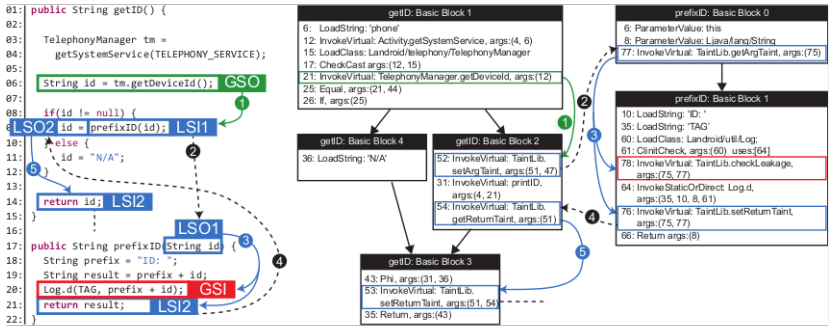

Figure 8: Two simple methods, their corresponding (simplified) IR CFGs and the annotated local and global sources and sinks that we use to track tainted data.

Test Coverage ARTist can be utilized to compute coverage information when testing arbitrary third-party apps. A simple module generates an identifier for a basic block within a method, assigns it an ID that is injected into the code as a constant, and adds a call to the *Codelib* into this block that provides the corresponding ID as an argument. In the *Codelib*, we can now see a continuous stream of IDs that we can map back to methods and their basic blocks. This is particularly interesting when testing applications with automated test drivers.

4.4.3 ArtistGui

ArtistGui, the ARTist Graphical User Interface, is the wrapper application that provides a layer of convenience around the ARTist version of `dex2oat`. It is used as a management tool to import and use ARTist modules, start the instrumentation process for applications, keep them instrumented across app updates (if the user wishes so) or remove the instrumentation altogether. Figure 9 shows three screenshots from the app that outline the workflow of instrumenting an app. After modules have been imported from the file system, they are available in the instrumentation dialog when tapping on an installed application in the list. Eventually, the app will also automatically download the correct ARTist version, display and load new modules, and keep everything updated.

4.4.4 Dexterous

The preprocessing shown in Figure 3 involves partially merging module *Codelibs* into the target dex files. The reason is that while we can change the code of existing methods within the compiler, we cannot add new ones since a lot of required information such as method IDs refer back to the dex files's internal structures. Therefore, we need to add our own code to the target's APK or JAR file. However, this is not enough since Android resolves invoked methods via a dex file-local method ID. But since our new methods are not called in

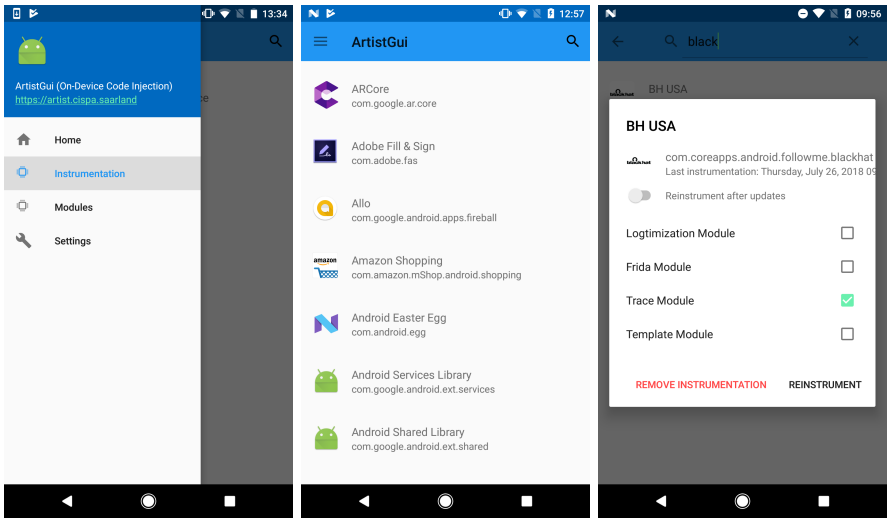

Figure 9: Left: ArtistGui navigation drawer. Center: Installed apps overview. Right: App instrumentation dialog.

the original target, they are not in the method ID tables of the existing dex files. This problem is solved by our *Dexterous* tool, which is an extension of the DexMerger facility. In contrast to the original version that merges multiple dex files together, we do not merge code but just the required headers including type, class and method IDs. In a nutshell, we add all chosen *Codelib* methods and types to the headers of all target dex files but do *not* merge their code. The *Codelibs* are added as dedicated dex files to the target's APK or JAR file. *Dexterous* can be used as an Android library (as done in ArtistGui) or as a standalone tool (as done for the custom ROM deployment path).

5 Project State and Future

Even though the project started in 2015 already, the open source instrumentation framework that we are using today is quite new and there is still a lot to be done. We recently entered the beta phase with the release of a new documentation[4], proper versioning for GitHub releases and installable module SDKs, and we are looking forward to the feedback we will receive. The goal is to build a community around the ARTist ecosystem with developers contributing modules and code for the core projects, as well as users giving feedback. We hope that this whitepaper gives an introduction to what ARTist is capable of and we are actively listening in our Gitter chat[5] for questions and opinions. If you are interested, feel free to drop us a message.

[4] `https://artist.cispa.saarland/`
[5] `https://gitter.im/project-artist/Lobby`

5.1 Future Work

There is a lot that can be done with ARTist. In this section, we focus on four main aspects that can be improved in the foreseeable future. First and foremost, we want to create further interesting modules that are also useful to the bigger community. It seems that in particular the security community has some special requirements, such as encrypted traffic inspection and interception, that requires specialized tools like ARTist.

Second, assuming there will be more modules available from us and the community, we want to have a module store where developers can upload their own creations and users can inspect and download them straight into ArtistGui. This has been done for Xposed and we envision a similar approach here.

Third, given that we are currently in beta state, the existing tools in our ecosystem can still be improved a lot by, e.g., adding more automation. This includes the build time where continuous integration and automated builds and tests can increase the code quality, as well as automatically downloading and updating ARTist versions and modules through ArtistGui.

Fourth, we want to (mostly) get rid of the system-centric deployment. While this is a valuable option for seasoned system developers and researchers, the toolchain required for building AOSP as a whole is very involved and the whole process requires a lot of time to setup. At least for the typical ARTist use cases here, i.e. instrumenting the systemserver and *boot.oat*/*boot.art*, it might be possible to avoid this. Since those components are re-compiled on the device anyway for each OTA upgrade, it should be possible to trigger this directly from ArtistGui, therefore dropping the custom ROM requirement and strengthening our app-based deployment path.

6 Conclusion

In conclusion, the novel Android Runtime provides really interesting opportunities for (security) research and our instrumentation framework ARTist built on top of the dex2oat on-device compiler is a straightforward way to get started with the topic. It can be deployed to rooted stock ROMs by just installing the ArtistGui app, allows for on-point instrumentations on the instruction level, abstains from changing the OS and provides performance improvements over similar approaches by shifting the instrumentation ahead-of-time. ARTist is still a young community project, so if you want to get involved, this is the perfect time to step up and get a say in the project.

7 Acknowledgements

Looking back, I want to acknowledge my team from CISPA that worked with me on ARTist at different stages of the project. First, I want to acknowledge all co-authors of the original EuroS&P paper: Michael Backes, Sven Bugiel, Philipp von Styp-Rekowsky and Sebastian Weisgerber. Second, a shout out to

Jie Huang, the main author of the 2017 CCS app compartmentalization paper [8] that heavily relies on ARTist and showcases this really interesting use case. Third and finally, a big thank you to those still actively involved in pushing the ARTist ecosystem further and developing new modules and features: Sebastian Weisgerber, Parthipan Ramesh, Alexander Fink and Maximilian Jung.

References

[1] P. Sabanal, "Hiding behind ART," Online: https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf, 2015.

[2] M. Sun, T. Wei, and J. C. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016.

[3] A. Authority, "Optimizing Compiler the evolution of ART," Online: https://www.androidauthority.com/art-optimizing-compiler-605011/, 2015.

[4] O. Schranz, "Towards Compiler-Assisted Taint Tracking on the Android Runtime," Master's thesis, Saarland University, December 2015. [Online]. Available: https://publications.cispa.saarland/1001/

[5] A. Bechtsoudis, "Fuzzing Objects d'ART: Digging Into the New Android L Runtime Internals," Online: http://census-labs.com/media/Fuzzing_Objects_d_ART_hitbsecconf2015ams_WP.pdf, 2015.

[6] Linaro, "HKG15-300: Art's Quick Compiler: An unofficial overview ," Online: https://de.slideshare.net/linaroorg/hkg15300-arts-quick-compiler-an-unofficial-overview, 2015.

[7] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android Runtime Instrumentation and Security Toolkit," in *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017.

[8] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017.