2017 IEEE European Symposium on Security and Privacy

ARTist: The Android Runtime Instrumentation and Security Toolkit

Michael Backes*†, Sven Bugiel*, Oliver Schranz*, Philipp von Styp-Rekowsky*, Sebastian Weisgerber*

*CISPA, Saarland University
Saarland Informatics Campus

†Max Planck Institute for Software Systems
Saarland Informatics Campus

Email: {backes, bugiel, schranz, styp-rekowsky, weisgerber}@cs.uni-saarland.de

***Abstract*—With the introduction of Android 5 Lollipop, the Android Runtime (ART) superseded the Dalvik Virtual Machine (DVM) by introducing ahead-of-time compilation and native execution of applications, effectively deprecating seminal works such as TaintDroid that hitherto depend on the DVM. In this paper, we discuss alternatives to overcome those restrictions and highlight advantages for the security community that can be derived from ART's novel on-device compiler *dex2oat* and its accompanying runtime components. To this end, we introduce *ARTist*, a compiler-based application instrumentation solution for Android that does not depend on operating system modifications and solely operates on the application layer. 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 called *Optimizing*. We document the results of this study in this paper to facilitate independent research on this topic and exemplify the viability of *ARTist* by realizing two use cases. In particular, we conduct a case study on whether taint tracking can be re-instantiated using a compiler-based app instrumentation framework. Overall, our results provide compelling arguments for the community to choose compiler-based approaches over alternative bytecode or binary rewriting approaches for security solutions on Android.**

1. 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 influencing works such as TaintDroid [1] for analyzing privacy-relevant data flows within applications, AppFence [2] for protecting the end-users' privacy, Moses [3] for domain isolation, or Spandex [4] for password tracking, 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 native code that is executed in the new Android runtime (short ART). While this leap did not affect the app developers, it broke legacy compliance of the previously mentioned security solutions that rely on instrumentation of the DVM and 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 [5], [6] or adapting to the novel but uncharted on-device compiler infrastructure.

Our contributions. In this paper, we present a compiler-based solution that can be used to study the feasibility of re-instantiating previous solutions such as dynamic, intra-application taint tracking and dynamic permission enforcement and that provides a more robust, reliable, and integrated application-layer instrumentation approach than previously possible. Concretely, we make the following contributions in this paper.

Uncovering the uncharted ART compiler suite. Since the novel ART compiler suite, *dex2oat*, is still uncharted, we uncover its applicability for compiler-based security solutions to form expert knowledge that facilitates independent research on the topic. In particular, we deep-dive into its most recent backend called *Optimizing* that became the default with Android 6 Marshmallow, and expose its relevance for *ARTist* as well as future work in this area.

Compiler-based app instrumentation. We design and implement a novel approach, called *ARTist* (*ART Instrumentation and Security Toolkit*), for application instrumentation based on an extended version of ART's on-device compiler *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. We thoroughly discuss further benefits and drawbacks of security-extended compilers on Android in comparison to bytecode and binary rewriting. Our results provide compelling arguments for preferring compiler-based instrumentation over alternative bytecode or binary rewriting approaches.

DOI 10.1109/EuroSP.2017.43

Feasibility study for compiler-based taint tracking. To demonstrate the benefits of a solution such as our *ARTist*, we conduct a case study on whether compiler-assisted instrumentation can be utilized to realize a dynamic intra-application taint tracking solution. Our resulting prototype is evaluated using microbenchmarks and its operational capability is shown using an open source test suite with known ground truth.

Infrastructure for large-scale on-device evaluations. In order to prove the robustness of our approach, we created an evaluation infrastructure that allows to scale our on-device dynamic app testing to thousands of apps from the Google play store. It is easily extendable and can be used to not only evaluate *ARTist*-based approaches but in general dynamic on-device security solutions.

Publication of results. To allow researchers to utilize and extend the results of this paper, we open-sourced the code of *ARTist* and our evaluation framework called *monkey-troop* at https://artist.cispa.saarland.

Outline. The remainder of this paper is structured as follows. In Section 2, we present the results of our study of the *dex2oat* compiler and its *Optimizing* backend. We analyze the requirements for an application-layer instrumentation solution in Section 3 and compare bytecode and binary rewriting with compiler-based approaches. We present our *ARTist* design in Section 4. Section 5 illustrates use cases for *ARTist*, followed by a more detailed case study on compiler-assisted taint tracking. We discuss limitations and future work of our solution in Section 6 and conclude this paper in Section 7.

2. Background

We provide general background information on Android's managed runtime to set the context of our compiler extensions (Section 2.1), and subsequently present technical background information on the compiler suite *dex2oat* (Section 2.2) and in particular on its *Optimizing* backend (Section 2.3).

2.1. Android Runtime

Android is essentially a Linux-based operating system with an extensive middleware software-stack on top of the kernel. The middleware provides native libraries, a feature-rich application framework that implements the Android SDK, and a managed runtime on top of which system as well as third-party applications and a small number of framework services are executed. The runtime executes bytecode generated from Java-based applications and Android's SDK components. The runtime provides the code executed within its environment with the necessary hooks to interact with the rest of the system, such as the operating system, the application framework services, or the native Android user space (i.e., components running outside the managed runtime). Every process executing an application runtime environment is usually forked from a warmed-up process, called Zygote, which has all the necessary libraries and a skeleton runtime for the app code preloaded.

Runtime prior to Android 5. On Android devices prior to version 5, the runtime consisted of the DEX bytecode interpreter (or Dalvik virtual machine), which was specifically designed for devices with constrained resources (e.g., register-based execution model instead of stack-based). It executes Dalvik executable bytecode (short dex), which is created from the Java bytecode of applications at application-build time. Thus, every application package ships the dex bytecode compiled from the application Java sources. Additionally, since Android version 2.2, Dalvik uses just-in-time compilation of hotspot code segments in order to improve the runtime performance of applications.

Runtime since Android 5. With Android 5, Google moved over from an interpreter-based app execution to an on-device, ahead-of-time compilation of apps' dex bytecode to native code that is executed in a newly introduced managed runtime called *ART*. This shift in the runtime model was intended to address the app performance needs of Android's user and developer base. The new compiler suite was designed from scratch to allow for compile time optimizations that improve application performance, startup time, battery lifespan, and also to solve some well-known limitations of the previous interpreter-based runtime, such as the 65k method limit[1]. In particular, Google made the *Optimizing* compiler backend, which was introduced as an opt-in feature in Android 5, the default backend in Android 6. In the following Sections 2.2 and 2.3, we will elaborate in more technical detail on this new compiler suite and in particular on the *Optimizing* backend.

Prior documentation of ART. Even though the Android source code is publicly available as part of the Android Open Source Project (AOSP), little attention has yet been given to ART from a security researcher's perspective. Paul Sabanal had an early look [7] at the Android Runtime right after its silent introduction as a developer option on Android 4.4 KitKat. Beside providing information on the ART executable file formats, the paper discusses the idea of hiding rootkits in framework or app code, assuming root access has already been granted. However, especially in its early phase, the Android Runtime has undergone frequent changes, which, unfortunately, has made this documentation outdated by now.[2]

Another work [8] focuses on fuzzing the new runtime with automatically generated input files in order to detect bugs and vulnerabilities. While providing some high-level overview on the compiler structure and its backends, it, unfortunately, omits any deeper information on the *Optimizing* backend.

Thus, this background Section also serves the purpose of filling a gap in the technical documentation of those new Android features.

1. http://developer.android.com/tools/building/multidex.html
2. E.g., there is a large version gap between the documented oat version 45 and the current version 79 at the time of this writing (February 2017).

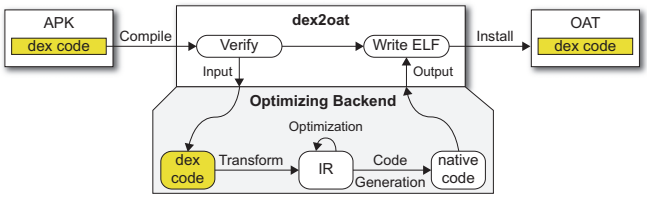

Figure 1: A high-level overview of the *dex2oat* compiler using *Optimizing* backend including the transformation to the IR, optimizations, and native code generation.

2.2. *DEX2OAT* Compiler Suite

Android's on-device compiler *dex2oat* is responsible for the validation of applications and their compilation to native code. It was designed from scratch to be highly flexible and of modular structure, providing numerous configuration possibilities, multiple compiler backends, and native code generators for supported Android platforms. The general workflow of the compiler suite is depicted in Figure 1 and its steps will be explained in the remainder of this Section. Providing a full technical documentation of the entire compiler suite and all its intricacies would unfortunately exceed the space limitations of this paper. Therefore, in this Section, we only focus on those parts relevant to this paper.

2.2.1. Input File Format. As an input format, *dex2oat* expects the very same dex files that DVM used to interpret. This strategical decision ensured that neither developers nor app store operators needed to adapt their code to ART. Developers still upload their apps as Android Application Package (APK) files that bundle the app's code with its resources. When a new app is installed on the device, *dex2oat* compiles the app's dex bytecode and the ART runtime executes it, which is completely transparent for the end user. Using this strategy, ART is still compatible with the old Android app base without enforcing a fallback to interpretation, which would loose all benefits that the new compiler provides.

2.2.2. Compilation. Before the actual compilation is performed, each input dex file is checked for validity. Those checks are more extensive and stricter than those implemented in the DVM in order to allow for state-of-the-art code optimizations. The compilation itself is done on a per-method base and can be parallelized. *dex2oat* delegates the actual compilation completely to the backend and only writes the results of the compilation to an oat file along with the original dex code. There are three compilation phases shared between all backends:

Transformation: A graph-based intermediate representation (IR) is created from the dex code. Depending on the actual backend, multiple IRs are possible.
Optimization: Given a populated IR graph, the code is optimized. Each backend provides its own set of optimization measures, ranging from very basic techniques to state-of-the-art algorithms.
Native code generation: The IR nodes are transformed to native code using a code generator for the specific CPU architecture of the current platform. The level of sophistication of the register allocation algorithm and implementation of the code generator depend on the backend.

Backends. On an Android stock device running version 5 (Lollipop) or higher, *dex2oat* can choose between two different backends, *Quick* and *Optimizing*. Although *Quick* was *dex2oat*'s default backend until Android 6, we focus, in the remainder of this Section and paper, on the newer *Optimizing* backend. This choice is not only motivated by the fact that *Optimizing* is the default backend since Android 6, but also by the fact that *Quick* is essentially derived from Dalvik and lacks a sophisticated IR that can support state-of-the-art compiler optimizations—including sophisticated security-oriented algorithms. However, *Optimizing* is designed completely from scratch and little is yet known about its internal structure and design. In Figure 1 the compilation steps of *Optimizing* are depicted. More insights on the inner workings of the new default compiler backend will be provided in Section 2.3.

2.2.3. Oat File Format. Oat files are Androids new file format for apps that are ready to be loaded and executed by the ART runtime. Even though the format was newly created for the Android platform, technically speaking oat files are specialized ELF shared objects that are loaded into processes, i.e., loading a compiled app into an application process is comparable to loading an (ELF) shared library into the process space of a dynamically linked executable. Besides the native code generated with *dex2oat*, oat files contain the complete original dex code, which is required to hold up consistency between the code that the developer wrote in Java, the dex code that used to be interpreted, and the compiled code, or to allow falling back to interpretation mode during app debugging.

2.3. Optimizing Intermediate Representation

We introduce insights into *dex2oat*'s *Optimizing* backend that we derived mainly from the AOSP source code of the ART project. *Optimizing*'s intermediate representation is essentially a control flow graph on the method level, which the Android developers denote as HGraph. It is further enriched with structural data about the program and populated with instruction nodes, denoted as HInstructions. Figure 2 presents an example Java code and Figure 3 presents the resulting[3] HGraph of the `getID` function in the *Optimizing* IR. We will come back to this example in our case study in Section 5.2.

3. Presented code is simplified and limited to relevant instructions for the sake of readability.

```
01: public String getID() {
02:
03:   TelephonyManager tm =
04:     getSystemService(TELEPHONY_SERVICE);
05:
06:   String id = tm.getDeviceId();
07:
08:   if(id != null) {
09:      id = prefixID(id);
10:   } else {
11:      id = "N/A";
12:   }
13:
14:   return id;
15: }
16:
17: public String prefixID(String id) {
18:   String prefix = "ID: ";
19:   String result = prefix + id;
20:   Log.d(TAG, prefix + id); // leak id!
21:   return result;
22: }
```

Figure 2: An example code snippet containing a leak of the device's phone number to the logging facility.

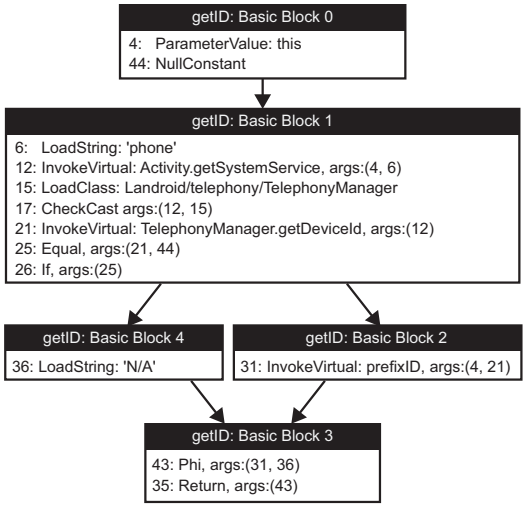

Figure 3: Generated IR in SSA form for the `getID()` method in Figure 2.

2.3.1. HGraph. The HGraph serves as the single intermediate representation of the app code. When the graph is created, dex instructions of the app's bytecode are scanned one by one and the corresponding HInstructions are created and interlinked with the current basic block and the graph. In order to allow for complex optimizations, the graph is transformed into *single static assignment form* (SSA). While this is done immediately beginning from Android 7 Nougat, Lollipop and Marshmallow devices first build up an alternative graph structure and subsequently transform it to the SSA graph we have now. Pairs of value definitions and usage, so-called *def-use-pairs*, are created during a liveness analysis and explicitly interlinked afterwards. At this point, *phi nodes* are introduced where static analysis cannot reliably decide which value will be assigned at a given position.

In this form, the graph is amenable to a multitude of possible optimizations. The available optimizations include algorithms such as *BoundsCheckElimination* to remove redundant bounds checks, *GVNOptimization* to remove duplicate code, dead code elimination, or *loop invariant code motion* to optimize hotspot code in loops. In Section 4, we will show how this form is also amenable to security-oriented instrumentation, thus supporting compiler-based security solutions on Android, such as dynamic taint tracking (see Section 5.2).

2.3.2. HInstructions. The HGraph nodes roughly correspond to dex instructions. Beside this transformation, nodes in the HGraph have additional attributes that have no equivalent in dex bytecode (e.g., an SSA index). The HInstructions distinguish between arguments and inputs. While the former correspond to the arguments given to an operator or method, the latter encode additional dependencies that may not be immediately observable given only the underlying dex code, as in the case of static method invocations that in addition to their arguments have an HLoadClass or HClinit as their input. All HInstructions share a basic set of information: Type, inputs, uses, id, and further data is attached to each node in order to ease the creation of and working with the HGraph. Each node is uniquely identified within the graph by its id that is assigned and incremented continuously during node creation. The type can be Void for methods that have no return value, Not for strings and object types of any kind, and additionally any of the Java primitive types. In order to get the actual object type, a fallback to the original dex file is required. This loose coupling between HInstructions and dex instructions as well as the presence of a method local dex program counter in each node show that the IR nodes are not completely independent of the original dex file.

Semantic consistency. In addition to the instructions that represent the original application logic, the HGraph also contains meta-instructions to preserve the semantic consistency between the original Java code of the developer, the dex bytecode shipped with APKs, and the native bytecode actually executed in ART. First, additional instructions are inlined into the graph to support meaningful debugging (e.g., to map from segmentation faults in ART to actual stack traces) and to conduct various forms of runtime checks (e.g., checking type casting, bounds checking, division-by-zero checks, or null pointer exceptions). Second, instructions to represent so-called *suspension points* are added, which effectively subdivide the application code into multiple chunks. Each suspension point between two chunks acts as a synchronization point between native code and original dex bytecode in the program execution and also serves as an entry point for garbage collectors or debuggers.

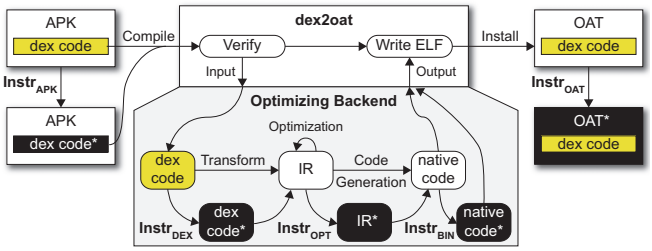

Figure 4: The code instrumentation points before, during, and after the compilation for different representations of the app code. Instrumented code is depicted in black boxes.

3. The Case for Compiler-Assisted Security on Android

A dedicated line of work, including the TaintDroid project and its derivatives (such as [2], [9]), relied on instrumentation of the now abandoned Dalvik virtual machine. As a consequence, the research community faces the dilemma on how to continue this line of work and is left with two choices (see Figure 4): Either compensating the missing runtime instrumentation through app rewriting techniques—dex bytecode ($\mathsf{Instr}_{\mathrm{APK}}$) or binary ($\mathsf{Instr}_{\mathrm{OAT}}$)—or by taking advantage of Android's new compiler suite ($\mathsf{Instr}_{\mathrm{DEX}}$, $\mathsf{Instr}_{\mathrm{OPT}}$, and $\mathsf{Instr}_{\mathrm{BIN}}$). Although dex bytecode rewriting is well-established in contexts such as inline reference monitoring [10], [5], [11], [12] and taint analysis [13], [14], and ART now supports porting binary rewriting techniques from commodity systems, we build in this paper on compiler-based instrumentation to not only re-instantiate previous approaches that relied on Dalvik VM instrumentation, but also to explore novel security solutions that leverage the compiler features.

In the following, we analyze the concrete requirements that an instrumentation solution should provide and discuss, for each of the above approaches (i.e., bytecode rewriting $\mathsf{Instr}_{\mathrm{APK}}$, binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$, and compiler-based instrumentation), their respective benefits and shortcomings in fulfilling those requirements. Table 1 provides a summary of our requirements analysis.

R1. Enforceable security policies. Each of the three approaches operates on one of the different representations of the same app code, i.e., bytecode, IR, or binary. Hence, all three approaches are identical in their capabilities of instrumenting the code and none of the solutions addresses any security policy alone.

R2. Strong security boundary. Both rewriting and the compiler-based approach rely on injecting monitoring code into the app's process space and can, therefore, not provide a strong security boundary between monitoring and (potentially) malicious app code (✗), e.g., native code. Thus, all of them can only provide security guarantees for at most honest-but-curious apps.

	Bytecode rewriting	Compiler-based	Binary rewriting
R1. Enforceable security policies	identical		
R2. Strong security boundary	✗	✗	✗
R3. Application layer only	✓	✓	✓
R4. User privilege only	✓	(✗)	(✗)
R5. Platform independence	✓	✓	✗
R6. Signature preservation	✗	✓	✓
R7. Robustness against optimization	✗	✓	✓
R8. Integrated approach	✗	✓	✗
R9. Supported versions	all	6+	5+

✓= fulfilled; ✗= not fulfilled

TABLE 1: Comparison of security and deployment features between bytecode rewriting, compiler-based instrumentation, and binary rewriting.

R3. Application layer only. All approaches can be implemented purely at application layer (✓). Deploying bytecode rewriting techniques $\mathsf{Instr}_{\mathrm{APK}}$ in form of separate apps has been presented in the literature [10], [5], [11], [12]. A compiler-based solution can be deployed as a separate app that ships and controls the security-instrumented compiler suite (see also Section 4.2). For both the compiler-based approach and the binary rewriting, the main requirement is access to the storage location of applications' oat files, which does not require system modification.

R4. User privilege only. While dex code is freely available for non-forward locked apps, accessing applications' oat files makes elevated privileges necessary. However, in Section 6.1.4, we discuss an approach that would allow both binary and compiler-based rewriting to circumvent this problem without requiring elevated privileges.

R5. Platform independence. Bytecode rewriting $\mathsf{Instr}_{\mathrm{APK}}$ (✓) and compiler-based instrumentation (✓) can be applied on all platforms supported by Android, since they modify the code before platform-dependent native code is generated. Binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$, in contrast, depends on the actual hardware architecture of the platform (✗), thus requiring extra effort to support different hardware platforms.

R6. App signature preservation. App signatures are the foundation of Android's same origin model that governs the app update policy and sharing of resources between apps, like a common process or UID. Consequently, modifying bytecode $\mathsf{Instr}_{\mathrm{APK}}$ and the resulting obligation to resign and repackage apps breaks this same origin model (✗). In contrast, compiler-based instrumentation (✓) and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ (✓) do not modify the original app package and therefore do not invalidate the signature.

R7. Robustness against code optimization. The instrumentation point determines whether any instrumented code will be subject to optimization at compile time. Applying optimization algorithms to instrumented code has the potential to interfere with the semantics of the modification through, e.g., instruction reordering, inlining, or similar techniques of state-of-the-art compilers like *Optimizing*. Current bytecode rewriting approaches $\mathsf{Instr}_{\mathrm{APK}}$ are applied before compilation; thus any instrumentation has to be robust against optimization by *Optimizing*—an aspect not yet further investigated by contemporary research (✗). On the other hand, binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ is restricted to instrumenting optimized code (✓), but misses the chance to reuse the rich *Optimizing* framework to also optimize added security code. The sweet spot is compiler-based instrumentation that provides full control over which optimizations are applied when and in which ordering (✓). Even better, this enables creating optimizations that are specifically tailored towards improving the instrumented code by utilizing the static program information that is present in the compiler.

R8. Integration into toolchain. Integrating an instrumentation system into an existing toolchain ensures perpetual development and fixes by the community as well as access to established and well-tested tools and frameworks. In this case, even though the ART project is open source and therefore open to the community, the compiler is mostly maintained by Google itself. Consequently, compiler-driven solutions that do not break with the toolchain's regular functionality benefit from the continuing improvements (✓). In the case of *ARTist*, the amount of code that needed to be changed is minimal and therefore easy to adapt for newer versions of the toolchain. Bytecode rewriting $\mathsf{Instr}_{\mathrm{APK}}$ and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ are developed separately from the toolchain and do not reap those benefits (✗).

R9. Version support. While bytecode instrumentation $\mathsf{Instr}_{\mathrm{APK}}$ can be applied to *all* Android versions, compiler-based approaches and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ depend on ART and therefore can only be applied since Lollipop (*5+*), where a compiler-based solution (as presented here) should utilize the *Optimizing* backend on Android *6+* in preference to *Quick*.

Sweet spot. In conclusion, comparing the security and deployment features that the three available instrumentation approaches provide, a compiler-based approach has very appealing properties and occupies a sweet spot among all approaches.

4. *ARTist* Design

In this section, we present the architecture of the *ART Instrumentation and Security Toolkit*. *ARTist* consists of two separate 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*.

4.1. Security-Instrumented Compiler

ARTist's *sec-compiler* enhances Android's *dex2oat* with additional instrumentation routines. This section will highlight design challenges and decisions we encountered while building *sec-compiler*, such as the placement of our instrumentation code within the compiler, the modification capabilities of our approach and the inclusion of custom libraries into target applications.

Choice of instrumentation point.

Given *dex2oat*'s modular design, multiple possibilities for the placement of app modifying code are immediately apparent. For instance, *dex2oat*'s design would easily allow porting bytecode and binary rewriting approaches ($\mathsf{Instr}_{\mathrm{DEX}}$ & $\mathsf{Instr}_{\mathrm{BIN}}$) into the compiler infrastructure (cf. Figure 4). Both techniques, although thoroughly studied in the literature, would benefit from instantiation within the compiler infrastructure by improving upon known shortcomings such as the requirement to re-sign modified applications. However, of the different choices, *ARTist*'s *sec-compiler* is concretely designed to operate on the intermediate representation of *dex2oat*'s *Optimizing* backend ($\mathsf{Instr}_{\mathrm{OPT}}$), where the existing optimization infrastructure and static code information in the *Optimizing* IR allow for efficient and precise code modification.

More precisely, our app instrumentation code is realized as an HOptimization, an optimization pass over the compiler's intermediate representation of the target app. As a consequence, our instrumentation pass neatly integrates into the compiler's optimization framework, including automated execution and access to the currently compiled method's HGraph. Implanting our instrumentation routines into the optimization workflow additionally grants us full control over the ordering and execution of optimizations in general, which opens up the opportunity for arbitrary reordering or even introduction of own optimization passes. Multiple such passes can combine different instrumentation routines or specifically crafted optimizations can improve the performance of our security code within target apps.

Generally speaking, the HOptimization interface's loose coupling allows for neat integration of new functionality into the compiler while, at the same time, keeping the effort for adaption of patches and maintainability to a minimum. We will refer to such independent extensions as *Modules* for the remainder of this paper.

Spotting instrumentation targets. HGraph supports the visitor pattern [15] that enables us to iterate over, inspect, and modify each single HInstruction of the app's code. In contrast to method hooking techniques, we can therefore operate at the instruction level. In *ARTist*, HGraphVisitors are primarily used to identify instrumentation targets and apply the desired modification. However, they can also be utilized to bootstrap static analysis. We will see concrete implementations using a visitor to collect instrumentation sites for our dynamic permission enforcement system in Section 5.1 and starting points for backward slicing in our taint tracking case study in Section 5.2.

Modification capabilities. With full access to a method's HGraph, *ARTist* can arbitrarily modify the target application's code, e.g. change inputs, types, or even remove, add, or replace instructions. *ARTist* even provides a dedicated API to inject arbitrary method calls into HGraphs, which, combined with the capability to inject whole libraries (see 4.2), allows to inject arbitrary code into target applications. *Module* developers simply declare the instrumentation location, the method to be invoked and the inputs to be passed. An example can be found in Section 5.1 where our dynamic permission enforcement use case *Module* makes heavy use of this feature.

All instrumentation routines operate on HNodes, meaning the dex frontend and native code generators stay agnostic towards our changes and can therefore be used as is. The result of this integrated solution is that we still take 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+.

Configuration. Using *Modules*, the instrumentation and modification process is already flexible. To further increase flexibility, *Modules* can, in turn, depend on policy configuration files that govern the instrumentation process. While the design of such policy files highly depends on the concrete *Module*, there are recurring and common patterns, for instance, the amount and type of instrumentation targets that should be detected, as demonstrated in Section 5.1. In general, this allows adaptation of existing instrumentation solutions to new targets or provisioning them with new security policies.

4.2. Compiler Deployment App

There are multiple possibilities to apply *sec-compiler* to installed apps on an Android device. A naïve approach simply replaces the system's *dex2oat* and corresponding libraries with *sec-compiler*'s augmented ones. With this strategy, each app installed on the device is automatically compiled using our custom *dex2oat* and therefore instrumented accordingly. However, we choose an alternative approach that does not require system modification. Based on the fact that *dex2oat* and its libraries are regular dynamically-linked ELF binaries, we can ship our custom compiler and its libraries as assets in our *deployment app*. On the device, *deployment app* manages and exposes *sec-compiler*'s instrumentation capabilities through an easy-to-use user interface. When the user chooses an application to be instrumented, *deployment app* executes our custom compiler to create an alternative version of the app's oat file, which we denote as oat' (*oat prime*).

Executing the compiler. Instead of shipping *deployment app* with a statically linked *dex2oat* binary that includes our *ARTist* extensions, we opted for utilizing a copy of Android's default *dex2oat* binary and leveraging its modularity to ship our extensions to the compiler suite as separate libraries. 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*.

Executing instrumented code. In order to execute the instrumented app, we need to trick the system into loading our instrumented file oat'. Since 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) to replace the oat file. We discuss alternatives to the naïve approach in Section 6.1.4, which abstain from extended privileges by using app virtualization or reference hijacking. The user, however, stays agnostic to this change since she is still able to launch the applications as usual, e.g. using the app launcher, but triggering execution of the instrumented version instead.

Inlining custom code. While *sec-compiler* can inject arbitrary code on the instruction level, in some cases it is preferable to inject a complete custom library into the application and wire it with the existing code using *sec-compiler*'s instrumentation capabilities. For example, both use cases in Section 5 rely on this feature to inject their companion libraries. The *deployment app*, in a preprocessing step, utilizes the DexMerger utility to combine the app's original bytecode with the additional code library. During compilation, connections between original and new code are built in form of invocations of the added code's methods.

5. Use Cases

We demonstrate the applicability and usefulness of our system by discussing several use cases out of which we exemplarily realized two as *ARTist Modules*. First, we implemented an Inline Reference Monitor (IRM) injection *Module* to allow for dynamic permission enforcement. Second, we conduct a case study on realizing intra-app taint tracking through inlining of taint tracking code. In addition, we discuss further ideas for *ARTist Modules*.

5.1. IRM for Dynamic Permission Enforcement

In the literature, Inline Reference Monitoring (IRM) is mostly implemented by modifying the bytecode before the installation [10], [5] or by hooking into an application's method at the caller or callee side at runtime [12]. By utilizing a security-instrumented compiler, IRM can be implemented without the need to resign and repackage apps as it is required by established approaches. Moreover, *dex2oat*-based IRM can operate at instruction granularity instead of at the method level. Those capabilities are showcased by our IRM injection *Module* that allows for dynamic permission enforcement, as shown by [12], [10], [16] on Android versions before Marshmallow.

The module is split into two distinct parts: the code injection routine that will inline permission enforcement code and the accompanying library that acts as a policy

decision point. While the first directs the instrumentation process at installation time, the latter enforces the user's policy at runtime.

Code injection. We first utilize *ARTist* to locate the call sites of permission-protected SDK methods that are defined in a policy configuration file. Afterwards, *ARTist* injects additional calls to our companioning library right before the call sites to check whether the critical method invocations should be allowed. This ensures that the control flow is diverted to our policy decision point before the execution of permission-protected methods.

The limitations imposed by the choice of this rather basic strategy are discussed in Section 6.2.2.

Policy decision point. The library that our *Module* injects into target apps provides an API to check the app's current state of permissions. Based on the given user permission policy, the library either allows or rejects the execution of a protected SDK method.

5.2. Case study: Taint Tracking

Established approaches for dynamic taint tracking on Android [1] rely on instrumenting the by now scrapped DVM for intra-application taint tracking or directly rewrite bytecode [13], [14]. In this case study, we explore the applicability of a compiler-based instrumentation framework like *ARTist* to re-instantiate intra-app taint tracking for applications on Android version 6 and higher. That is, through a prototypical implementation, we want to investigate whether inlining taint tracking logic into the application code base with *ARTist* at compilation time can be a surrogate for solutions prior to Android version 5. Note that this case study does not aim to be a full replacement of existing solutions like TaintDroid [1], but demonstrates a new potential foundation for future taint-tracking on Android.

5.2.1. Module Design. In general, we want to track information as it flows through the code using tracking logic inlined by a new HOptimization in the *Optimizing* backend. However, simply assigning each single value that should be tracked a taint tag and updating the tag for each single instruction operating on it will incur a major performance penalty. To minimize the runtime impact, we split our approach into two phases: *analysis* and *instrumentation*. During the *analysis* phase, we identify flows of tainted information between *sources* and *sinks*. By restricting ourselves only to those relevant flows of the values we are interested in, we avoid generating irrelevant but costly taint tracking code for parts of the method that never actually influence the data that is observed and gain noticeable performance improvements over more naïve taint tracking. During the *instrumentation* phase, code will be inlined that creates, propagates, and checks the taint values along the identified data flows. Our combined analysis and instrumentation achieves flow-, path-, object-, and context-sensitive taint tracking.

While [13] and [17] also utilize static analysis to optimize and guide the instrumentation process, both assume a holistic view on the application in form of a control or data flow graph. In contrast, *dex2oat* backends operate on a per-method level, leaving the primary challenge for our taint tracking *Module* to achieve similar tracking properties while inspecting one method at a time. A naïve solution to this problem would be to retrofit the compiler suite to provide an application-wide view and instrumentation. However, our prototype demonstrates how we can still achieve taint tracking for the whole application while restricting ourselves to a per-method view and instrumentation. To this end, we introduce in the following a new design for storing and propagating taint tags, in particular we have to refine the definitions of *sink* and *source*.

5.2.2. Analysis Phase. In order to optimize the instrumentation with taint tracking code, we exploit the processing features (e.g., HGraph's Visitor [15] pattern support) of the *dex2oat* compiler to detect the data flow sources and sinks and afterwards use its static analysis features to identify the relevant data flows and the operations along those flows that have to be instrumented.

Refining source and sink definition. The literature on taint tracking for Android defines sources and sinks as the API methods that input privacy-sensitive information into the application process (e.g., framework functions that return sensitive data, such as the location or telephony API) or, respectively, leak privacy-sensitive information from the application process (e.g., file handles, Internet sockets, or logging facilities). Since *dex2oat* is operating on a per-method level, we cannot assume that our analysis is able to always connect a sink and a source (e.g., when they are located in different methods). To address this problem, we have to connect the data flows of tainted variables across the different methods while maintaining the per-method-based analysis. To this end, we introduce, in addition to the above mentioned sinks and sources from the literature—in the following denoted as *global sinks/sources*—new *method-local sinks/sources*, more precisely HInstructions, which form the entry and exit points for inter-procedural data flows. Thus, global sinks and sources are points of interest for taint tag creation and check, respectively, while local sinks and sources are for inter-procedural tag propagation. For local sinks and sources, we differentiate between three categories each: *Local sources* include parameters provided to the current method (LSO1), return values from method invocations (LSO2), and values read from fields (LSO3). Conversely, *local sinks* are method invocations that leak values through its arguments from the current method (LSI1), return statements of the current method (LSI2), and field setting instructions (LSI3). At the beginning of the analysis phase, we collect all sinks within all methods and, in a subsequent step, detect all relevant sources for those sinks (see next paragraph).

Creating intra-procedural data flows. For each global and local sink collected in the current method, we create a backward slice by tracing back the sink's inputs until a source or constant is reached. Constants cannot be tainted

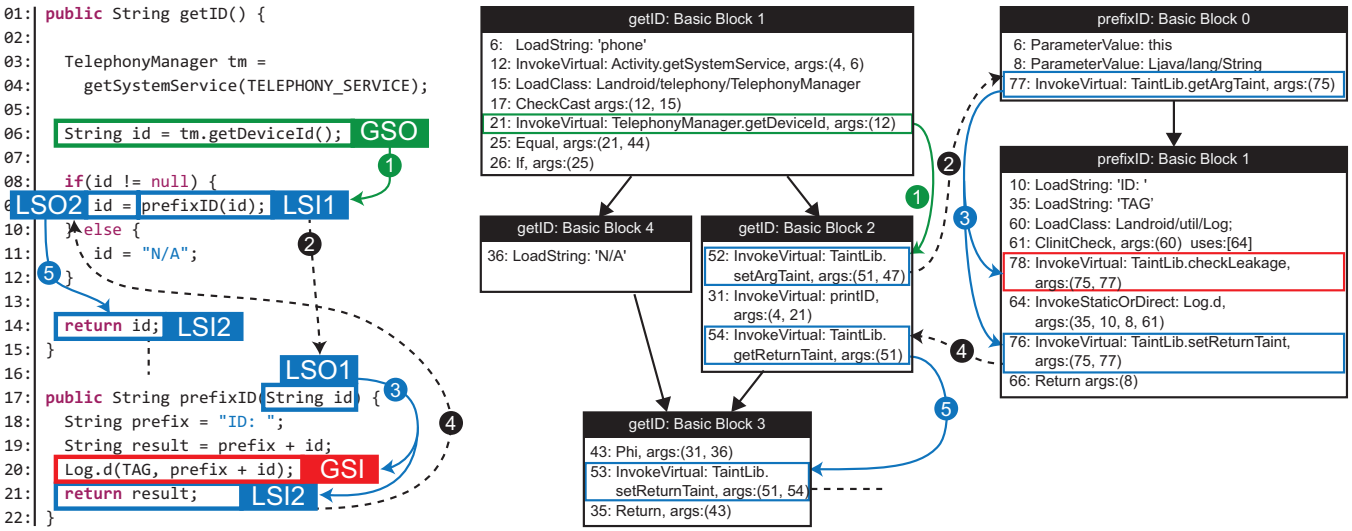

Figure 5: Tracking tainted variable *id* from example of Figure 2. All discovered sinks and sources are marked. Solid lines indicate intra-procedural data flows of tainted variables, dashed lines inter-procedural data flows between local sink-source pairs. The right-hand side depicts the inlined taint tracking code to propagate taint tags.

and are therefore omitted, the found sources however define which data can potentially leak through the sink, so our slice is defined by a sink and all its influencing sources appearing in the currently analyzed method.

For instance, Figure 5 continues the example code from Figure 2. In the Java code on the left-hand side, all sinks have been identified (i.e., the parameter `id` passed to function `prefixID` in line 9 is a local sink of type LSI1, the return statements in lines 14 and 21 form local sinks of type LSI2, and in line 20 the call to `Log` forms a global sink). Using backwards slicing (solid lines (1), (3), and (5)) the local sources in lines 17 (LSO1) and 9 (LSO2) as well as the global source in line 6 (`getDeviceID` call to retrieve device's phone number) have been identified. Each resulting backward slice is defined by its starting point (i.e., the sink) and all found endpoints (i.e., the sources). Together, those backward slices form the input for the instrumentation phase.

Because the backward slice dictates the targets for the instrumentation phase, high precision is desirable for improved runtime performance, but soundness is crucial since missed data flows result in false negatives. Consequently, our slicing algorithm over-approximates to compensate for known shortcomings of static analysis like missing runtime information, e.g., when encountering *phi* nodes or native method invocations. While over-approximating *phi* nodes by tracing back *all* its inputs is sound, handling native code is more involved. Our heuristic assumes that all data provided as an argument to a native function will influence its result, hence the return value taint is the combined taint of all inputs. However, this is sound for side-effect-free (pure) functions only. Native code in general, as well as reflection, are limitations we share with similar approaches as ours.

5.2.3. Instrumentation Phase. During the instrumentation phase, we inline code that creates taint tags for global sources and that checks taints at global sinks at runtime. Additionally, we inline code that inter-procedurally propagates taints at runtime from a local sink to a local source, ensuring the data flow of a tainted value across multiple methods correctly propagates the taints.

TaintLib. In order to improve the flexibility of our solution by not restricting our system to a specific implementation for storing, updating, and checking taints, we make use of *ARTist*'s modular design and deploy the taint tracking logic in form of a new *companion library* called *TaintLib* that is merged by *deployment app* into the app code at compile time. *TaintLib*, in turn, relies on a policy file that defines the global and local sources/sinks as well as the sources' taints tags. *TaintLib* provides source type-specific *taint-set* methods, calls to which are inlined at all sources, and sink-type-specific *taint-get* methods, calls to which are inlined at all sinks. By injecting *TaintLib* method calls instead of concrete taint tracking logic, we decouple the instrumentation from the taint management code. For global sources, *taint-get* retrieves and sets the taint tag according to the policy and *taint-set* at global sinks checks[4] the taint tag. In contrast, for local sinks *taint-set* propagates the tag together with the tainted value to the next local source, where it is retrieved with *taint-get*. By instrumenting all methods alike, an implicit contract between all methods is established and fulfilled, i.e., every time a *taint-get* tries to obtain the taint value of a method parameter on the callee side, we know the corresponding *taint-set* has been executed in the calling

4. While a naïve check halts the program when tainted data is about to leak, invoking a sanitizer as suggested by [17] can easily be implemented in *ARTist*.

method to provide the taint data. In case the slice contains multiple *sources*, the output of their corresponding *taint-gets* is combined by injecting a call to a merger method that combines the taint tags.

To continue our running example, the right hand side of Figure 5 presents the IR of the code snippet with *taint-set* and *taint-get* calls inlined. For instance, the `setArgTaint` call for LSI1 in basic block 2 of `getID` (HInstructions 52) precedes the local sink in HInstructions 31 that invokes the `prefixID` function. The `setArgTaint` instruction transfers the taint of `id` inter-procedurally to the `getArgTaint` instruction in HInstruction 77 of basic block 0 of `prefixID` (dashed line ❷), from where it is intra-procedurally propagated using the backwards slicing information (solid line ❸). Similarly, the taint is propagated back from `prefixID` to `getID` through the return statement and variable assignment (dashed line ❹).

Inter-procedural taint tag propagation channel. In the case of parameters (LSO1 and LSI1) and method returns (LSO2 and LSI2), there are always pairs of *taint-sets* and *taint-gets* present at runtime, due to the fact that for each callee method, there is a caller method that also has been instrumented. Combining this with the observation that a caller-callee method pair is always executed in the same thread, the taint propagation can be realized using thread local storage for a taint stack. At the caller side, the taint information is pushed onto a per-thread stack and at the callee side it is popped again, vaguely resembling the x86 calling convention for passing arguments to methods. Keeping in mind that almost every injected *TaintLib* method call accesses the taint information, replacing more straightforward approaches for taint storage (like a single HashMap) with cheaper stack operations also benefits the overall performance of our taint tracking solution.

In the case of field operations (LSO3 and LSI3), we can neither assume them to appear in pairs nor to be executed on the same thread and therefore employ a thread-safe mapping in the form of a ConcurrentHashMap. This, however, raises the challenge of providing easily computable, stable and unique keys. If we consider our taint tags not to store the taint value of a certain value, but of a certain location, we can compute stable identifiers for fields and use them as keys. For static class fields, identifying the specific class and field is sufficient and can be precomputed during compilation. The current implementation injects the computed key as a constant into the HGraph and provides it as an argument to a field *taint-set* or *taint-get*. For object fields, we do not only need to identify classes but concrete objects, which requires runtime information. In this case, we only inject the field identifier as a constant and provide it together with the field's concrete object to a carefully crafted *TaintLib* function. The returned key is robust to object aliasing such that we do not lose track of objects in, e.g., collections. Afterwards, we can use this key in a *taint-set* or *taint-get* for the object field.

It is important to note that our approach to taint tracking depends not only on the entity for which we store taints (i.e., variable locations instead of values), but also on the type of data to which we assign taint values. In our model, we track taints only for primitive types and the taint tag of objects is transitively given by their field's tags. In case of non-primitive fields, the rule applies recursively because eventually all objects can be decomposed to primitives. This design decision is motivated by the fact that tracking all *taint-set* and *taint-get* operations on fields and on all method invocations is more fine-grained than storing taint information at the object level.

5.3. Further Use Cases

Dynamic analysis. Compiler-based solutions are inherently well-suited for white box approaches that require an understanding of the application's internals. One example is the taint tracking *Module* described above that re-instantiates TaintDroid-inspired intra-app taint tracking. Other examples are existing works on commodity systems [18], [19] that already utilize compilers for information flow control, which can now be realized on Android as well.

Container solutions. Modifications of the Android runtime environment have been used in the past (for instance *Divide*[5], now part of Google Android for Work) to establish container solutions that, e.g., encrypt file system I/O of apps or restrict inter-application communication. Using a compiler-based approach such as *ARTist*, similar container solutions can be established by replacing the corresponding method invocations (e.g., calls to Java's I/O classes) with calls to injected security-enhanced versions of the same.

Code replacement and compile-time patching. Google has recently started separating security-critical libraries, such as the notorious WebKit, from application packages into standalone apps that are called by apps on-demand. This allows Google to maintain those libraries on an ecosystem-wide scale and roll out security patches more effectively. Since *ARTist* is not only able to inject but also to replace or remove code from an app's code base, *ARTist* can also be used to apply *compile-time patches* by replacing vulnerable libraries within apps with fixed versions. In an extreme case, this mechanism could allow for removing entire libraries by mocking all their method invocations (e.g., removing ads), or moving code partitions behind a strong security boundary, such as a dedicated process, and reconnect the code through inter-process communications (e.g., as done in the AdSplit [20] or AdDroid [21] solutions).

Beyond security: profiling and debugging. Besides its application in the security domain, using *ARTist* to inject tracing, debugging or profiling code allows to gain additional insights into third-party applications. A basic example is the method call-tracing we employ in our robustness evaluation in Section 6.1.1.

5. http://www.divide.com

6. Discussion

This section evaluates *ARTist* and its *Modules* in terms of robustness, performance, inherent and implementation-specific limitations, and discusses ideas for future work.

6.1. *ARTist*

We first evaluate the general *ART Instrumentation and Security Toolkit* and discuss its general limitations.

6.1.1. Robustness. In order to prove its applicability, we conducted an evaluation on top apps from the Google Play Store.

Evaluation Infrastructure In order to scale our evaluation to thousands of apps, we created a dynamic on-device app testing infrastructure called *monkey-troop*. It allows us to automatically test apps with *ARTist Modules* on an arbitrary number on connected devices in parallel, thereby heavily reducing the time required for large-scale evaluations. Its pipeline works as follows:

1. Setup Before we start the actual testing, our special *Module* within *ARTist* is installed as an application on all test devices.

2. Filter In order to be a valid target for our evaluation, some applications are filtered out early. Some apps, even though they are in our list, are not available in the Google Play Store anymore and thus we cannot download them for our tests. Also, those apps that can be downloaded and installed successfully are tested by the *monkey* UI exerciser tool *before* we apply our instrumentation, thus ruling out apps that are already crashing without any modification from our side. Dropping them them off the list avoids impairment of our evaluation results.

3. Test We apply the instrumentation of the installed *ARTist Module* by recompiling the target applications. Afterwards, we test them again using the *monkey* UI test automation tool and monitor the execution. Using the same *monkey* seed during filtering and testing ensured reproducibility. If the application does not crash, we count a success.

4. Collect We collect and store all data generated by the testing scripts, as well as `logcat` dumps from the device to allow for further analyses after the evaluation.

***ARTist* Robustness Evaluation** We used *monkey-troop* to test the robustness of *ARTist*-based instrumentation on 3060 top apps from the Google Play Store. As automated app testing with high coverage is still an open problem, the *ARTist Module* created for the evaluation ensures the execution of our custom code by injecting it into each single onCreate method in any developer-written class. Thus, starting an application through its launcher activity (as done by *monkey*) always triggers our injected code.

The injected tracking code implements method-call tracing by utilizing stack inspection to print the current method's name to the log. Using this setup, we can evaluate the

Category	Tested	Success	Percentage
Books And Reference	44	39	88.64%
Business	37	33	89.19%
Comics	44	41	93.18%
Communication	45	38	84.44%
Education	38	35	92.11%
Entertainment	34	32	94.12%
Family	30	28	93.33%
Family?age=age Range1	41	40	97.56%
Family?age=age Range2	40	39	97.5%
Family?age=age Range3	29	27	93.1%
Family Action	33	31	93.94%
Family Braingames	43	43	100.0%
Family Create	47	44	93.62%
Family Education	46	45	97.83%
Family Musicvideo	52	49	94.23%
Family Pretend	40	38	95.0%
Finance	37	30	81.08%
Game Action	34	32	94.12%
Game Adventure	25	23	92.0%
Game Arcade	31	29	93.55%
Game Board	48	47	97.92%
Game Card	38	33	86.84%
Game Casino	25	22	88.0%
Game Casual	25	25	100.0%
Game Educational	36	34	94.44%
Game Music	41	37	90.24%
Game Puzzle	31	30	96.77%
Game Racing	32	30	93.75%
Game Role Playing	20	19	95.0%
Game Simulation	21	21	100.0%
Game Sports	40	40	100.0%
Game Strategy	25	23	92.0%
Game Trivia	42	38	90.48%
Game Word	50	46	92.0%
Health And Fitness	36	31	86.11%
Libraries And Demo	20	18	90.0%
Lifestyle	42	41	97.62%
Media And Video	36	33	91.67%
Medical	42	41	97.62%
Music And Audio	40	32	80.0%
News And Magazines	43	38	88.37%
Personalization	47	44	93.62%
Photography	44	38	86.36%
Productivity	34	31	91.18%
Shopping	39	34	87.18%
Social	33	25	75.76%
Sports	32	30	93.75%
Tools	48	44	91.67%
Transportation	49	46	93.88%
Travel And Local	41	36	87.8%
Weather	41	38	92.68%
51 Categories	1911	1761	92.15%

TABLE 2: *ARTist* robustness evaluation results for the Google Play categories. Note that the *Family* categories contain apps from other categories as well. Filtered apps are omitted.

robustness of our instrumentation on real-world applications. The significant performance overhead incurred by the expensive stack inspection routine is only of secondary interest since we solely focus on robustness testing here. Table 2 shows the results of our evaluation. Out of 1911 tested apps, 1761 (92.15%) were successfully instrumented and tested, clearly showing the robustness of *ARTist*'s instrumentation capabilities.

We conducted a manual investigation on the remaining applications in order to find the root cause of their failure during the evaluation. We detected that a lot of errors are seemingly not a result of our instrumentation but false positives. Even though the same touch events are delivered to the app before and after instrumentation, there are still unexpected errors that are seemingly unrelated to our modifications. For example, we encountered apps failing with a `SecurityException` because of missing permissions or `IllegalStateException` because the application is already initialized. Both should have been triggered during the filtering in the first place. Also, we can rule out deviating behavior because of statefulness since we make a clean install of the app before instrumenting it, thus removing any state potentially created during the first *monkey* test. Our findings therefore indicate that the filtering failed in those cases.

We conclude that our large-scale evaluation approach meets its purpose by providing an estimation on the robustness of instrumenting real-life applications. However, the degree of automation and the lack of a more sophisticated UI testing tool for Android apps introduces a certain imprecision that needs to be taken into account when working with those results.

6.1.2. Performance. The actual runtime overhead induced for apps instrumented with *ARTist* largely depends on the concrete *Module*, e.g., taint tracking requires more injected code than permission enforcement. Therefore, we provide concrete measurements for instrumented applications in the context of our implemented use case *Modules* in Sections 6.2.1 and 6.3.1.

6.1.3. Conceptual Limitations.

Native code support. *Optimizing* operates by design on dex input only. Bundled native libraries (i.e., C/C++) that are connected via JNI are never transformed into *Optimizing*'s IR and therefore, neither instrumented nor inspected by our prototype. Native code components are a limitation of the attacker model not only our concept but are indeed an open challenge for most of the solutions by the Android security research community, e.g., code analysis as well as IRM solutions in particular.

Potential fallback to dex. The oat files produced by *dex2oat* still contain the original dex byte code of the app to allow fallback to interpretation mode. Naturally, fallback to interpretation would render our instrumentation of the compiled dex byte code futile. This fallback is currently limited to app debugging. However, no guarantees exist that such a fallback cannot be triggered maliciously. Similarly, dynamically loaded dex code [22], [23] (e.g., via the DexClassLoader) is by default compiled to native bytecode, but no guarantee can be given that dynamically loaded code cannot fall back to interpretation.

6.1.4. Implementation Limitations.

Permanence of instrumentation. Instrumentation of an app's oat file might be reverted through an application update or a firmware update where apps are re-compiled. Thus, there exists a window of opportunity for an attacker to start an uninstrumented app after a system or app update. Apps, however, cannot be started programmatically after install/update until the user has started the app manually and both scenarios can be detected by *deployment app* via system notifications (i.e., broadcasts). Assuming that the system notifies the *deployment app* fast enough in order to re-instrument the updated app before the user manually starts the app, the window of opportunity in which an uninstrumented app is started can be closed.

Deployment strategy. In order to create a pure application layer solution, our prototype currently relies on the naïve approach of requesting elevated privileges to replace the installed app oat file with the instrumented version. We can eliminate this requirement by integrating *ARTist* with an application-layer only sandboxing solution that provides file system virtualization, such as *Boxify* [24] or *NJAS* [25], or by resetting the execution environment and replacing loaded libraries using reference hijacking [14]. Both approaches enable the manipulation of file paths from the original to the instrumented oat file at application startup time.

6.2. Dynamic Permission Module

We briefly evaluate the performance impact of our dynamic permission module and discuss limitations.

6.2.1. Evaluation. The additional security checks inlined by our *Module* are only inserted before permission-protected SDK method calls. Thus, we cannot rely on benchmark apps, because they rarely trigger the added functionality. Therefore, we evaluate the performance impact of our permission-checking code using custom microbenchmarks. Table 3 depicts the results of our measurements for calls that are protected by 3 distinct permissions. The overhead encountered in the microbenchmarks ranges between 1.18% and 30.65%, showing the feasibility of our prototype if we consider that only permission-protected method calls suffer from this overhead.

6.2.2. Limitations.

Restriction to synchronous calls. In order to demonstrate the straightforward implementation of an *ARTist Module*, we opted for a simple instrumentation strategy that only covers synchronous permission-protected method calls. As

Microbenchmarks				
Tested Method	Permission	Baseline	Instrumented	Penalty
WifiManager.getConfiguredNetworks()	ACCESS_WIFI_STATE	0.681 ms	0.742 ms	8.89%
WifiManager.isWifiEnabled()	ACCESS_WIFI_STATE	0.071 ms	0.072 ms	1.18%
WifiManager.getScanResults()	ACCESS_COARSE_LOCATION	0.452 ms	0.591 ms	30.65%
BluetoothAdapter.startDiscovery()	BLUETOOTH_ADMIN	0.910 ms	0.940 ms	3.32%

TABLE 3: Microbenchmarks averaged over 60.000 runs. The *baseline* benchmarks measure the pure execution time of the permission-protected call while the *instrumented* benchmarks measure the protected call and the additional permission check.

a result, the current prototype does not support callbacks or asynchronicity and its implementation should therefore be considered a proof-of-concept only.

Best effort permission map. In order to direct *ARTist* to the instrumentation targets, i.e., the application's permission-protected method calls, we utilize the method call to permission mapping provided on the website of the state-o-the-art tool *Axplorer*[26][6]. Unfortunately, the latest available map is generated for SDK 24, which means that our permission enforcement *Module* only supports Marshmallow. In addition, as stated on the website, the map is incomplete since many permission checks moved into the `AppOpsManger`, which requires different analysis techniques and is therefore not fully supported yet. Our *Module* inherits this limitation from the used permission map.

6.3. Taint Tracking Module

We evaluate our taint tracking *Module* in terms of feasibility, performance, and limitations of its current prototypical implementation.

6.3.1. Evaluation.

Runtime overhead. We leverage an Android microbenchmark application to evaluate the performance of our prototype. Since our taint-instrumentation only affects the performance of Java code, we specifically chose the *Passmark* benchmark, which does not contain native libraries and implements all benchmarks in Java. Table 4 compares the results of the baseline benchmark with a non-instrumented *Passmark* app to those of an instrumented and taint-aware version. The results show an overhead ranging between 7.74% and 30.73%, which is within an acceptable range for a taint tracking approach that is not fully tuned for performance. This result is also roughly comparable to microbenchmark results of TaintDroid's [1] interpreter-based approach. However, as stated in [27], microbenchmarks are not very representative in user-driven scenarios such as Android apps. Hence, we take this result with a grain of salt.

Overall performance can be enhanced by introducing custom optimizations specifically tailored towards improving taint tracking code. One approach would be to eliminate *taint-sets* and *taint-gets* that are based on stack operations and cancel each other out, e.g., alternating *pushs* and *pops* of the same tag as seen for methods that return the return value of another method call. Moreover, the analysis phase allows to abstain from instrumenting apps that do not contain any global taint sinks in order not to impact performance at all in this case.

6. http://axplorer.org

Passmark			
Test	Baseline	Taint-Aware	Penalty
CPU	32521	22526	30.73%
Disk	24893	20777	16.53%
Memory	3627	3346	7.74%

TABLE 4: Passmark results averaged over 5 runs, higher is better.

Functional Evaluation. We conducted this case study to research whether intra-application taint tracking can be achieved with a compiler-based instrumentation framework such as *ARTist*; thus our functional evaluation focuses on detecting different kinds of data leaks in apps. However, to the best of our knowledge, there is no standardized test suite specifically tailored towards evaluating dynamic taint tracking systems for Android apps, and testing real applications is not feasible because they lack the required ground truth. In order to overcome this unsatisfactory situation, we decided to exploit an open-source suite called *DroidBench* [28], [29] that was initially created to benchmark static taint tracking systems. Even though this does not immediately apply to a dynamic system such as ours, we can still leverage the fact that it provides us with an assortment of applications with different but well-defined leakage behavior. Table 5 summarizes our *Module's* results for those tests and categories within scope. Tests for implicit flows, inter-component communication, and reflection are omitted because they currently exceed the scope of our proof-of-concept taint tracking. As we are *abusing* the benchmark suite, we need to be careful which conclusions we draw from the test results. The first insight we gain, however, is that our case study succeeded in showing that intra-app taint tracking can be implemented as a pure application-layer solution using compiler-driven instrumentation. The second insight we derive is that, as indicated by lower results such as those for the *Android Specifics* category, our proof-of-concept does not yet catch up with previous works such as TaintDroid. Nonetheless, our work not only shows the feasibility of the approach but also lays the foundation for creating a full-fledged taint tracking system for Android versions above Marshmallow that utilizes compiler-based instrumentation and does not require operating system modification.

DroidBench		
Category	Successful Tests	Ratio
Callbacks	14/15	93%
Lifecycle	13/14	92.9%
General Java	14/20	70%
Aliasing	1/1	100%
Android Specifics	5/9	55.6%
Field & Object Sensitivity	7/7	100%
Overall	54/66	81.8%

TABLE 5: Results for the DroidBench taint tracking evaluation. Broken tests and categories not applicable to our system are omitted.

6.3.2. Limitations.

No tracking of implicit flows. Like TaintDroid [1], our system currently does not track implicit flows (i.e., data leakage using control flow dependencies) and malevolent apps could exfiltrate data in a way that is unnoticeable by our prototype. As the TaintDroid authors discuss, mitigating leakage through control flows would require static analysis and access to the app's source code—both of which TaintDroid could not provide. *ARTist*, however, is already provided with the full app code and it would be highly interesting future work to investigate to which extent the structural program information of the IR and analytical features of the compiler backend (i.e., *Optimizing*) can help to remedy the limitations of customary taint tracking solutions on Android.

Taint tracking boundaries. The compiler is restricted to the app's codebase, which introduces imprecision when leaking information through SDK methods, where a *taint-set* at the caller side (developer code) but not the *taint-get* at the callee side (SDK) can be inlined. In particular, and in contrast to object types, storing primitives or strings in collections or sharing them across threads are corner cases where the taints will not be propagated appropriately. This shortcoming can be solved by using pre-computed control-flow models for framework methods [30] to generate corresponding *taint-set* and *taint-get* pairs that model the transition of data through the framework. A preferable technical solution in the future, which removes the potential over-approximations of SDK internal states in control-flow models [30] and which could be of interest beyond taint tracking, is the instrumentation of the *core image*. The core image is a pre-compiled oat file of the framework classes that is pre-loaded into every application process via Zygote. Since the core image is created with *dex2oat* during the device startup once after each system update, it can be instrumented using a *sec-compiler* as in *ARTist*. However, in either case and as in the original work [1], data that already left the phone (e.g., through a network socket) cannot be tracked.

Inter-application communication. Our prototype is currently limited to *intra*-application tracking and lacks support for *inter*-application tracking, for instance, through the file system or Binder IPC. This opens the possibility of confused deputy [31], [32] or collusion attacks [33], [34] to exfiltrate data. Assuming that all installed apps are instrumented, a fix to this problem would be the instrumentation of the I/O method calls in order to write out taints together with the data (e.g., into a file or Binder Parcel) and restore taints at the receiver side. When abandoning the requirement for a pure application-layer solution, our system could also be complemented with the original TaintDroid file system and IPC infrastructure, which is unaffected by the loss of DVM, in order to track taints across applications.

7. Conclusion

In this paper, we presented *ARTist*, a *dex2oat* compiler-based instrumentation solution for Android applications that operates at the application layer only. In order to be able to design and implement *ARTist*, we first and foremost had to thoroughly study the yet uncharted internals of the new compiler suite and in particular of its *Optimizing* backend. A deeper understanding of this compiler suite and the new ART runtime is of interest for the security community insofar as ART and *dex2oat* replaced the interpreter-based runtime (DVM) of Android versions prior to Lollipop (i.e., version 5) and hence also voided applicability of any security solution that relies on interpreter instrumentation (e.g., TaintDroid [1] and its derivatives [2], [9]). We studied the feasibility of our approach by implementing two distinct use cases. Furthermore, our case study highlights the capability of a compiler-based instrumentation framework to re-instantiate basic taint tracking for Android apps at the application layer. In general, our results provide compelling arguments such as higher robustness and better integration for preferring compiler-based instrumentation over alternative bytecode or binary rewriting approaches. We open sourced the results of this paper to allow for independent research on the topic.

Acknowledgment

We would like to thank the reviewers for their helpful comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

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 *USENIX OSDI'10*.

[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 *ACM CCS'11*.

[3] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: supporting operation modes on smartphones," in *ACM SACMAT'12*.

[4] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati, "Spandex: Secure password tracking for android," in *USENIX SEC'14*.

[5] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," in *IEEE CS MoST'12*.

[6] H. Hao, V. Singh, and W. Du, "On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android," in *ACM ASIACCS'13*.

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

[8] 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.

[9] "Google code: Droidbox," https://code.google.com/p/droidbox/.

[10] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *ACM CCS SPSM'12*.

[11] B. Davis and H. Chen, "Retroskeleton: Retrofitting android apps," in *ACM MobiSys'13*.

[12] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard - enforcing user requirements on Android apps," in *TACAS'13*.

[13] J. Schütte, D. Titze, and J. De Fuentes, "AppCaulk: Data leak prevention by injecting targeted taint tracking into android apps," in *TrustCom14*.

[14] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang, "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices," in *ICSE'16*.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] R. Xu, H. Saïdi, and R. Anderson, "Aurasium – Practical Policy Enforcement for Android Applications," in *USENIX SEC'12*.

[17] B. Livshits and S. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," in *ACM SIGPLAN'13*.

[18] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *ACM CCS'08*.

[19] F. Araujo, W. Kevin *et al.*, "Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception," in *USENIX SEC'15*.

[20] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *USENIX SEC'12*.

[21] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *ACM ASIACCS'12*.

[22] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *NDSS'14*.

[23] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *ACM WISEC'12*.

[24] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *USENIX SEC'15*.

[25] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "NJAS: Sandboxing unmodified applications in non-rooted devices running stock android," in *ACM CCS SPSM'15*.

[26] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *USENIX SEC'16*.

[27] B. Livshits, "Dynamic taint tracking in managed runtimes," Microsoft Research, Tech. Rep.

[28] 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 '14*.

[29] Online: https://github.com/secure-software-engineering/DroidBench.

[30] 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 *NDSS'15*.

[31] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX SEC'11*.

[32] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *ISC'10*.

[33] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *NDSS'11*.

[34] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," in *NDSS'12*.