Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART

Jakob Bleier
TU Wien, Vienna, Austria
jakob.bleier@seclab.wien

Martina Lindorfer
TU Wien, Vienna, Austria
martina.lindorfer@seclab.wien

ABSTRACT

The Android operating system has evolved significantly since its initial release in 2008. Most importantly, in a continuing effort to increase the run-time performance of mobile applications (apps) and to reduce resource requirements, the way code is executed has transformed from being bytecode-based to a binary-based approach: Apps are still mainly distributed as Dalvik bytecode, but the Android Runtime (ART) uses an optimizing compiler to create binary code ahead-of-time (AOT), just-in-time (JIT), or as a combination of both.

These changes in the build pipeline, including increasing obfuscation and optimization of the Dalvik bytecode, invalidate assumptions of bytecode-based static code analysis approaches through identifier renaming and code shrinking. Furthermore, customized apps can be distributed pre-compiled with devices' firmware, sidestepping the bytecode altogether. Finally, Android apps have always relied on native binary code libraries for performance-critical tasks.

We propose to narrow the gap between bytecode and binary code by leveraging the ART compiler's capability to create well-formed ELF binaries, called OATs, as the basis for further static code analysis. To this end, we created a pipeline to automatically and efficiently compile APKs to OATs into a benchmark dataset of 1,339 apps. We then evaluate five popular disassemblers based on how well they can analyze these OATs based on how well they can detect function boundaries. Our results, in particular, compared to the success rate of two bytecode-based analyzers, demonstrate that our OAT-based approach can help to bring a wider set of code analysis tools and techniques to the area of Android app analysis.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**.

KEYWORDS

Static analysis; Android Runtime; Bytecode vs Binary; Disassembly

ACM Reference Format:
Jakob Bleier and Martina Lindorfer. 2023. Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART. In *European Workshop on System Security (EuroSec '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3578357.3591219

EuroSec '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0085-9/23/05…$15.00
https://doi.org/10.1145/3578357.3591219

1 INTRODUCTION

Android has become the most widely used operating system [51] and makes it easy to develop and publish applications (apps) through its standardized APIs and distribution model—even for publishers with no development experience thanks to app generators [42]. As of Fall 2022, the Google Play Store, the largest repository of Android apps, hosted 2.6M apps [53], with 97,000 new apps being published each month [52]. However, it is only one of multiple distribution channels: in addition to a plethora of alternative markets [34, 58], original equipment manufacturers (OEMs) and network operators customize and bundle apps with their devices' firmware [12].

Because of its popularity and integration in virtually every context of modern life, Android and its apps are also an attractive target for malicious actors along the whole supply chain. Mobile malware typically injects malicious code into popular apps, a process called *repackaging*, leading to a long line of research on app clone detection [37]. However, this is by far not the only open issue from a security and privacy perspective: Other ongoing research ranges from the detection of vulnerabilities (potentially caused by missing updates in included third-party code) [9, 32], over the characterization of privacy-invasive behavior through third-party libraries [64], to general attribution issues in the Android ecosystem [28].

All of these use cases rely on, or could benefit from, reliable and complete static code analysis. Unfortunately, due to the changes in the build and execution environment that we document as part of this paper, existing approaches are either outdated or rely on unrealistic assumptions: (1) approaches depending on the accuracy of method or class signatures are defeated by common bytecode obfuscation; (2) clone detection approaches often include dead code eliminated by code shrinking; (3) apps might no longer be distributed as bytecode but pre-installed as optimized binaries; (4) apps generally can include native code that typically falls under the limitations of any bytecode-based approach.

In this paper, we investigate whether we can leverage the changes in the Android Runtime (ART), which compiles Dalvik bytecode into binary (i.e., native machine) code, to our advantage. We demonstrate that it is feasible to transform apps into well-formed Executable and Linkable Format (ELF) binaries (distributed as OATs on Android) and evaluate five popular commercial as well as open-source disassemblers (IDA Pro [31], Ghidra [40], Binary Ninja [57], radare2 [44], angr [47]) on how they handle them.

In summary, we make the following contributions:

- We compile a subset of 1,339 open-source apps available F-Droid as APKs and then to OATs as our benchmark dataset.
- We evaluate five disassemblers on how accurately they identify function boundaries (i.e., offsets and sizes)—the basis for any further static code analysis.

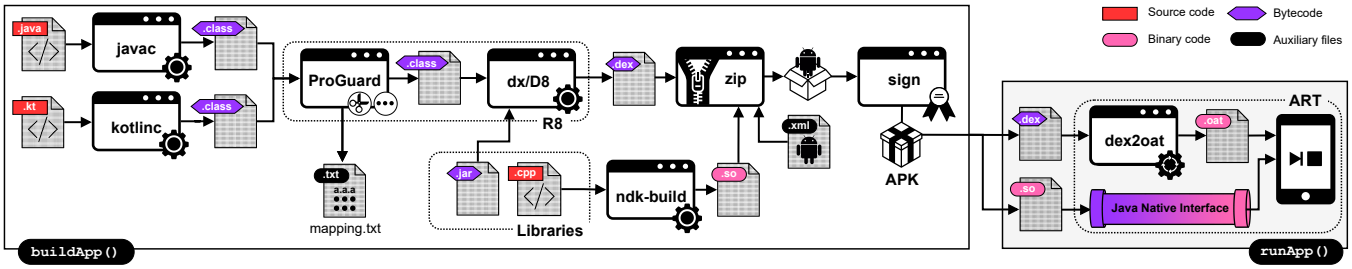

Figure 1: ***Lifetime of an Android app from source code to execution:*** **Java and Kotlin source code and libraries are compiled into optimized and obfuscated Dalvik bytecode. Native libraries written in C/C++ are compiled to binary ELF shared objects. This code is then bundled with resources, such as the manifest file and images, into an archive and signed. During run time, native libraries are directly loaded by ART, while Dalvik bytecode is compiled into OATs (i.e., ELF binaries with additional metadata).**

- We compare the overall success rate of these binary disassemblers in handling apps compared to traditional bytecode processing with Soot [48] (100% vs. 94%) and SootUp [49].

Our results show that OATs are a promising approach and can indeed narrow the gap between Dalvik (bytecode) and native (binary) code. In future work, we will evaluate how well we can repurpose the large existing toolbox of binary analysis approaches, e.g., for code clone detection, for Android app analysis.

2 ANATOMY OF AN ANDROID APP

We start by providing an overview of how Android apps are built and executed (summarized in Figure 1), and how these processes have evolved over the years since Android's first release in 2008.

App format and structure. Apps are distributed in the Android Package (APK) file format. An APK is a compressed ("zipped") archive containing metadata about the app, its code, and resources, such as images or localization files. The required `AndroidManifest.xml` provides metadata such as the package name, which uniquely identifies the app within the Google Play Store (and once installed on the device), any requested permissions, and declared interfaces. A signature is required to publish and install an APK [23], but the certificates are typically self-signed.

Bytecode compilers. Originally, apps were written in Java and compiled with the javac compiler to Java bytecode (.class) and then with dx to Dalvik bytecode (.dex). Pre-compiled third-party libraries, such as advertisement SDKs, can be included and "dexed" as well, for example, in .jar (= collection of .class files) format. Android also supports bundling libraries and their resources into an Android Archive (.aar), which is packaged as an "app within the app." The compiled Dalvik bytecode is then stored in one or more Dalvik Executable (DEX) files, packed into an APK, and distributed to run on Android devices. Tools written for Java can be easily integrated into the pipeline at this stage, such as the optimizer and obfuscator ProGuard [27], which has been a part of Android's build tools since Android 2.3 (Gingerbread, released in 2010) [14].

Since its first release, Android has transformed significantly: Supported since 2017, Kotlin became the preferred programming language in 2019 [19, 30]. The compilation of source code to Dalvik bytecode has changed as well: Combining the Java-based ProGuard with the DEX compiler dx had the drawback that the obfuscation-unaware compiler had a hard time optimizing the intermediate Java bytecode. As a result, Facebook developed their own DEX bytecode optimizer Redex [11]. Google experimented with the Jack (Java Android Compiler Kit) and Jill (Jack Intermediate Library Linker) toolchain, which combined Java to DEX compilation with shrinking and obfuscation [15]. Both were later deprecated in favor of D8/R8: In 2019 D8 became the default "dexer" instead of dx, while R8 fully replaced dx. R8 included several features ProGuard performed separately from compilation [25, 26], such as obfuscation (renaming variables, methods, and class identifiers), optimization of code-related features, and code shrinking (removing unused code, also known as tree shaking) [16, 18].[1] Even though not enabled by default during development, these steps are recommended for the final version of an app [18]. When R8 obfuscation is enabled, it generates a `mapping.txt` file to map original to obfuscated method names. This file is not included in the final APK, thus not available to security analysts, but only intended for the developer, e.g., to recover the original names in stack traces for debugging.

Android Runtime (ART). While an apps' code is distributed as Dalvik bytecode, the runtime compiles it to binary machine code to increase performance and reduce system resources. In 2014, with the release of Android 5.0 (Lollipop), Google fully replaced the Dalvik VM with ART using ahead-of-time (AOT) compilation, i.e., compiling the whole app during installation [21]. Because the necessary re-compilation during app and system updates turned out to be too resource-intensive, Android 7.0 (Nougat, released in 2016) added just-in-time (JIT) compilation. In the current Android 13.0, ART uses both AOT and JIT compilation.

The ART compiler dex2oat compiles Dalvik bytecode to native binaries, so-called *Of Ahead Time* (OAT) files. An OAT file (.oat) is essentially an ELF shared object containing additional sections with OAT metadata [55]. In addition to the hybrid AOT+JIT model, dex2oat can compile apps in their entirety to generate OAT files with all methods compiled to binary code. Note that, dex2oat is a new compiler implementation not based on the commonly used Clang or gcc. Furthermore, e.g., for system apps or framework libraries provided by device manufacturers or other parties along the supply chain, OAT files ship pre-compiled and bundled with the firmware of a device, i.e., their Dalvik bytecode is unavailable for analysis.

[1] The configuration names are the same for backwards-compatibility, but R8's actual implementation is different from the standalone ProGuard [26].

Android Native Development Kit (NDK). To enable the inclusion of highly optimized binary files, developers can bundle pre-compiled native[2] code in their apps through the Native Development Kit (NDK) [22]. This native code can be called through the Java Native Interface (JNI) from Java or Kotlin code. Native libraries are distributed as ELF shared object (.so) files, and by default stripped of their symbol table and debugging information during the build process with the clang-based `ndk-build` [18]. Similar to `mapping.txt` generated by R8 it generates `native-debug-symbols.zip`, which also stays with the developer for recovering names. Unlike R8-generated code though, native libraries are loaded directly by ART at run time without further compilation and optimization.

3 OATS: CHALLENGES & OPPORTUNITIES

As detailed in Section 2, app code is typically distributed as Dalvik bytecode, but it is executed as binary code. Note that, modern Android systems almost exclusively run on ARM (~98% [50, 56]), while the x86/64 architecture is mainly used by emulators. In general, analyzing binaries is challenging for multiple reasons, which have been well documented in the literature [1, 47]: the detection (i.e., recovery) of function boundaries, and dealing with obfuscation.

Function recovery. The correct identification of functions in binary code is a topic of still ongoing research, and its accuracy affects any code analysis approaches that build on the identified functions. Because native code can be stripped of debug and symbol information, the starting addresses of all functions cannot always be determined with certainty. Quarkslab [7] evaluated disassemblers and binary exporters with a focus on performance comparison of IDA Pro and Ghidra. The latter was significantly slower for large binaries, but in addition to disassembly, performs decompilation, which IDA Pro does not do automatically. Jiang et al. [36] surveyed eight popular state-of-the-art tools that can parse and extract instructions and functions from ARM binary files: three commercial (Binary Ninja, Hopper, IDA Pro) and five non-commercial ones (angr, BAP, Ghidra, objdump, radare2). They observed room for improvement when it comes to function boundary detection, especially since the approach that works well for x86/x64 binaries, Nucleus [2], does not work on the ARM instruction set. Jiang et al. [35] recently extended their study to evaluate the disassemblers' performance in locating instruction boundaries, function boundaries, and function signatures of ARM binaries. However, their work focused on the `gcc` and `clang` compilers, i.e., their results only give an indication about how well disassemblers handle the native libraries built with them. In contrast, we dig deeper into the disassemblers' performance on binaries generated by dex2oat, which assists us with metadata about correct function boundaries.

Optimizing (away) obfuscations. Even when functions have been identified correctly, optimizations and obfuscations still pose significant challenges for any analysis approach. Ren et al. [46] surveyed 47 binary similarity approaches and found that all of them only assumed default compiler optimizations. Ren et al. further studied the impact of non-default optimizations and proposed BinTuner, which tries to find the compiler optimization settings that maximize binary code differences based on iterative compilation.

[2]Native and binary code are terms that can be used interchangeably, but we call binary libraries pre-compiled in the APK *native* and DEX code compiled with ART *binary*.

Finally, a common approach to analyze binary code is to first normalize it by lifting it into an intermediate representation (IR), i.e., a higher-level language, and then re-compile it with optimizations [1, 8, 13]. If successful, the re-compilation with aggressive optimizations can normalize the code and thus remove obfuscations. This optimized re-compilation is also an intuition behind our approach, although we essentially transform the code to a lower-level language: we aim to use the ART compiler to mitigate optimizations and obfuscations – introduced at a higher level during the bytecode creation and "dexing" – in the binary code used at later stages.

4 RELATED WORK

Android app analysis toolchains. *droid [45] surveyed 300 Android analysis papers published between 2010—2016 and found both a lack of maintenance and issues with reproducibility—an issue we encountered as well. Pauck et al. [43] focused on static taint tracking and similarly observed a lack of reproducible comparisons as well as datasets that represent real-world apps. Mauthe et al. [39] surveyed Android decompilers, i.e., tools to lift Dalvik bytecode to Java source code. While they observed obfuscation as a minor factor in this use case, they encountered tool failures due to technical limitations. We faced similar technical limitations with Dalvik-based tools to process apps due to underlying issues in Soot, the most commonly used bytecode processing framework.

ART-assisted security. The replacement of the Dalvik VM with ART not only broke existing approaches but opened up new possibilities for security mechanisms. CompARTist [33] instruments ART to sandbox third-party libraries from the main app. TaintART [54], TaintMan [63], and NDroid [60] use ART to implement dynamic taint tracking. ARTist [4] is a more general app instrumentation framework with taint tracking as one of its use cases. Malton [61] instruments ART to monitor the execution of malware. Other approaches, such as Tiro [59] and DexLego [41] instrument ART to defeat malware using run-time packing and extract Dalvik bytecode. In contrast, our approach does not aim at reconstructing bytecode, but instead directly processes the well-formed binary code generated by ART. Most closely related to our work, Deoptfuscator [62] implements an analyzer on top of ART to find potentially obfuscated methods and then instructs ReDex to optimize those away. However, in addition to requiring repackaging (and re-signing) the app, which is invasive and can be detected by app integrity checks used for tamper-proofing an app [29, 65], this does not allow for the analysis of pre-installed OAT-only apps.

5 OUR APPROACH

To evaluate the practicality of OAT-based app analysis we measure how accurate binary disassemblers are at identifying correct functions in OAT files without guidance from metadata obtained from the Dalvik bytecode. While `oatdump`, which is included in ART, parses OATs and can link them to the Dalvik bytecode they were compiled from, it is meant for humans and the output is not (yet) stable across versions. However, we can use it as a ground truth to compare the function boundaries provided by various disassemblers. We start our evaluation with APKs, which we automatically compile to OAT files for further analysis.

5.1 Dataset

We use a subset of apps from the open-source F-Droid market [10] and compile the APKs from their source. In addition to building the apps, we extended the automated build server [10] to collect detailed build configurations, e.g., whether obfuscation, optimization, or code shrinking is enabled for an app. We limit our selection to apps that support the Gradle build system, the official toolchain for Android [17], and use the Gradle version specified in the build settings. Out of 2,874 candidate apps we automatically built 1,339 (46.59%) apps successfully. The majority of failed builds only support Java 8 and do not work with Java 11, which has been supported since Gradle 5/Android 9 (Pie, API level 28, released August 2018) [16]. We discard these apps as this indicates they have not been updated for several years and do not support current Android versions.

Binary code compilation. To generate OAT files we implemented apk2oat, which can compile an app's bytecode in three settings:

- *Phone.* The first option is to install apps on a physical phone and execute the dex2oat version shipped with its Android firmware to compile them to OAT binaries suitable for the underlying hardware platform (almost exclusively ARM). This requires root privileges on the device, but only to access the output files of the compilation.
- *Emulator.* Installing apps on an emulator provides more flexibility and is easier to parallelize. At the time of our implementation, the emulator did not allow ARM emulators to be run on x86 hosts, prohibiting cross-compilation.[3] This limitation seems to have been removed since.
- *Host-based AOSP.* The Android Open Source Project (AOSP) also provides the source code for dex2oat. Thus, it is possible to build and run dex2oat on any arbitrary x86 host and cross-compile the apps to ARM OAT binaries because it allows specifying the target architecture.

In our experiments, we use the last option as it is the most convenient and scalable. Compared to our initial experiments on a physical phone we are not limited to targeting an Android version that we can root. Thus, we used the most current AOSP version at the time of writing, Android 13 (Tiramisu, `android-13.0.0_r3` released August 2022; API level 33, OAT version 225). Additionally to passing the corresponding ARM boot image from the AOSP to dex2oat, we use the `–generate-debug-info` flag and set the compiler to `filter=everything`, and `backend=optimizing`. The former setting explicitly does not change the generated code but adds, amongst other metadata, function symbols and DEX method names to make our analysis easier.

Bytecode baseline. To provide a comparison with a Dalvik-based approach, we run experiments with Soot (version 4.4.0, released January 2023) and SootUp (version 1.0.0, released December 2022). Because both analyze APKs and not OATs, we only investigate how many apps they can successfully analyze, i.e., can parse and prepare the Dalvik bytecode for further analysis. Soot is used as the basis for a variety of app analysis tools, such as FlowDroid [3] for data flow analysis and SimiDroid [38] for app similarity. SootUp is a recent overhaul of this framework and is not backward compatible.

[3] `https://android.googlesource.com/platform/external/qemu/+/1381d44efe69ba0b37fb7f7ef868125e279fc14a/android/emulator/main-emulator.cpp#910` (git version from January 13, 2022)

5.2 Ground Truth on Functions

ART provides oatdump to show information about an OAT file such as the command used to compile it, as well as the disassembly and, if available, Dalvik bytecode for each function in the binary. While its output is human-readable, it is not stable across versions yet and it is not well documented. We implemented a parser for AOSP version 13.0.0_r3 to extract the following information from the compiled OAT files for each of its functions:

- *signature*, consisting of package, class, and method names as well as argument types.
- *dex offset*, identifying the method uniquely in a DEX file.
- *availability*, indicating whether a method was compiled or not. dex2oat skips Dalvik methods that are marked as abstract, but also optimizes other methods away. This makes a comparison to methods reported by bytecode-based approaches challenging.
- *real offset*, identifying the position of the function in the binary OAT file. This is calculated by adding the code offset of the function to the global *oat offset*.
- *function size*, in bytes.

5.3 Selected Disassemblers

For our evaluation, we chose the following disassemblers:

- IDA Pro [31] (version 7.7). A commercial tool that allows for interactive and automated analysis with scripts and plugins.
- Binary Ninja [57] (version 2.3.2660). A commercial tool that supports instrumentation through Python.
- Ghidra [40] (version 9.2.3). An open-source tool with a custom disassembler that supports scripting and plugins. Due to its time and memory requirements, we disabled certain Analyzers that are not relevant to our evaluation, such as the `GCCExceptionAnalyzer` and Const Analyzers.
- radare2 [44] (version 5.7.7). An open-source tool with bindings in various languages that make it easy to instrument. It uses the Capstone disassembler for ARM64 and we instructed it to take symbols into account.
- angr [47] (version 9.2.6). An open-source tool that provides functionality for static as well as dynamic analysis. Experimental Android support uses Soot in the background.

In addition, we used the BinExport [20] plugin version 12@-432130350 for IDA Pro, Ghidra, and Binary Ninja. It is a format used to export analysis data for downstream applications, such as BinDiff [66] for binary similarity. For all disassemblers and BinExport we created wrappers to extract all known function offsets as well as their size, when available. IDA Pro and BinExport do not provide a size for the detected functions. Instead, we have to sum the length of all basic blocks that are in the functions flow graph to determine the size of a function.

We identified additional tools but did not include them in our evaluation due to the already significant engineering effort required in creating the dataset as well as the analysis pipeline. BAP [5], Hopper [6], and ddisasm [24] support automated analysis for ARM64 and we plan to include them in future work. Furthermore, while LIEF is an ideal candidate that can not only parse Dalvik bytecode but also OAT files, it currently only supports parsing OAT files up to version 138 (Android 9), and for this reason, we had to discard it.

6 RESULTS

6.1 Overall Success Rate

We could reliably compile all 1,339 APKs to OAT files "ahead-of-time, " i.e., in their entirety, using the host-based AOSP approach (see Section 5.1) in just 2.68 seconds per app on average. Almost all tools successfully finished analysis of all OATs as well, only Ghidra failed to provide results due to resource constraints in our setup: At first, we only could analyze 1,315 before we increased the available memory available from 20 GB to 50 GB. This resulted in a total of 1,333 apps being successfully analyzed by Ghidra. However, even then a couple of apps caused it to exhaust the assigned memory and hit the timeout of 45 minutes per app, which we set for all tools.

The success rates of the binary disassemblers on the OAT files are generally higher than for Soot, which we used to process the bytecode in the original APK files. For 78 (5.83%) apps, Soot was not able to finish the analysis at all due to an 8-year-old open issue concerning unhandled opcodes.[4] Its successor, SootUp, successfully processed all APKs but failed to lift seven methods in five apps to Java code for further analysis.

6.2 Function Boundary Detection

Ground truth. Analyzing the OATs with oatdump succeeded for all apps and yielded a total of more than 37.08 million functions. Of those, 2.08 million functions (0.06%) did not have any binary code in the OAT file. We inspected some of the most common functions without code and found them to be either declared abstract in the Java/Kotlin source or optimized away by dex2oat (e.g., by inlining or dead code elimination). Additionally, 6.76 million functions shared their code. This means their Dalvik bytecode was compiled to identical binary code and dex2oat saved storage space by pointing to the same offset instead of duplicating the memory. This left us with 28.24 million unique function offset and size pairs across all apps. None of the functions that shared an offset had different sizes. The number of unique functions per app in an OAT file is on average 21,093 with a median of 16,928.

Disassembler performance. We present the detailed results across disassemblers in Table 1 and a summary in Figure 2. For each function identified by a disassembler, we compare it to the information obtained from oatdump. If the offset and size match, we have a *full match*. If only the offset matches but the size is not correct, it is a *offset match* (i.e., partial match). If the function starts at an unknown offset, it is a *superfluous* function. If a disassembler could not identify a function it is *not found*, e.g. when analysis of the containing file failed or the preceding function boundary claims the next code block. An ideal situation would be to have only full matches. With default settings, none of the evaluated tools achieve this result and show varying capabilities to produce a trustworthy basis for further code analysis relying on correctly identified functions, such as code similarity on a function granularity.

IDA Pro has the highest recall and was able to correctly retrieve 98.13% of functions, matching their offset as well as size while Ghidra only found 79.27%. However, the BinExport plugin for IDA Pro has a low precision with only 35.65% correctly identified functions, second to last only with the performance of the BinExport plugin for Binary Ninja, which reports just 35.17% of functions correctly. radare2 performs very well and has the second highest precision with 92.93%, but does not have a BinExport plugin.

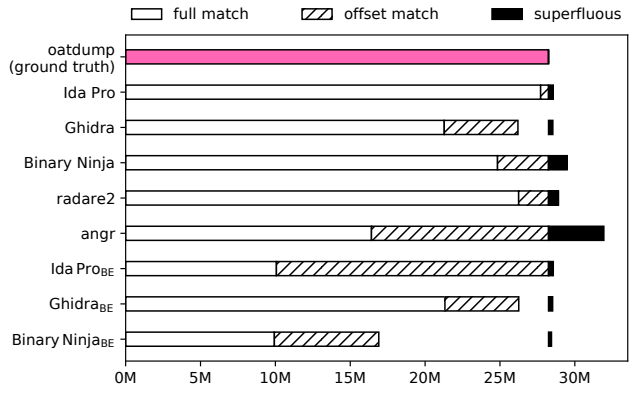

Figure 2: *Summary of disassembled function boundaries compared to oatdump's ground truth.* We evaluated the disassemblers and available BinExport plugins ($_{BE}$) by comparing the identified function offsets and sizes to what oatdump reports. A *full match* constitutes a match of the function offset and its size, for an *offset match* the size differs, and if an offset is returned by the disassembler that is not present in the oatdump output, we mark it as *superfluous*.

When investigating the apps with the lowest scores across all disassemblers we found that they have very few methods and the majority of them are used as "data apps" for other apps. For example, the flight information system app `player.efis.pfd` uses six additional apps to provide terrain data for various regions, each of those having only six identifiable methods according to oatdump.

The stark differences between IDA Pro and Binary Ninja and their BinExport files are surprising and warrant future research. We speculated at first there was a mistake in our parsing of BinExport files because it uses a complex layout to prevent malicious binaries to explode the state. However, this difference to the direct disassembler output is not present for Ghidra. The missing functions in the Binary Ninja BinExport are likely due to unaddressed "`TODOs`" in the plugin in the version we used as it is considered "Pre-Release."[5]

7 DISCUSSION & FUTURE WORK

Takeaways. Our results show that analyzing apps as binary OAT files is an imperfect but promising alternative to Dalvik-based bytecode analyzers–in particular considering that we evaluated disassemblers in their default configuration without leveraging any metadata available through oatdump (yet). Overall, the success rate is still higher than that of bytecode-based Soot when it comes to the number of successfully processed apps, up to 100% compared to 94.17%. The recently released SootUp improves over Soot, but it also fails to analyze all methods due to a `MethodTooLargeException`. SootUp first lifts Dalvik to Java bytecode before using a Java analysis library. Unlike Java, however, Dalvik does not have a maximum

[4] `https://github.com/soot-oss/soot/issues/331` (opened January 2015)

[5] `https://github.com/google/binexport/releases/tag/v11-20201202-rc0` (January 2021)

PROCESSED APPS PER TOOL			DETECTED FUNCTIONS				PERFORMANCE	
Name (version)	Availability	Success	Full Match	Only Offset	Not Found	Superfluous	Precision	Recall
Bytecode: Soot (4.4.0)	🙯	1,261 (94.17%)	-	-	-	-	-	-
Bytecode: SootUp (1.0.0) ∗	🙯	1,339 (100.00%)	-	-	-	-	-	-
oatdump (13.0.0_r3)	🙯	1,339 (100.00%)	28.24M (100.00%)	-	-	-	1.00	1.00
IDA Pro (7.7)	$	1,339 (100.00%)	27.72M (98.13%)	0.53M (1.87%)	0.00M (0.00%)	0.31M (1.11%)	0.97	0.98
Ghidra (9.2.3)	🙯	1,333 (99.55%)	22.39M (79.27%)	5.23M (18.51%)	0.63M (2.22%)	0.30M (1.08%)	0.80	0.79
Binary Ninja (2.3.2660)	$	1,339 (100.00%)	24.84M (87.94%)	3.40M (12.04%)	0.01M (0.02%)	1.24M (4.38%)	0.84	0.88
radare2 (5.7.7)	🙯	1,339 (100.00%)	26.25M (92.93%)	1.99M (7.06%)	0.00M (0.00%)	0.66M (2.35%)	0.91	0.93
angr (9.2.6)	🙯	1,339 (100.00%)	16.42M (58.13%)	11.83M (41.87%)	0.00M (0.00%)	3.69M (13.08%)	0.51	0.58
IDA Pro$_{\text{BinExport}}$ (v12)	$	1,339 (100.00%)	10.07M (35.65%)	18.17M (64.35%)	0.00M (0.00%)	0.32M (1.12%)	0.35	0.36
Ghidra$_{\text{BinExport}}$ (v12)	🙯	1,333 (99.55%)	22.39M (79.27%)	5.23M (18.51%)	0.63M (2.22%)	0.29M (1.03%)	0.80	0.79
Binary Ninja$_{\text{BinExport}}$ (v12)	$	1,339 (100.00%)	9.93M (35.17%)	6.97M (24.69%)	11.34M (40.14%)	0.20M (0.70%)	0.58	0.35

Table 1: *Results of running disassemblers (with and without the BinExport plugin;* 🙯*=open source,* $*=commercial tool) on OAT files.* The success rate indicates how many files were successfully analyzed and did not crash the tool or time out after 45 minutes. We ran Soot and SootUp on the original APKs but did not process its output (∗ SootUp exited successfully on all apps but failed to analyze 7 functions in 5 apps). Full matches for oatdump indicate the expected functions, i.e., each unique offset that a function has since multiple Dalvik methods can be compiled to the same binary function. All percentages relate to this expected number of functions. *Full Match* indicates the offset and size are matching, *Only Offset* indicates the disassembler identified a function but its size is not correct. *Superfluous* functions are ones identified at offsets that do not match any known functions obtained from oatdump, conversely, *Not Found* are known functions a disassembler failed to locate.

method size of 65,535 bytes, and the currently included version of dex2jar does not support breaking up these methods. We filed an issue with SootUp to update its dependencies.[6]

One challenge is still the volatility of the OAT format itself, which is why we opted against assuming that the disassemblers could handle OAT files and their metadata. While LIEF provides a dedicated OAT parser, it is not kept up to date and demonstrates the difficulty in creating robust analysis tools in a steadily changing environment.

We also show what differences exist between the disassemblers that are often seen as interchangeable steps in a bigger analysis pipeline. Especially for binary analysis, there are not many high-level tools available that are compatible with multiple disassemblers, making comparisons difficult. Additionally, we did not evaluate multiple compiler settings for OAT files that might aggravate those differences (and might already be applied to pre-installed apps that do not have an APK or even Dalvik bytecode available). Since our goal is to provide a robust foundation for analyzing apps as binaries, we hope this offers a critical view on comparing downstream tools using different disassemblers.

Finally, we opted to create a dataset that is reproducible, but also possible to keep up to date in order to evaluate future changes to the Android build system. While the exact set of apps that is possible to be built through F-Droid will change over time, the implementation of such an approach can be replicated and we plan to release our data and code in a future update to this paper.

Limitations. We only compare binary-based and bytecode-based tools on their success rate of analyzing files and do not yet apply further processing like call graph reconstructions or assessing downstream tooling. Because some methods in bytecode are optimized away by dex2oat, a direct comparison is challenging. We also do not include native libraries for now, which would bias the results as bytecode-only-based tools have no option to include them.

Since `oatdump` depends on the additional `.vdex` file, which contains Dalvik code, to create its output and is part of ART, we trust it to create a list of expected methods. However, the OATs contain several thunk functions that are not declared in the `oatdump` output but can be considered as valid methods, which explains some superfluous functions. For example, angr is very eager to declare a function at any byte sequence that can be interpreted as one or more instructions. We chose to interpret the results conservatively and nevertheless declare those functions as false positives. The precision is thus a lower bound.

Since our dataset is based on apps from a market with a focus on privacy and transparency, they do not represent the apps found on the more popular Google Play Store. Especially the use of advertisement or closed-source libraries is severely underrepresented, as well as the use of commercial compilers, optimizers, and obfuscators. Including those apps would introduce a source of uncertainty for downstream analysis as they do not provide any reliable information about build settings, compilers, or libraries used. Additionally, apps that use WebViews will have little Dalvik or native code included in the APK to be analyzed.

Future work. A widely used application of static app analysis is to compute the similarity of functions or whole apps. This enables use cases such as malware and library detection, identifying vulnerabilities automatically, or attributing privacy-violating behavior. One common technique for computing the similarity of code is to lift it and re-compile it with optimizations, which serves as normalization before further processing. In the case of ART, the compilation to binary code uses an optimizing compiler and thus can be leveraged

[6] `https://github.com/soot-oss/SootUp/issues/563` (opened March 2023)

for such tasks. Our next step is to identify how much this affects analysis and evaluate downstream binary tooling for app analysis. Even if binary tools are not perfect in finding function boundaries, their reliability to analyze apps without additional assumptions could outweigh the effort to keep specialized Android-tooling up to date. Unless there is a reason for Google to abandon the compilation of Dalvik bytecode to binary code, or deprecate ahead-of-time compilation, general-purpose binary tools will continue to work with OATs.

Analyzing the bytecode as binary code has the additional benefit that the same tools can be used to analyze the app's main code as well as its native libraries. So far, it was necessary to develop tools that could represent Dalvik and native code at the same time. By utilizing ART's dex2oat compiler, it is possible to find a unified representation of an app's Dalvik and native code. This however requires additional instrumentation since the way the Android framework, native libraries, and the binary app code are loaded into memory depend on ART.

While our approach and evaluation were motivated by our frustration with finding and running "State-of-the-Art" Android analysis tools, a more detailed survey on the state of app analysis will be beneficial. The changes in the Android build pipeline described in Section 2 pose significant challenges to approaches that, e.g., assume package, class, and method signatures to be preserved, or unused library code not to be discarded. While binary analysis might seem to be at a disadvantage by not having access to the high-level Dalvik metadata, it has the advantage of being more mature and expecting even more adversarial code.

Finally, OATs can be used as an additional dataset for binary tooling. This can help to evaluate how well binary analysis tools work in a different context and their ability to work with a compiler besides the widely benchmarked `clang` and `gcc`.

8 CONCLUSION

By leveraging ART's dex2oat compiler, we evaluate the disassembly of ahead-of-time compiled APKs to binary OATs from a function boundary perspective. For this, we compiled 1,339 open-source apps to OAT files and analyzed them with IDA Pro, Ghidra, Binary Ninja, radare2, and angr. We also evaluated the BinExport plugin (where available) and compared the reported function boundaries with a ground truth obtained from oatdump. While no disassembler showed perfect results in finding all functions, the automated and unguided disassembly is failing only in a few cases and enables further processing of the apps. We envision that our approach and evaluation can provide a foundation for future research on combining app and binary analysis, such as code clone detection for malware analysis, library identification, attribution, and more.

ACKNOWLEDGEMENTS

This research received funding from the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19056], and SBA Research (SBA-K1), a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

REFERENCES

[1] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz. "BinRec: Dynamic Binary Lifting and Recompilation". In: *Proc. of the European Conference on Computer Systems (EuroSys)*. 2020.

[2] D. Andriesse, A. Slowinska, and H. Bos. "Compiler-Agnostic Function Detection in Binaries". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps". In: *ACM SIGPLAN Notices* 49.6 (June 2014).

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

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform". In: *Proc. of the International Conference on Computer Aided Verification (CAV)*. Source code: `https://github.com/BinaryAnalysisPlatform/bap`. 2011.

[6] Cryptic Apps. *Hopper*. 2023. URL: `https://www.hopperapp.com` (visited on 03/29/2023).

[7] R. David and A. Challande. *An Experimental Study of Different Binary Exporters*. Sept. 2019. URL: `https://web.archive.org/web/20230403134319/https://blog.quarkslab.com/an-experimental-study-of-different-binary-exporters.html` (visited on 03/28/2023).

[8] Y. David, N. Partush, and E. Yahav. "Similarity of Binaries through Re-Optimization". In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017.

[9] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee. "Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale". In: *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[10] F-Droid. *F-Droid Server Tools Repository*. `https://gitlab.com/fdroid/fdroidserver/`. 2023. (Visited on 11/30/2021).

[11] Facebook. *Redex: An Android Bytecode Optimizer*. Source code: `https://github.com/facebook/redex`. Mar. 2023. URL: `https://fbredex.com` (visited on 03/28/2023).

[12] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez. "An Analysis of Pre-installed Android Software". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 2020.

[13] P. Garba and M. Favaro. "SATURN - Software Deobfuscation Framework Based On LLVM". In: *Proc. of the ACM Workshop on Software Protection (SPRO)*. 2019.

[14] Google. *Android 2.3 Platform and Updated SDK Tools*. Dec. 2010. URL: `https://web.archive.org/web/20230403133705/https://android-developers.googleblog.com/2010/12/android-23-platform-and-updated-sdk.html` (visited on 03/28/2023).

[15] Google. *Experimental New Android Tool Chain - Jack and Jill*. 2017. URL: `https://web.archive.org/web/20230403133308/http://tools.android.com/tech-docs/jackandjill` (visited on 03/29/2023).

[16] Google. *Android Gradle plugin release notes: 3.4.0*. Apr. 2019. URL: `https://web.archive.org/web/20210427035226/https://developer.android.com/studio/releases/gradle-plugin#3-4-0` (visited on 03/28/2023).

[17] Google. *Configure your build*. `https://web.archive.org/web/20230403133647/https://developer.android.com/studio/build`. 2021. (Visited on 03/28/2023).

[18] Google. *Shrink, obfuscate, and optimize your app*. `https://web.archive.org/web/20230403134402/https://developer.android.com/studio/build/shrink-code`. 2021. (Visited on 04/03/2023).

[19] Google. *Android's Kotlin-first approach*. https://web.archive.org/web/20230403133512/https://developer.android.com/kotlin/first. 2023. (Visited on 03/28/2023).

[20] Google. *BinExport*. 2023. URL: https://github.com/google/binexport (visited on 10/06/2022).

[21] Google. *Configuring ART*. https://web.archive.org/web/20230403133204/https://source.android.com/docs/core/runtime/configure. 2023. (Visited on 04/03/2023).

[22] Google. *Get started with the NDK*. https://web.archive.org/web/20230403133421/https://developer.android.com/ndk/guides. 2023. (Visited on 03/24/2023).

[23] Google. *Sign your app*. https://web.archive.org/web/20230403134222/https://developer.android.com/studio/publish/app-signing. 2023. (Visited on 03/29/2023).

[24] GrammaTech. *Datalog Disassembly*. Mar. 2023. URL: https://github.com/GrammaTech/ddisasm (visited on 03/28/2023).

[25] Guardsquare. *ProGuard and R8: Comparing Optimizers*. https://web.archive.org/web/20230403133539/https://www.guardsquare.com/blog/proguard-and-r8. July 2018. (Visited on 03/28/2023).

[26] Guardsquare. *Comparison of ProGuard vs. R8: October 2019 edition*. Oct. 2019. URL: https://web.archive.org/web/20230403133603/https://www.guardsquare.com/blog/comparison-proguard-vs-r8-october-2019-edition (visited on 03/28/2023).

[27] Guardsquare. *ProGuard: Shrink your Java and Android code*. Source code: https://github.com/Guardsquare/proguard. 2023. URL: https://www.guardsquare.com/proguard (visited on 03/29/2023).

[28] K. Hageman, Á. Feal, J. Gamba, A. Girish, J. Bleier, M. Lindorfer, J. Tapiador, and N. Vallina-Rodriguez. "Mixed Signals: Analyzing Software Attribution Challenges in the Android Ecosystem". In: *IEEE Transactions on Software Engineering* (2023).

[29] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller. "Honey, I Shrunk Your App Security: The State of Android App Hardening". In: *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. 2018.

[30] G. Hecht and A. Bergel. "Quantifying the adoption of Kotlin on Android stores: Insight from the bytecode". In: *Proc. of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft)*. 2021.

[31] Hex-Rays. *IDA Pro*. 2023. URL: https://hex-rays.com/ida-pro/ (visited on 03/29/2023).

[32] J. Huang, N. Borges, S. Bugiel, and M. Backes. "Up-To-Crash: Evaluating Third-Party Library Updatability on Android". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019.

[33] J. Huang, O. Schranz, S. Bugiel, and M. Backes. "The ART of App Compartmentalization: Compiler-Based Library Privilege Separation on Stock Android". In: *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[34] Y. Ishii, T. Watanabe, F. Kanei, Y. Takata, E. Shioji, M. Akiyama, T. Yagi, B. Sun, and T. Mori. "Understanding the Security Management of Global Third-Party Android Marketplaces". In: *Proc. of the ACM SIGSOFT International Workshop on App Market Analytics (WAMA@ESEC/SIGSOFT FSE)*. 2017.

[35] M. Jiang, Q. Dai, W. Zhang, R. Chang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren. "A Comprehensive Study on ARM Disassembly Tools". In: *IEEE Transactions on Software Engineering* (2022).

[36] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren. "An Empirical Study on ARM Disassembly Tools". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2020.

[37] L. Li, T. Bissyandé, and J. Klein. "Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark". In: *IEEE Transactions on Software Engineering* 47.4 (2021).

[38] L. Li, T. F. Bissyandé, and J. Klein. "SimiDroid: Identifying and Explaining Similarities in Android Apps". In: *Proc. of the IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*. Source code: https://github.com/lilicoding/SimiDroid. 2017.

[39] N. Mauthe, U. Kargén, and N. Shahmehri. "A Large-Scale Empirical Study of Android App Decompilation". In: *Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021.

[40] National Security Agency (NSA). *Ghidra*. Source code: https://github.com/NationalSecurityAgency/ghidra. 2021. URL: https://ghidra-sre.org (visited on 10/08/2021).

[41] Z. Ning and F. Zhang. "DexLego: Reassembleable Bytecode Extraction for Aiding Static Analysis". In: *Proc. of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018.

[42] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes. "The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 2018.

[43] F. Pauck, E. Bodden, and H. Wehrheim. "Do Android Taint Analysis Tools Keep Their Promises?" In: *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018.

[44] *radare*. Source code: https://github.com/radareorg/radare2. URL: https://rada.re/ (visited on 10/08/2021).

[45] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor. "*droid: Assessment and Evaluation of Android Application Analysis Tools". In: *ACM Computing Surveys* 49.3 (2016).

[46] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li. "Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study". In: *Proc. of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2021.

[47] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. Source code: https://github.com/angr/angr. 2016.

[48] soot-oss. *Soot*. Mar. 2023. URL: https://github.com/soot-oss/soot (visited on 03/28/2023).

[49] soot-oss. *SootUp*. Mar. 2023. URL: https://github.com/soot-oss/SootUp/ (visited on 03/29/2023).

[50] StackOverflow. *What Percentage of Android Devices Runs on x86 Architecture?* Nov. 2018. URL: https://android.stackexchange.com/questions/186334/what-percentage-of-android-devices-runs-on-x86-architecture/213663#213663 (visited on 03/28/2023).

[51] Statcounter. *Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share. 2023. (Visited on 02/02/2023).

[52] Statista. *Average number of new Android app releases via Google Play per month from March 2019 to October 2022*. https://web.archive.org/web/20230403134010/https://www.statista.com/statistics/1020956/android-app-releases-worldwide/. Oct. 2022. (Visited on 02/02/2023).

[53] Statista. *Number of available applications in the Google Play Store from December 2009 to September 2022*. https://web.archive.org/web/20230403133902/https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Sept. 2022. (Visited on 02/02/2023).

[54] M. Sun, T. Wei, and J. C. Lui. "TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime". In: *Proc.*

of the ACM SIGSAC Conference on Computer and Communications Security (CCS). 2016.

[55] R. Thomas. *LIEF Documentation: Android Formats*. URL: https://web.archive.org/web/20230403133359/https://lief-project.github.io/doc/latest/tutorials/10_android_formats.html (visited on 03/29/2023).

[56] Unity. *Mobile (Android) Hardware Stats*. Mar. 2017. URL: https://web.archive.org/web/20170808222202/http://hwstats.unity3d.com/mobile/cpu-android.html (visited on 03/28/2023).

[57] Vector 53. *Binary Ninja*. 2023. URL: https://binary.ninja (visited on 03/28/2023).

[58] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu. "Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets". In: *Proc. of the Internet Measurement Conference (IMC)*. 2018.

[59] M. Y. Wong and D. Lie. "Tackling Runtime-based obfuscation in Android with TIRO". In: *Proc. of the USENIX Security Symposium*. 2018.

[60] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan. "NDroid: Toward Tracking Information Flows Across Multiple Android Contexts". In: *IEEE Transactions on Information Forensics and Security* 14.3 (2019).

[61] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART". In: *Proc. of the USENIX Security Symposium*. 2017.

[62] G. You, G. Kim, S. Han, M. Park, and S.-j. Cho. "Deoptfuscator: Defeating Advanced Control-flow Obfuscation Using Android Runtime (ART)". In: *IEEE Access* 10 (2022).

[63] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang. "TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices". In: *IEEE Transactions on Dependable and Secure Computing* 17.1 (2020).

[64] X. Zhan, T. Liu, Y. Liu, Y. Liu, L. Li, H. Wang, and X. Luo. "A Systematic Assessment on Android Third-party Library Detection Tools". In: *IEEE Transactions on Software Engineering* (2021).

[65] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele. "AppJitsu: Investigating the Resiliency of Android Applications". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021.

[66] Zynamics. *BinDiff*. https://www.zynamics.com/bindiff.html. 2021. (Visited on 06/25/2021).