Anestis Bechtsoudis

Security Engineer – CENSUS S.A.
anestis@census-labs.com
@anestisb

# Fuzzing Objects d'ART

Digging Into the New Android L Runtime Internals

## Abstract

In an effort to deal with performance challenges in the Android ecosystem, Google has made an investment aiming to fully replace the old JIT Dalvik VM with the brand new AOT (Ahead-Of-Time) ART runtime. It has been more than a year since ART was open-sourced and its first production releases are reaching the market. However, there is currently almost zero public knowledge about the security maturity of ART and its interfacing functionality.

This talk is the first milestone of a greater research effort aiming to analyze all of the new ART runtime internals, depict the exploitation impact of identified bugs in the Android ecosystem and mark the requirements for the development of new tools. To assist this analysis, the first DEX file format smart fuzzing engine has been implemented supporting a series of rulesets mirroring the various fuzzing requirements. The input generation and fuzzing toolset we have developed run directly on Android devices and monitor the investigated processes.

DEX smart fuzzing techniques and evaluation metrics will be presented against the initial target of the ART runtime, which is the bytecode optimization and compilation chain (DEX parser, IR processing & code generation) for the ARM architecture. In order to prove the efficiency of our smart fuzzing techniques, we compare our results against dumb fuzzing iterations with identical characteristics.

## Introduction

ART has become the default (and only supported) runtime since version 5.0 (Lollipop) of the Android operating system, aiming to greatly improve execution performance of Java applications. ART was firstly introduced in October 2013 at Android KitKat release as a beta option available for supported devices under development mode. While being open-sourced for more than a year, very limited knowledge exists about the security maturity of ART components and their interfacing functionality in the Android ecosystem. In an effort to fill this knowledge gap a large research project has been started aiming to analyze the new runtime against its three major functionalities: Compilation (bytecode optimization), Runtime Initialization and Runtime Execution.

The ART Dalvik bytecode (DEX) compilation process is the first milestone of this research project. To assist our source code review and functional analysis, we have developed a new fuzzing framework highly optimized for Android devices. DEX file format smart fuzzing techniques have been developed under this framework aiming to get better security testing coverage for the target ART compilation chain units. The core fuzzing tool components (data generation & executors) have been designed using native C code aiming to run efficiently on the low-resource target devices.

This paper describes the design decisions behind the ART fuzzing framework components, the evaluation metrics used to improve the fuzzing framework's performance and the initial results from the fuzzing process.

## Previous Work

There is very little previous work from the information security community that has been published on ART components security. The most noticeable publications are the following:

- dexFuzz project (Stephen Kyle, ARM)
- State of the ART: Exploring the New Android KitKat Runtime (Paul Sabanal, HITB2014AMS, IBM X-Force)
- Hiding Behind ART (Paul Sabanal BHASIA2015, IBM X-Force)
- Android Internals: A Confectioner's Cookbook (Jonathan Levin)
- Introduction to Android 5 Security (Lukas Aron, Petr Hanacek)
- The State of ASLR on Android Lollipop (Daniel Micay, COPPERHEAD)

While not being designed for security fuzz testing, the project that drew our attention during framework development is the recently released dexFuzz tool. It is the first public DEX structural mutations tool that has been designed to detect ART compiler errors using differential testing techniques. dexFuzz has been merged to AOSP upstream and is available under the master development branch.

## ART Runtime Basics

Within this section we provide the reader a brief overview of the ART components and the essential background information required to understand our implementation approaches of the DEX smart fuzzing and the framework itself. Since the out-of-process fuzzing executors are designed to run on the device, we have chosen a system-oriented approach for this introduction. A detailed analysis of all ART components and supported file formats is outside of the scope of this paper.

One of the first user-mode components that are executed after system has successfully booted, is the system ART runtime initialization. *Figure-1* illustrates the basic functions that are invoked as part of the root Zygote process initialization.

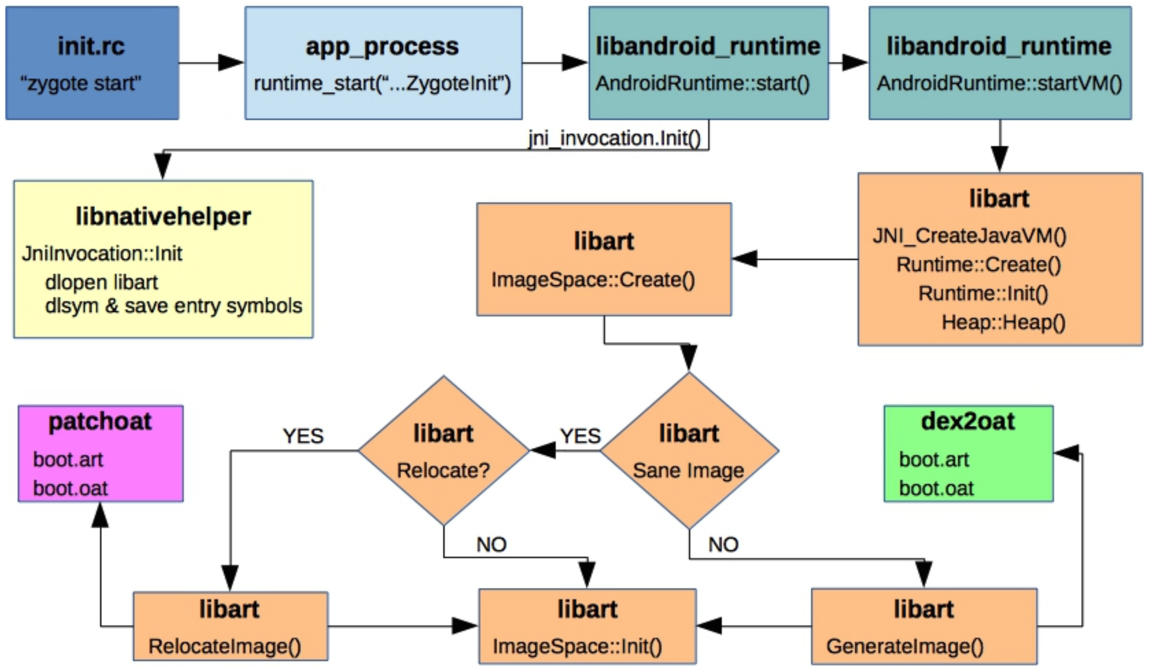


*Figure 1 - System runtime initialization flow*

*ImageSpace* class in *libart* will detect the current state of system image files and will continue with the creation (dex2oat) or delta repair (patchoat) actions if dalvik-cache does not contain a valid version of the required files.

While the previous Dalvik runtime was relying on JIT compilation techniques to optimize DEX bytecode execution, ART is using a series of AOT (Ahead-Of-Time) techniques to pre-optimize bytecode into native code at installation time (before execution). *dex2oat* is the main compile driver tool developed for ART to complete this process for the supported target architectures (ARM, ARM64, x86, x86_64, MIPS, MIPS64). The compile driver is invoked with a series of compilation & runtime configuration arguments which control the optimization and execution context of the bytecode.

ART supports two compiler backends in libart-compiler, Quick and Optimizing. As illustrated in *Figure-2*, Quick backend is using two intermediate representations (IR) to apply the various optimization iterations before invoking the code generation process. Quick has been designed to be reliable and fast sacrificing native code performance. On the other hand Optimizing backend is using a single IR graph where multiple optimization passes are invoked against, aiming for better performance at generated code. At the time of writing, Quick backend is the current default option, while Optimizing backend is still under heavy development.

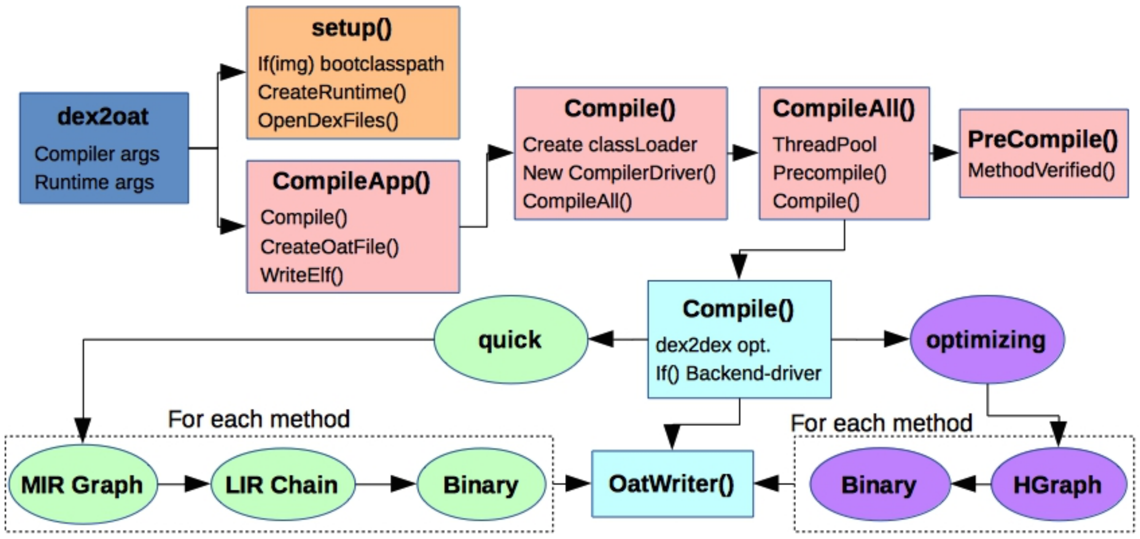


*Figure 2 - Bytecode optimization process*

With an exception of some optimization related instruction opcodes[1] used during internal dex-to-dex transformation steps, ART is compatible with the old Dalvik executables[2]. DEX file format has 18 basic sections (not all of them are mandatory) and is heavily using relative referencing between items of these sections and their inner encoded data tables. Members of items on each section or encoded inner blob can be categorized into four representative groups:

1. Index (_idx) references to items in other sections
2. Relative offset (_off) references to items in other sections
3. Data placeholders (mostly of implicit size)
4. Attribute metadata (from predefined enumeration lists with accepted values)

*Figure-3* illustrates an example of section items references for some of mandatory sections.

[1] https://github.com/anestisb/oatdump_plus#dalvik-opcode-changes-in-art
[2] https://source.android.com/devices/tech/dalvik/dex-format.html

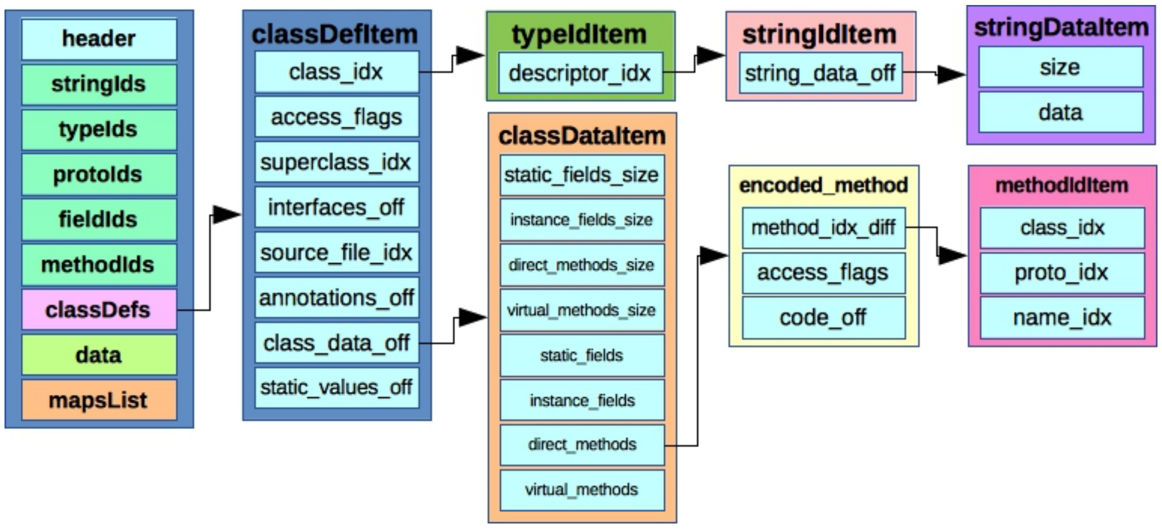


*Figure 3 - DEX file format structure references example from some basic sections*

## Fuzzing Framework Design

The first step when designing an out-of-process fuzzing project is to identify the target library binaries and a driver executable that consumes them. Considering that ART has only one (so far) compiler driver (dex2oat), the fuzzing target executable was easily defined. *dex2oat* is dynamically linked to both libart and libart-compiler libraries and is capable to trigger all supported DEX optimization configurations that we want to fuzz test.

Having identified the target executable, the next step is to select the target platform architecture. Since dex2oat supports cross-compilation our first thought was to utilize ART host tools (Linux or Darwin) and benefit from a more powerful environment compared to the Android OS ARM/ARM64 or x86/x86_64 devices. After a quick source code analysis this plan has been abandoned due to the different memory layout of the ART host tools. Some noticeable examples are the base libc heap allocator and the emulated ashmem (Android shared memory). As a result we turned our focus into native Android OS devices. Considering that device runtime ISA (kRunrtimeISA flag in source code) affects compilation and runtime attributes of ART (different instruction set features, ART threads stack layout, etc.), we have decided to work against ARM devices since they have the biggest market share of Android OS devices. Android QEMU emulator for ARM has been also disqualified as an acceptable option due to different CPU variant (generic) that affects compilation attributes, and its poor performance.

With the target environment configuration having been defined the final design step was to decide on the fuzzing strategy. Building upon the decision to execute core fuzzing actions against native ARM devices, significantly better execution performance can be achieved by generating the test data also on the target device. On a different scenario campaigns will face a huge I/O overhead between host and target.

Test data generation can be achieved by following either a mutation-based or a generation-based approach. Mutation fuzzing techniques (random, block-based, ruleset-based, etc.) are applying a series of mangling routines against a valid (for the target) set of existing input data. On the other hand generation-based fuzzing tools are using as an input a complete grammar or model capable to describe the target data structure in order to generate the test cases. Based on our team's previous experience with executable file formats fuzzing projects and considering the low-resource system that data generation engine will run, we decided to proceed with a mutation-based approach. In an effort to improve the mutations efficiency against the target ART compiler, data mangling algorithms have been designed to honor part of the basic DEX file structure requirements. Evolution of these algorithms (smart mutation rules) is achieved in a manual way by analyzing the feedback data that have been collected from the learning campaigns (*Figure-4*).

Fuzzing framework components can be divided into two levels: target device and host. Target device level incorporates the data generation engine, the fuzzing core and a set of post-running helper tools (crash verifier, minimizer, etc.). The fuzzing core implementation relies on a fork() – exec() approach to spawn the fuzzing executors and utilizes POSIX signals to detect abnormal process termination. At the host level we have the AOSP build server to generate the appropriate build type (coverage, ASAN, etc.) for the target device and the crashes classifier component. Crash classifier is utilizing remote GDB and supported python scripting to debug generated crash triggers in an effort to detect unique crashes and annotate some of their characteristics. Unique crashes are detected by generating a backtrace hash signature for each verified file.

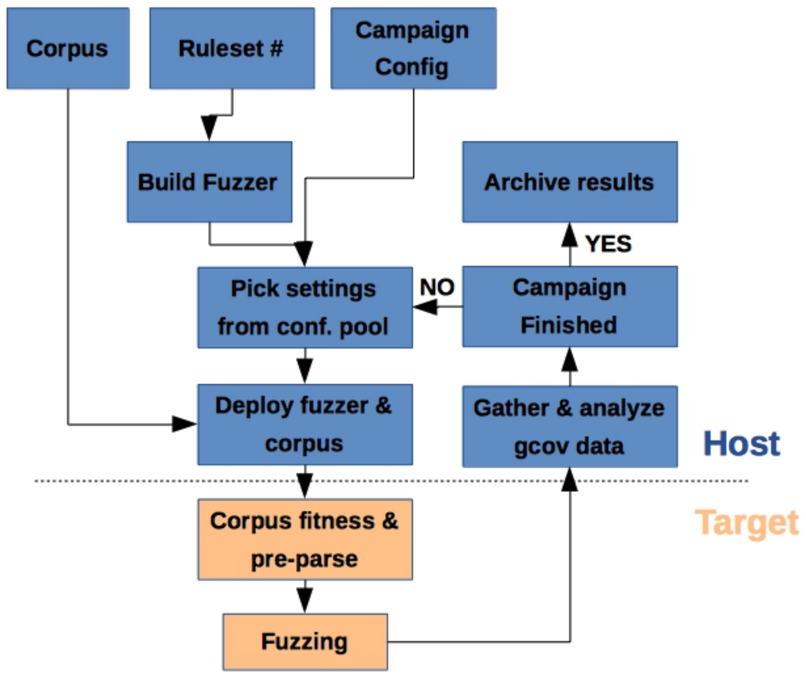


*Figure 4 - Learning phase*

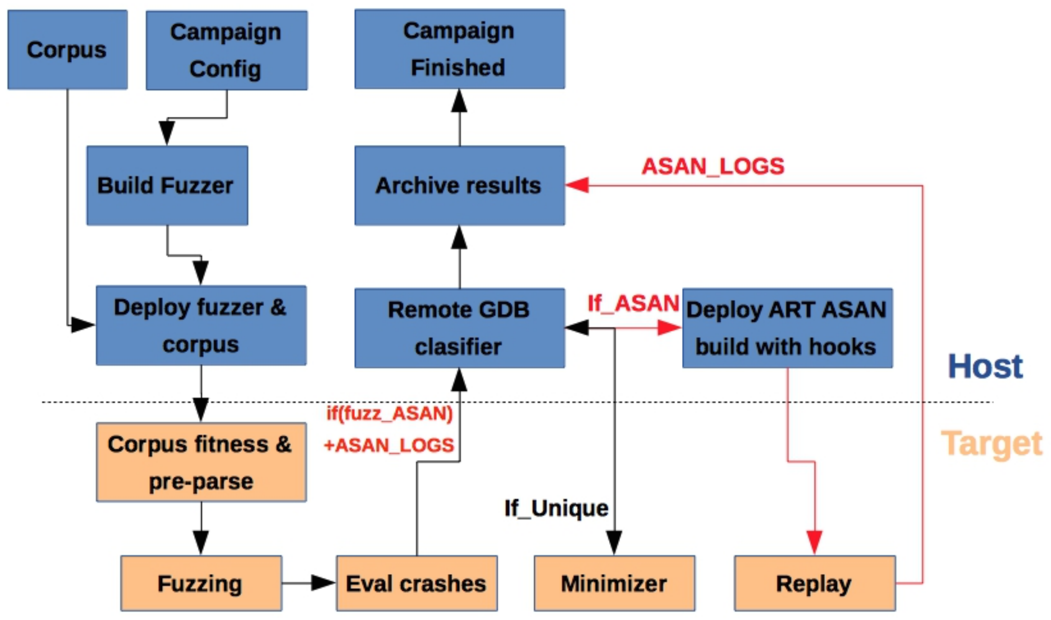


*Figure 5 - Execution phase*

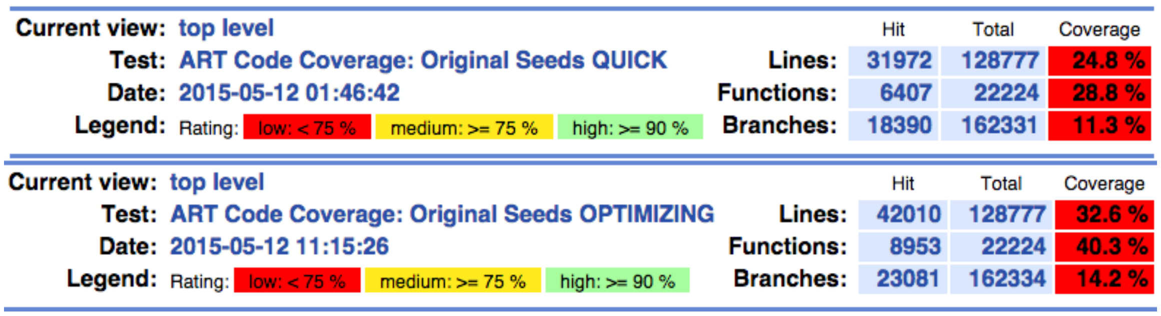
## Measuring Performance

In order to measure the fuzzing executors' performance two well-known techniques from software testing are used: code-coverage and execution hit counters. Code coverage data aim to measure the percentage of source code that has been reached (executed) by the fuzzing campaigns. While this information offered a valuable input for the mutation rule-sets evolution, our experiments quickly identified that it is not capable on its own to reliably measure the fuzzer's performance due to extensive DEX input validation applied by ART before bytecode optimization. As a result hit counter data have been collected for all DEX validation driver routines in order to accurately measure the percentage of mutated data that had successfully passed them.

Since AOSP (Android Open Source Project) is using the GCC cross-compile toolchain to build Android for ARM devices, we have used the built-in GCOV code coverage functionality to create ART profiling builds for the target Nexus devices in our lab. Per-iteration generated coverage data had to be collected and transferred from the device to the host, since the basic-block reconstruction metadata (.gcno files) are stored in the host (cross-profiling). The lcov tool is used to convert GCOV coverage data into usable data blobs for automated analysis. Finally genhtml tool has been also used to generate visual coverage reports for the purposes of this paper.

## Mutation Rules Evolution

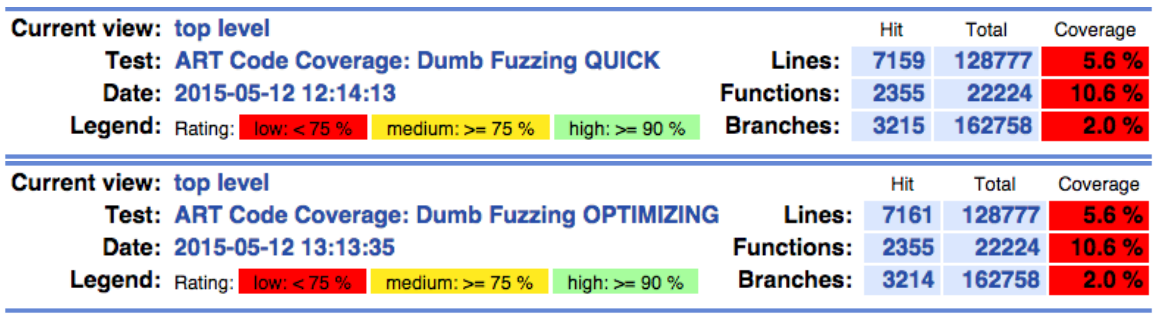
As part of our initial framework we developed a random mutation engine prototype to examine the behavior of a completely blind (dumb) fuzzing approach. Considering the basic DEX file structure we have excluded the DEX header from the mutated data. Additionally, a post-mangle CRC repair function has been implemented to patch the file checksum in the file header before executor invokes the target dex2oat binary. As illustrated by *Figure-6* and *Figure-7*, a dumb fuzzing approach is exercising less than 1/3 of the code, compared to the coverage of the original seeds that were used as input for the random mutations. These poor results of random mutations led us to develop a series of DEX smart fuzzing rules in order to efficiently fuzz the ART compiler.



| Current view: | top level | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Test: | ART Code Coverage: Original Seeds QUICK | Lines: | 31972 | 128777 | 24.8 % |
| Date: | 2015-05-12 01:46:42 | Functions: | 6407 | 22224 | 28.8 % |
| Legend: | Rating: low: < 75 % medium: >= 75 % high: >= 90 % | Branches: | 18390 | 162331 | 11.3 % |

| Current view: | top level | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Test: | ART Code Coverage: Original Seeds OPTIMIZING | Lines: | 42010 | 128777 | 32.6 % |
| Date: | 2015-05-12 11:15:26 | Functions: | 8953 | 22224 | 40.3 % |
| Legend: | Rating: low: < 75 % medium: >= 75 % high: >= 90 % | Branches: | 23081 | 162334 | 14.2 % |

*Figure 6 - Code coverage of original seed DEX files for QUICK & OPTIMIZING backends*



| Current view: | top level | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Test: | ART Code Coverage: Dumb Fuzzing QUICK | Lines: | 7159 | 128777 | 5.6 % |
| Date: | 2015-05-12 12:14:13 | Functions: | 2355 | 22224 | 10.6 % |
| Legend: | Rating: low: < 75 % medium: >= 75 % high: >= 90 % | Branches: | 3215 | 162758 | 2.0 % |

| Current view: | top level | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Test: | ART Code Coverage: Dumb Fuzzing OPTIMIZING | Lines: | 7161 | 128777 | 5.6 % |
| Date: | 2015-05-12 13:13:35 | Functions: | 2355 | 22224 | 10.6 % |
| Legend: | Rating: low: < 75 % medium: >= 75 % high: >= 90 % | Branches: | 3214 | 162758 | 2.0 % |

*Figure 7 - Code coverage using random mutations for QUICK & OPTIMIZING backends*

Considering the complexity and large scale of the DEX section item references, we expect that an equally extensive verification process must be executed by the runtime prior to processing any input DEX files. A closer look at a dex2oat systrace capture (as illustrated in *Figure-8*) while compiling a single DEX file demonstrates that significant time is consumed during the DEX file open and class verify stages, indicating a two layer DEX file verification process.

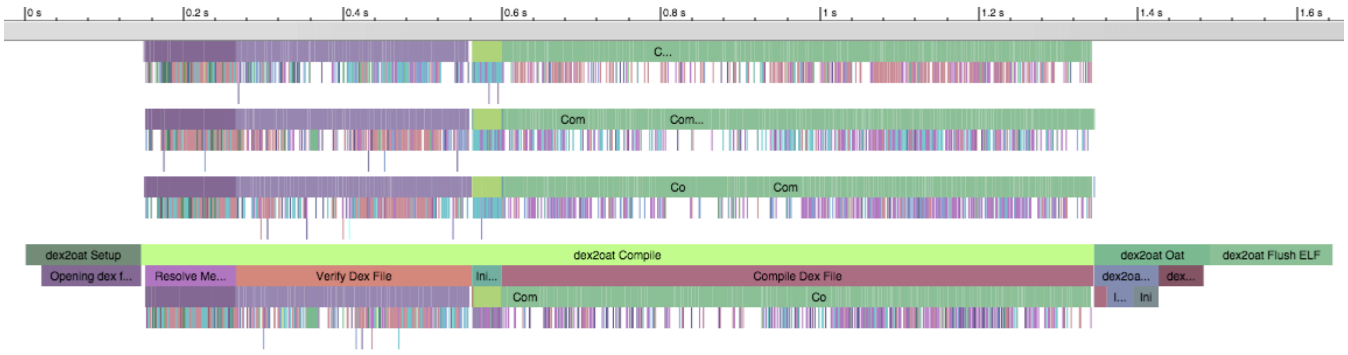


*Figure 8 - Systrace of dex2oat for a single DEX file in a 4core device*

The first DEX verification layer (*DexFileVerifier::Verify()*) is executed by the main thread as part of the open file action and is responsible of applying a series of quick static checks against the input file. These initial checks aim to verify the basic attributes of the file header (CRC, size, presence of mandatory sections, etc.) and to ensure that all relative references are complying to the file structure rules (CheckIntraSection() and CheckInterSection()). The second layer (*MethodVerifier::VerifyMethod()*) is executed in parallel from the worker threads during the class verification step. Layer2 is performing a deeper investigation by applying static and control-flow analysis against all class methods code items. Parallelization for class verification, initialization and compilation action occurs on the class level from a pool of POSIX pthreads.

Our initial attempt to introduce intelligence to DEX file mutation algorithms was to write a dedicated data mangling rule for 16 out of the 18 basic file sections. FileHeader and MapList are the two sections that have been excluded due to the nature of their data. The mutation routine of each rule limits the data generated to those that are appropriate for the matching section. Section ranges (start offset and size) are extracted from input seed files as part of the pre-parsing process using the information stored in MapList section. The MapList table includes an entry for each section item, storing the section's start relative offset and size in items. *Table-1* and *Table-2* demonstrate the achieved code coverage and DEX file verifier success ratio for section mutation rule developed during the first learning phase.

Results indicate that coverage performance has been significantly improved compared to random dumb mutations. However, the average DEX verifier success ratio is still very poor for most of the sections due to relative referencing violations. In order to further improve mutation algorithms an additional set of inner and combinational rules has been created as part of learning phase2. More specifically the sections' external structure has been honored during data mangling forcing altered data to conform to the referenced target range. For example if the class_idx member of a classDef item has been marked for mangling, the mutation algorithm is ensuring that new class_idx value will be within the accepted range of typeIds section. These inner rules can be then further chained together creating more complex mutation algorithms that still honor the biggest part of the file structural dependencies. *Table-3* and *Table-4* demonstrate the improved DEX verifier level1 and level2 success ratio for 6 rule-groups that have been promoted for phase2 due to their coverage efficiency. Each table entry includes the average hit ratio after a random choice of supported inner rule for 5K iterations.

*Table 1 - Level1 mutation rules code coverage for QUICK & OPTIMIZING backends*

| | Quick | | | Optimizing | | |
|---|---|---|---|---|---|---|
| **Ruleset** | **Lines** | **Functions** | **Branches** | **Lines** | **Functions** | **Branches** |
| Original | 24.80% | 28.80% | 11.30% | 32.60% | 40.30% | 14.20% |
| Dumb | 5.60% | 10.60% | 2.00% | 5.60% | 10.60% | 2.00% |
| stringIdItems | 23.80% | 28.50% | 10.40% | 31.20% | 39.50% | 13.10% |
| typeIdItems | 23.90% | 28.50% | 10.60% | 31.50% | 39.70% | 13.40% |
| protoIdItems | 24.70% | 28.80% | 11.20% | 32.30% | 40.10% | 14.00% |
| fieldIdItems | 24.70% | 28.80% | 11.20% | 32.20% | 40.10% | 14.00% |
| methodIdItems | 24.70% | 28.80% | 11.20% | 32.00% | 39.90% | 13.80% |
| classDefItems | 24.80% | 28.80% | 11.30% | 32.40% | 40.10% | 14.10% |
| typeList | 24.70% | 28.80% | 11.20% | 32.20% | 40.10% | 13.90% |
| annotationSetRefList | 24.50% | 28.70% | 11.20% | 32.30% | 40.10% | 14.00% |
| annotationSetItems | 24.50% | 28.70% | 11.10% | 31.90% | 39.90% | 13.80% |
| classDataItems | 24.50% | 28.70% | 11.00% | 32.10% | 39.90% | 13.80% |
| **codeItems** | **25.10%** | **28.90%** | **11.40%** | **32.80%** | **40.30%** | **14.30%** |
| stringDataItems | 24.40% | 28.70% | 10.90% | 32.10% | 40.00% | 13.80% |
| debugInfoItems | 24.70% | 28.80% | 11.30% | 32.50% | 40.20% | 14.20% |
| annotationItems | 24.60% | 28.70% | 11.20% | 32.40% | 40.20% | 14.10% |
| **encodedArrayItems** | **24.90%** | **28.90%** | **11.40%** | **32.70%** | **40.30%** | **14.30%** |
| annotationsDirectoryItems | 24.40% | 28.70% | 11.00% | 32.30% | 40.10% | 13.90% |

*Table 2 - Level1 mutation rules DEX verify levels success & fail ratio hit counters for QUICK & OPTIMIZING backends*

| | Quick | | | Optimizing | | |
|---|---|---|---|---|---|---|
| **Ruleset** | **Level1** | **Level2** | | **Level1** | **Level2** | |
| | **PASSED** | **HRD FAIL** | **SFT FAIL** | **PASSED** | **HRD FAIL** | **SFT FAIL** |
| Original | 100.00% | 0.00% | 1.58% | 100.00% | 0.00% | 1.58% |
| Dumb | 0.00% | - | - | 0.00% | - | - |
| stringIdItems | 0.14% | 0.29% | 7.72% | 0.32% | 0.00% | 5.33% |
| typeIdItems | 0.42% | 0.00% | 0.15% | 0.30% | 0.00% | 0.72% |
| protoIdItems | 12.64% | 0.00% | 2.58% | 12.14% | 0.00% | 1.78% |
| fieldIdItems | 8.72% | 0.06% | 1.06% | 8.60% | 0.06% | 0.72% |
| methodIdItems | 6.22% | 0.32% | 1.19% | 6.34% | 0.33% | 1.01% |
| classDefItems | 25.18% | 0.02% | 1.27% | 25.46% | 0.02% | 1.03% |
| typeList | 4.58% | 0.00% | 1.23% | 4.14% | 0.00% | 1.81% |
| annotationSetRefList | 4.38% | 0.00% | 1.53% | 4.34% | 0.00% | 1.31% |
| annotationSetItems | 0.78% | 0.00% | 10.58% | 0.50% | 0.00% | 8.15% |
| classDataItems | 3.82% | 0.12% | 0.77% | 3.76% | 0.08% | 1.91% |
| codeItems | 44.02% | 1.11% | 1.32% | 42.52% | 1.08% | 1.58% |
| stringDataItems | 6.88% | 0.00% | 1.18% | 7.26% | 0.01% | 0.92% |
| debugInfoItems | 45.20% | 0.00% | 1.41% | 46.04% | 0.00% | 1.96% |
| annotationItems | 9.62% | 0.00% | 5.87% | 10.06% | 0.00% | 6.39% |
| encodedArrayItems | 55.80% | 0.00% | 1.61% | 55.74% | 0.00% | 1.81% |
| annotationsDirectoryItems | 0.40% | 0.00% | 4.03% | 0.60% | 0.00% | 6.08% |

*Table 3 – Phase2: DEX verify level1 success ratio for sample random inner rule per section and annotations chain rule*

| | Quick | | Optimizing | |
|---|---|---|---|---|
| **Ruleset** | **Phase1** | **Phase2** | **Phase1** | **Phase2** |
| protoIdItems | 12.64% | 12.79% | 12.14% | 13.78% |
| fieldIdItems | 8.72% | 31.47% | 8.60% | 32.06% |
| methodIdItems | 6.22% | 38.72% | 6.34% | 38.78% |
| classDefItems | 25.18% | 37.35% | 25.46% | 37.26% |
| codeItems | 44.02% | 92.30% | 42.52% | 97.80% |
| annotations_chain | - | 22.98% | - | 22.54% |

*Table 4 – Phase3: DEX verify level2 fail ratio for sample random inner rule per section and annotation chain rule*

| | Quick | | Optimizing | |
|---|---|---|---|---|
| **Ruleset** | **Phase1** | **Phase2** | **Phase1** | **Phase2** |
| | **HARD FAIL** | **HARD FAIL** | **HARD FAIL** | **HARD FAIL** |
| protoIdItems | 0.00% | 0.49% | 0.00% | 0.91% |
| fieldIdItems | 22.74% | 38.30% | 17.99% | 35.90% |
| methodIdItems | 22.85% | 47.76% | 24.64% | 45.12% |
| classDefItems | 0.74% | 15.98% | 0.70% | 17.38% |
| codeItems | 86.56% | 15.60% | 86.33% | 15.34% |
| annotations_chain | - | 0.00% | - | 0.00% |

## Fuzzing Results

ART fuzz testing has been conducted against 4 Google Nexus devices (1 x N4, 2 x N5, 1 x N6) using the latest (at the time of writing) 5.1.x Android OS production release and AOSP master branch under ART commit #8e8bb8a (April 16, 2015). The Nexus4 device has not been tested under master AOSP branches due to small compatibility issues that were left unresolved during our analysis. Clang Address Sanitizer instrumentation builds have been generated only for the ART master branch against the same commit.

*Table-5* and *Table-6* demonstrate the number of unique (major or major + minor) identified bugs for each device. The number of bugs listed under the optimizing column are unique to the optimizing compiler and do not include bugs that have been triggered due to quick compiler failover. Additionally, *Figure-9* and *Figure-10* summarize vulnerability types for the crash triggers that have been analyzed so far by our team.

Table 5 Android 5.1.x unique crashes

| Device | QUICK | | OPTIMIZING | |
|---|---|---|---|---|
| | Major | Major.Minor | Major | Major.Minor |
| Nexus4 | 22 | 34 | 17 | 24 |
| Nexus5 | 31 | 49 | 23 | 28 |
| Nexus6 | 36 | 52 | 26 | 32 |

Table 6 - Android master ART commit #8e8bb8a unique crashes

| Device | QUICK | | OPTIMIZING | |
|---|---|---|---|---|
| | Major | Major.Minor | Major | Major.Minor |
| Nexus5 | 27 | 49 | 18 | 32 |
| Nexus5 ASAN | 9 | 15 | 13 | 17 |
| Nexus6 | 32 | 58 | 14 | 23 |
| Nesus6 ASAN | 13 | 25 | 9 | 13 |

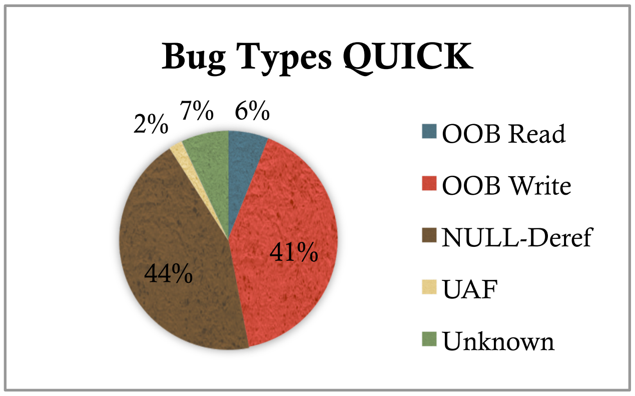


Figure 10 – Vuln. types for QUICK compiler

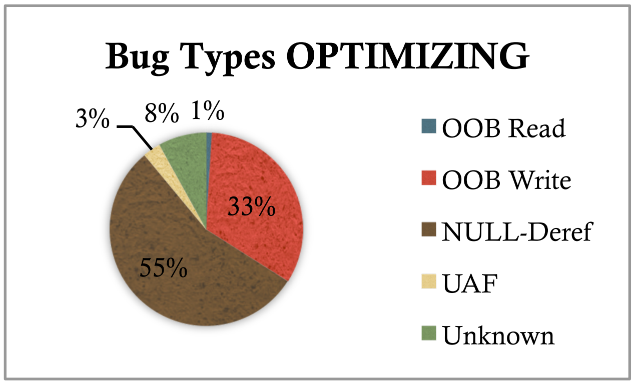


Figure 9 – Vuln. types percentage for OPTIMIZING compiler

## References

- dexFuzz project (Stephen Kyle, ARM)
- State of the ART: Exploring the New Android KitKat Runtime (Paul Sabanal, HITB2014AMS, IBM X-Force)
- Hiding Behind ART (Paul Sabanal BHASIA2015, IBM X-Force)
- Android Internals: A Confectioner's Cookbook (Jonathan Levin)
- Android Hacker's Handbook (Joshua J. Drake, Zach Lanier, Collin, Pau Oliva Fora, Stephen A. Ridley, Georg Wicherski)
- Introduction to Android 5 Security (Lukas Aron, Petr Hanacek)
- The State of ASLR on Android Lollipop (Daniel Micay, COPPERHEAD)
- Matteo Franchin - ART's Quick Compiler: an unofficial overview
- Making Software Dumber (Tavis Ormandy)
- DEX format spec: https://source.android.com/devices/tech/dalvik/dex-format.html
- Android ART official documentation: https://source.android.com/devices/tech/dalvik/configure.html
- AFL Fuzzer (Michał Zalewski): http://lcamtuf.coredump.cx/afl/technical_details.txt
- LLVM LibFuzzer: http://llvm.org/docs/LibFuzzer.html
- Melkor ELF Fuzzer (Alejandro Hernández, IOActive): https://github.com/IOActive/Melkor_ELF_Fuzzer