Fuzzing Objects d' ART

Digging Into the New Android L Runtime Internals

Anestis Bechtsoudis (@anestisb)
CENSUS S.A. - anestis@census-labs.com

HITBSecConf
Amsterdam 2015

Who am I

- Security engineer at CENSUS S.A.
  - Vulnerability research, reverse engineering, cryptography and network security
  - Lately focusing into researching access control, exploitation mitigation and integrity protection techniques for mobile and embedded systems
- Previous (academic) research
  - Side-channel cryptanalysis (FPGA / embedded devices)
  - Network protocols & implementation stacks abuse
- Obsessed with vulnerability hunting challenges

Outline

- Android L ART Runtime 101
  - Bytecode optimization & execution paths
  - ART components, attack surface & security bugs impact
- Developing ART compiler fuzzing toolset
  - Techniques to increase DEX fuzzer intelligence
  - Feedback data used for fuzzer evolution
- Fuzzing results
- Q & A

Warning

- Not aiming to fully cover
  - ART runtime functionality
  - DEX, OAT, ART file formats details
- Fuzzing techniques not designed to be generic
- ART under heavy development
  - OAT ver. 045 at 5.1.x, OAT ver. 062 at master
- Work in progress
- No free bugs ☹ (well, sort of)

Motivation

- ART security maturity
  - New code
  - Compilers hard to audit

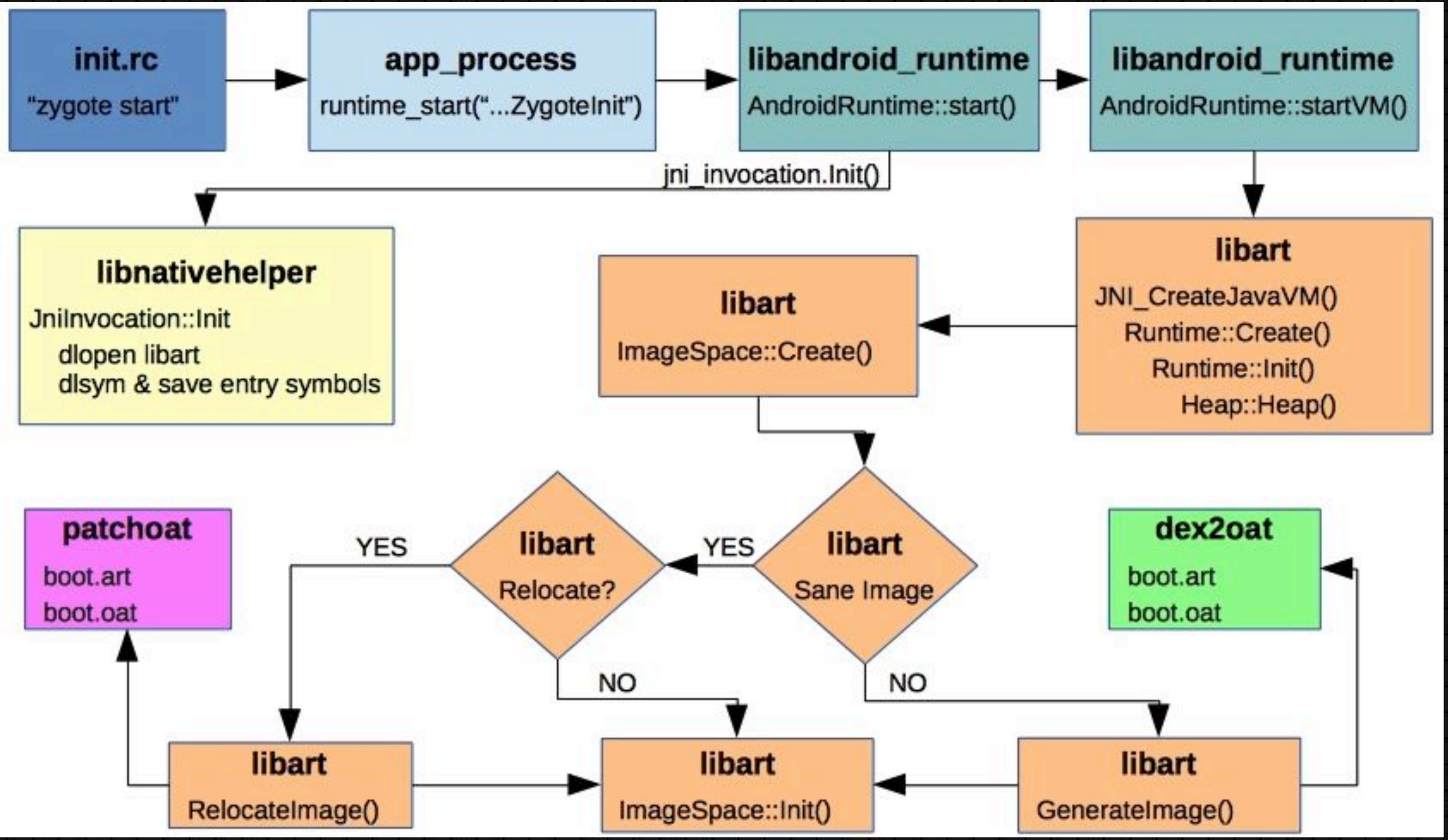
- Investigate optimization techniques
  - Compiler backends support cross-optimizations
- No public research on DEX security fuzzing
- Case study to research Android L ecosystem

Related Work

- dexFuzz project (Stephen Kyle, ARM)
  - Merged at ART upstream
- State of the ART: Exploring the New Android KitKat Runtime (Paul Sabanal, HITB2014AMS)
- Android Internals: A Confectioner's Cookbook (Jonathan Levin)
- Introduction to Android 5 Security (Lukas Aron, Petr Hanacek)

ART Runtime 101

Runtime Initialization



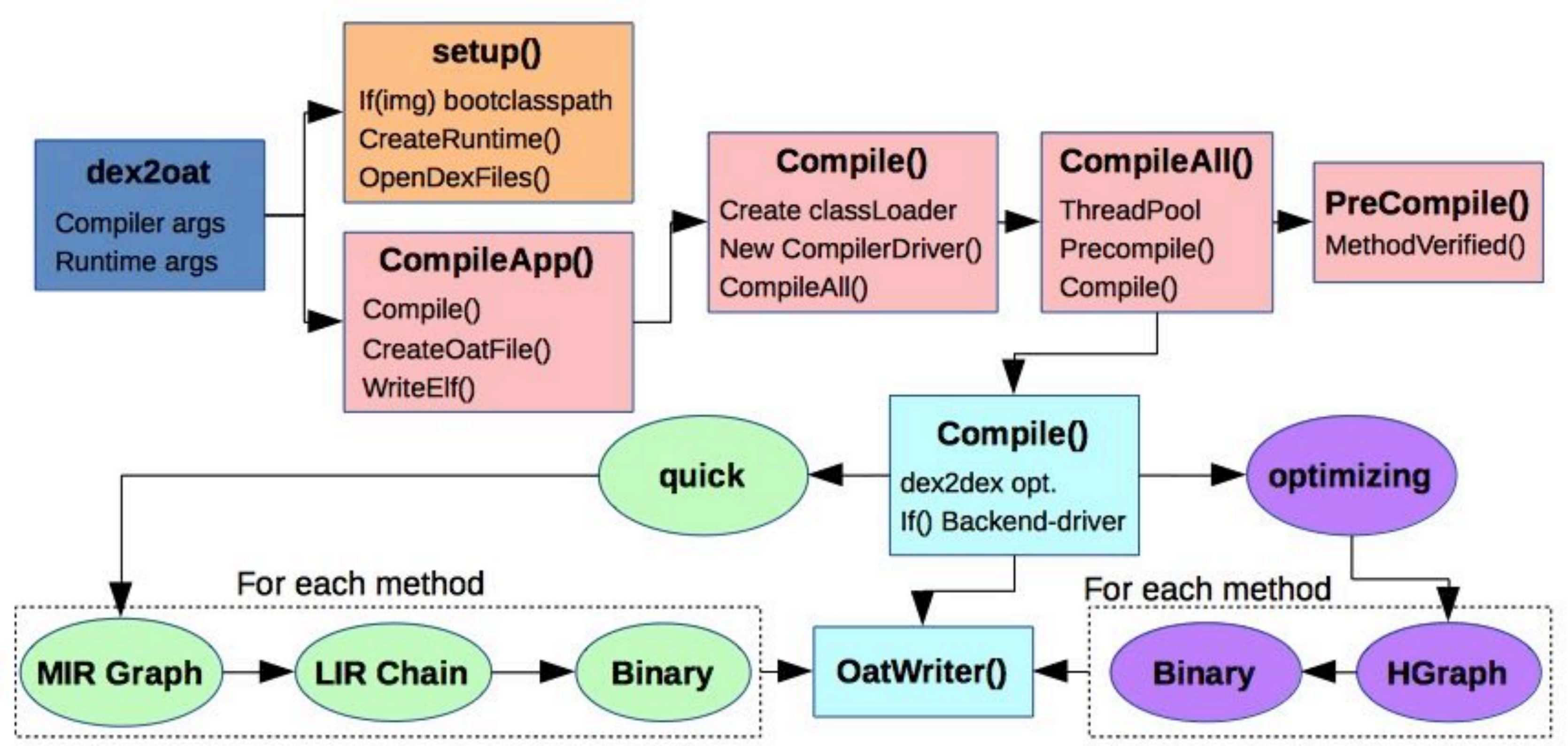
ART Components

- dex2oat: Ahead-of-Time (AOT) compiler
  - Dalvik bytecode (DEX) to native code (OAT) compilation
  - Generates ART image & framework / user-apps OAT
- patchoat: Relocate pre-optimized files
  - ART image & OAT files (--include-patch-information)
  - Delta patching: e.g. *ApplyOatPatchesTo(".text", delta_)*
- dalvikvm: Spawn standalone runtime
- oatdump: Image & OAT files disassembler
  - Our oatdump++ patches have merged upstream

ART File Formats

- ART image file (.art) – Usually labeled boot.art
  - Compacted heap of pre-initialized classes & related objects
    - Objects with absolute pointers within image
    - Absolute pointers from methods in the image to their code in oat
    - Absolute pointers from code in oat to methods in image
  - Is mapped before & links with matching (boot.)oat file
  - Needs to know where OAT will be loaded
- OAT file (.oat)
  - ELF dynamic shared object (page-able)
  - .rodata (oatdata), .text(oatexec, oatlastword), .oat_patches
  - OAT methods can be symbolicated (--include-debug-symbols)

Bytecode Optimization



Compiler Backends

- Method is the basic compilation unit
- Quick (default)
  - MIRGraph: 1 DEX Op – 1 MIR Node (+pseudo for annot.)
  - LIRChain: 1 instr – 1 LIR Node (+pseudo for annot.)
  - Sequence of nodes (static graphs) – two incompatible IR
- Optimizing
  - Under heavy development (--dump-cfg, --dump-passes)
  - Delegates to Quick if it fails to optimize method
  - Multiple passes (SSA, intrinsics, dead_code, simplifier, etc.)
  - Dynamic graph – single IR
- LLVM portable is no longer supported

Bytecode Execution

- Runtime can execute ODEX (oat):
  - Using compiled (optimized) native methods impl. (default)
  - Interpreter
    - Low-end devices (apps compiled with interpret-only flag)
    - App debugging (partially or fully) & VMSafeMode
  - JIT (under dev.) – Welcome back exec. cache & JIT spraying
- Runtime suspend points
  - Checks in generated code to stop Java threads in safe way
  - Consistency at checkpoints for native-execution, runtime & interpreter
- Memory consistency at suspend points for:
  - Garbage Collection
  - Sampling profiler (data collected at suspend points)
  - Debugging (breakpoints): De-optimize and switch to inter. thread

ART Target List

- Compilation chains for supported backends
  - Primary target for fuzz testing Phase-1
- Runtime initialization
  - Planned as Phase-2 target
- Runtime execution modes
  - Planned as Phase-3 target
  - Will mainly focus into native execution paths
  - Big challenge due to huge parameters list

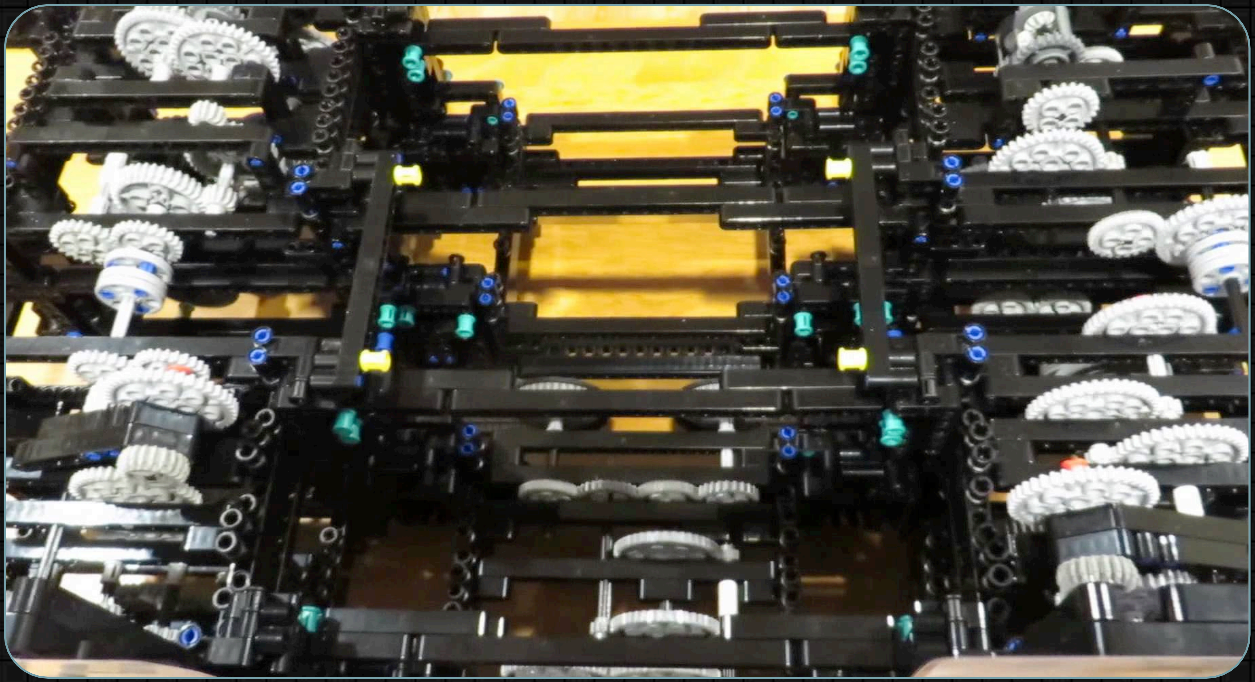
Value of ART Security Bugs

- Big attack surface
  - Java system services & user applications
  - system services running ART executables (e.g. installd)
- Bug consistency across the board
  - Stakeholders in distribution chain most probably won't modify ART components
  - Maybe different exploitation requirements per system
- Constantly improving Android security requiring chain of reliable bugs
- Possible arbitrary code exec at app install time

Exploiting ART Bugs

- Many processes may link to vulnerable code, although trigger is not guaranteed
- Exploitation impact
  - Privilege escalation
  - SELinux restrictions bypass (e.g. write dalvik-cache)
  - Android permissions escalation
  - Bypass 3rd party sandbox containers

Building the Fuzzing Env



Design Primitives

- Fuzzing target executable(s)?
- Target platform (ARM vs x86)?
  - Are x86 host tools a viable option?
  - QEMU emulator?
- Fuzzing strategy?
  - Data generation
  - Corpus selection
  - Monitor, Debug & Triage tools / techniques

Target Executable(s)

- Main target is libart-compiler
  - dex2oat
  - Jit::LoadCompiler (scoped for next phases)
- Get highest possible coverage for libart
- Using dex2oat binary as target
  - Fuzz test compilation chain supported backends
  - Input: DEX files, compilation & runtime settings

Target Platform

- OAT cross-compile available from host tools
  - mm build-art-host
  - Different memory layout (HAVE_ANDROID_OS)
    - ART base & GC heap allocators configuration
    - Emulated “ashmem”
- kRuntimeISA affects compiler & runtime parameters
  - Different instruction-set-features (mainly Optimizing)
  - ART runtime threads stack layout & entrypoints
- Android QEMU ARM emulator
  - CPU_VARIANT set to generic affecting compiler options
  - Very slow…

Target Platform

- Majority of Android OS devices have ARM
    - Analysis closest to production line setups
- Nexus family ideal for on-device fuzzing
    - Less effort for custom builds (specially against master)
    - Small Android L adoption from other vendors
- Fuzzing lab with 1 x N4, 2 x N5, 1 x N6
    - ARM64 out of scope for now

Fuzzing Strategy

- Mutation based:
  - Random (dumb) fuzzing (e.g. honggfuzz, zzuf)
  - Block-based (structure-aware) fuzzing (e.g. SPIKE)
  - Ruleset-based (smart) fuzzing (e.g. Melkor)
- Generation based:
  - Model interference assisted (e.g. PROTOS)
- Feedback driven evolutionary (self-learning)
  - Code coverage (e.g. AFL, LLVM LibFuzzer)
  - Symbolic Execution (e.g. SAGE)
  - Concolic Execution (e.g. jFuzz)

The Dumb Story

- Use code-coverage as comparison metric:
  - AOSP ARM binaries built with GCC toolchain (default)
  - Utilize GCC coverage instrumentation (--coverage)
  - Analyze data (gcov, lcov) & compare against original seeds
- Code a quick DEX file mutation random fuzzer
  - Use honggfuzz Android port as base
  - Implement a CRC repair post-mangle routine
- Pick a random pool of DEX seed files
  - Execute for various (small) mangle ratios

The Dumb Story

Current view: top level
Test: ART Code Coverage: Original Seeds QUICK
Date: 2015-05-12 01:46:42
Legend: Rating: low: < 75 % | medium: >= 75 % | high: >= 90 %

	Hit	Total	Coverage
Lines:	31972	128777	24.8 %
Functions:	6407	22224	28.8 %
Branches:	18390	162331	11.3 %

Current view: top level
Test: ART Code Coverage: Original Seeds OPTIMIZING
Date: 2015-05-12 11:15:26
Legend: Rating: low: < 75 % | medium: >= 75 % | high: >= 90 %

	Hit	Total	Coverage
Lines:	42010	128777	32.6 %
Functions:	8953	22224	40.3 %
Branches:	23081	162334	14.2 %

Current view: top level
Test: ART Code Coverage: Dumb Fuzzing QUICK
Date: 2015-05-12 12:14:13
Legend: Rating: low: < 75 % | medium: >= 75 % | high: >= 90 %

	Hit	Total	Coverage
Lines:	7159	128777	5.6 %
Functions:	2355	22224	10.6 %
Branches:	3215	162758	2.0 %

Current view: top level
Test: ART Code Coverage: Dumb Fuzzing OPTIMIZING
Date: 2015-05-12 13:13:35
Legend: Rating: low: < 75 % | medium: >= 75 % | high: >= 90 %

	Hit	Total	Coverage
Lines:	7161	128777	5.6 %
Functions:	2355	22224	10.6 %
Branches:	3214	162758	2.0 %

The Dumb Story

		Hit	Total	Coverage
Current view: top level - art/compiler - compiler.cc (source / functions)				
Test: ART Code Coverage: Dumb Fuzzing QUICK	Lines:	0	17	0.0 %
Date: 2015-05-12 12:14:13	Functions:	0	2	0.0 %
Legend: Lines: hit not hit | Branches: + taken - not taken # not executed	Branches:	0	7	0.0 %

```
               :      : namespace art {
               :      :
               :    0 : Compiler* Compiler::Create(CompilerDriver* driver, Compiler::Kind kind) {
   [ #  #  # ] :    0 :   switch (kind) {
               :      :     case kQuick:
               :    0 :       return CreateQuickCompiler(driver);
               :      :
               :      :     case kOptimizing:
               :    0 :       return CreateOptimizingCompiler(driver);
               :      :
               :      :     default:
               :    0 :       LOG(FATAL) << "UNREACHABLE";
               :    0 :       UNREACHABLE();
               :      :   }
               :      : }
               :      :
               :    0 : bool Compiler::IsPathologicalCase(const DexFile::CodeItem& code_item,
               :      :                                   uint32_t method_idx,
               :      :                                   const DexFile& dex_file) {
```

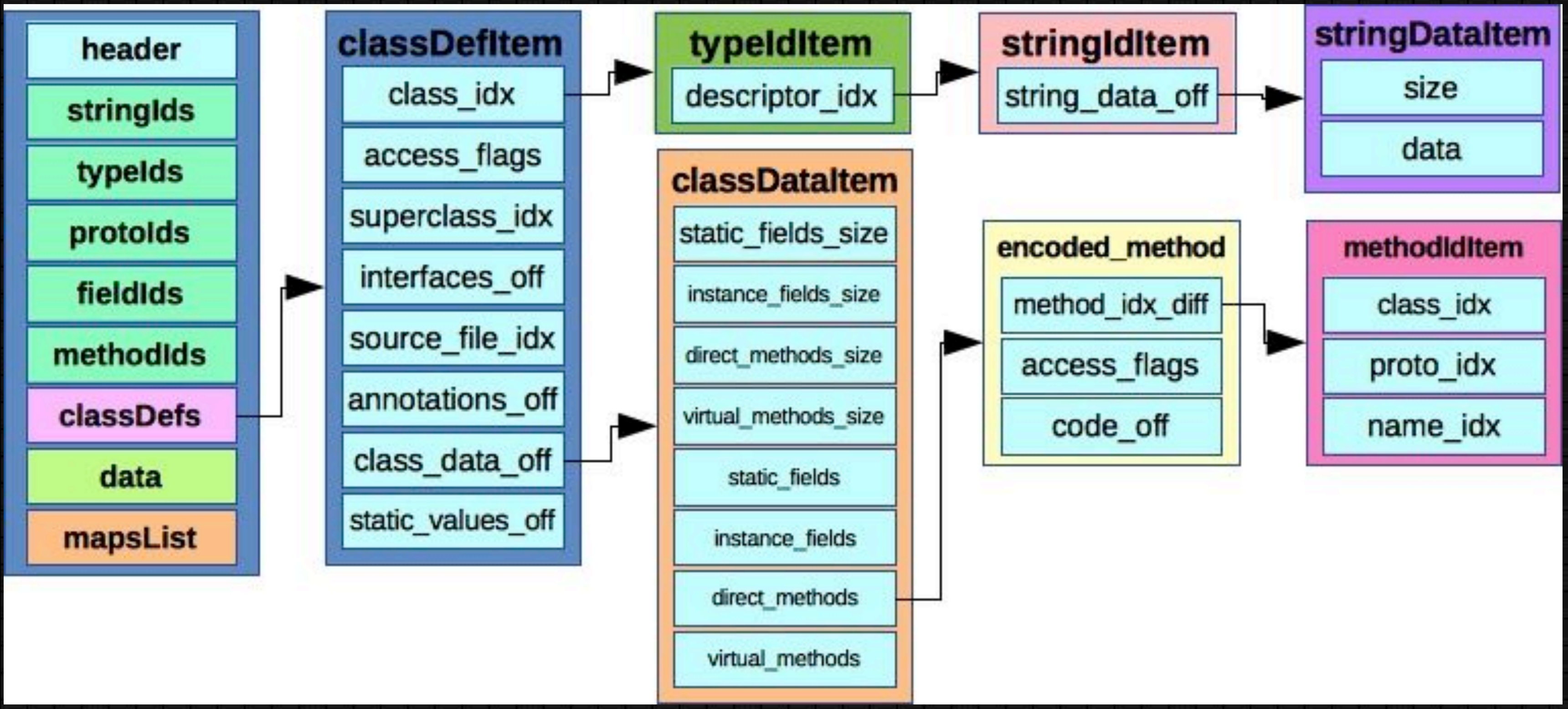
All 5K dumb iterations failed early

DEX Anatomy 101

- File Format Properties
  - Basic types + LEB128 (DWARF3-like) encoding
  - Relative indexing
  - Single file for all classes (stripped redundant constants)
  - 18 basic sections (more encoded data types internally)
    - Not all of them are mandatory (e.g. annotations)
    - Order doesn't matter for data sections
    - Implicit size requiring items parsing (e.g. codeItems)

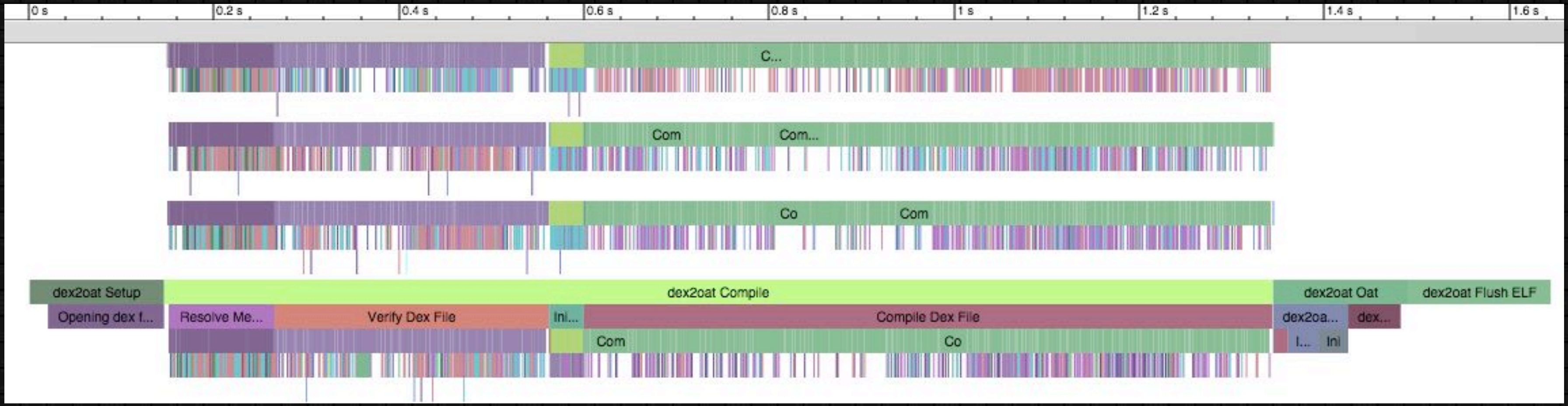
- Members of basic sections (roughly) categorized:
  - Index (Idx) references
  - Relative offset references (usually to items in data type sections)
  - Data placeholders (usually of implicit size)
  - Attribute metadata from predefined ENUM lists

DEX Anatomy 101



Visualizing Challenges

- Many strong dependencies / references between sections
- To what extent & where DEX validations are taking place?



Systrace of dex2oat with single DEX input using QUICK compiler

DEX File Verification L1

- *DexFileVerifier::Verify()* – Any single failure will abort compilation
  - *CheckHeader()*: Basic sanity checks (CRC, size, offsets range)
  - *CheckMap()*: Verify mapList section (order, sizes, data types, etc.)
  - *CheckIntraSection()*: Sections structure (padding, overlapping, size, etc.)
  - *CheckInterSection()*: Cross-section references (values sanity, ordering, etc.)

```
           :  5000 : bool DexFileVerifier::Verify() {
           :       :   // Check the header.
  [ - + ]: :  5000 :   if (!CheckHeader()) {
           :     0 :     return false;
           :       :   }
           :       :
           :       :   // Check the map section.
  [ - + ]: :  5000 :   if (!CheckMap()) {
           :     0 :     return false;
           :       :   }
           :       :
           :       :   // Check structure within remaining sections.
  [ + + ]: :  5000 :   if (!CheckIntraSection()) {
           :  4992 :     return false;
           :       :   }
           :       :
           :       :   // Check references from one section to another.
  [ + - ]: :     8 :   if (!CheckInterSection()) {
           :     8 :     return false;
           :       :   }
           :       :
           :     0 :   return true;
           :       : }
```

DEX File Verification L2

- *MethodVerifier::VerifyMethod()*
  - *VerifyInstructions()*: Code units static analysis, e.g.:
    - Execution cannot fall off the end of the code
    - Code does not end in the middle of the instruction
  - *CodeFlowVerifyMethod()*: Type safety & code-flow errors, e.g.:
    - Operand registers contain the correct type of values
    - Method invocation with correct arguments
- Fail types
  - Early: Reject entire class (e.g. no superClass)
  - Soft: Compiler tries, runtime re-verify enforced (e.g. except. handlers)
  - Hard: Entire class compilation is aborted (e.g. OOR register index):
  - Fatal: (SIG)Abort compilation (e.g. invalid method descriptor)

Data Generation Goals

- Improve fuzzing intelligence
  - Better code-coverage
    - Catch-up with original seed results
    - Find ways to improve
  - Increase DEX validation success ratio
    - Successfully pass Level1
    - Small number of Early, Hard and Fatal errors in L2
- Aim for good performance in a limited env
  - Data generation should happen on device
- Keep in mind the cross-debug / profile nature

DroidFuzz Framework

- Existing fuzzing tools not covering campaign needs
  - Lack of reliable ARM support
  - Big integration effort for DEX file format
  - Small level of control in self-learning algos / config
    - Campaign has highly targeted nature in a complex ecosystem
- DroidFuzz framework has been created
  - Smart mutations for DEX based on set of rule-sets
  - Manual finite evolution of rule-sets
    - Code-coverage & hit counters as evaluation metrics
  - Most components designed to run efficiently on target device

Device Level Components

- Data generation
  - Mutate input corpus based on provided rule-set
  - Evaluate corpus for fitness for chosen rule-set
- Fuzzer core
  - Worker processes based on a fork() - exec() model
  - Crashes detected using POSIX signals (SIGSEGV, SIGBUS, etc.)
- Post-running helper tools
  - Crash Verifier: Crashes checked for acceptance ratio (>=60%)
  - Minimizer: Smallest subset of changes from original seed
  - ptrace & capstone to create crashing frame fingerprint

ART Signal Handlers

Posix signals fuzzing textbook: strace for custom handlers

```
b55a614925e15b6.dex --oat-file=foo.oat --no-watch-dog                    <
sigaltstack({ss_sp=0xb6ef5000, ss_flags=0, ss_size=8192}, NULL) = 0
sigaction(SIGABRT, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGBUS, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGFPE, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGILL, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGPIPE, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGSEGV, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGSTKFLT, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGTRAP, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}, NULL) = 0
sigaction(SIGSEGV, NULL, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|SA_ONSTACK, 0xb6f08bd8}) = 0
rt_sigprocmask(SIG_BLOCK, [QUIT USR1 PIPE], [], 8) = 0
sigaction(SIGSEGV, {0xb58b02e5, [], SA_RESTORER|SA_SIGINFO|SA_ONSTACK, 0xb46ccefc}, {0xb6ef8f49, [], SA_RESTORER|SA_RESTART|SA_SIGINFO|
SA_ONSTACK, 0xb6f08bd8}) = 0
Process 17676 attached
Process 17677 attached
[pid 17676] sigaltstack({ss_sp=0xb4501000, ss_flags=0, ss_size=8192} <unfinished ...>
[pid 17677] sigaltstack({ss_sp=0xb10dd000, ss_flags=0, ss_size=8192}, NULL) = 0
[pid 17676] <... sigaltstack resumed> , NULL) = 0
Process 17678 attached
[pid 17678] sigaltstack({ss_sp=0xb0f5b000, ss_flags=0, ss_size=8192}, NULL) = 0
[pid 17677] sigaltstack({ss_sp=0, ss_flags=SS_DISABLE, ss_size=8192}, NULL) = 0
[pid 17678] sigaltstack({ss_sp=0, ss_flags=SS_DISABLE, ss_size=8192}, NULL) = 0
[pid 17678] +++ exited with 0 +++
[pid 17677] +++ exited with 0 +++
[pid 17676] sigaltstack({ss_sp=0, ss_flags=SS_DISABLE, ss_size=8192}, NULL) = 0
[pid 17676] +++ exited with 0 +++
+++ exited with 0 +++
```

bionic debuggerd init

ART nested signals init

ART Signal Handlers

- art/runtime/fault_handler.cc
  - Special treat of SIGSEGV in ART generated native code
  - Sigchain handlers to support nested signals
    - Prevent signal masking when unwinding generated code
- Compiler fuzzing not affected
  - FaultManager::IsInGeneratedCode()
- Runtime execution (OAT) fuzzing might be affected, depending on fuzzing approach
- SIGQUIT, SIGUSR1, SIGPIPE, SIGABRT also have special handling by ART

Host Level Components

- AOSP build server (prod. & master branches)
  - ART gcov coverage builds (learning)
  - Default ART prod. settings builds (fuzzing)
  - ASAN debug (master only) builds (fuzzing & analysis)
- Crashes classifier
  - Remote GDB debugging with python scripting
  - Unique crashes signature hash
    - Frame fp: Num, function, relative-PC (using ProcFS), lib name
    - Major: 0-4 frame fps, Minor: 5-9 frame fps

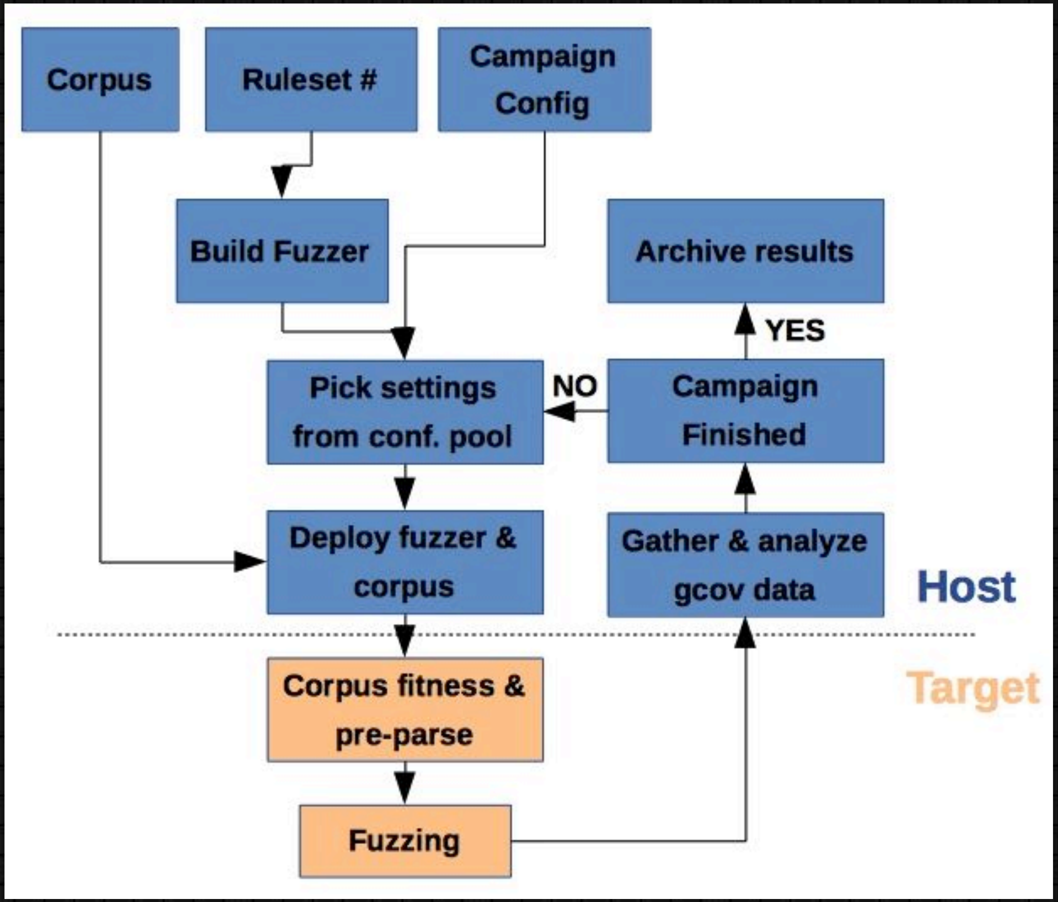
Major vs Minor Frame FPs

```
(gdb) bt
#0  art::HInstruction::ReplaceWith (this=this@entry=0xb5cf0300, other=0x0) at art/compiler/optimizing/nodes.h:518
#1  0xb6b5f1ce in art::SsaBuilder::VisitLoadLocal (this=<optimized out>, load=0xb5cf0300) at art/compiler/optimizing/ssa_builder.cc:133
#2  0xb6b55912 in art::HLoadLocal::Accept (this=<optimized out>, visitor=<optimized out>) at art/compiler/optimizing/nodes.cc:428
#3  0xb6b5f59c in art::SsaBuilder::VisitBasicBlock (this=0xbebe5bc0, block=0xb5cf0208) at art/compiler/optimizing/ssa_builder.cc:128
#4  0xb6b6006e in art::SsaBuilder::BuildSsa (this=this@entry=0xbebe5bc0) at art/compiler/optimizing/ssa_builder.cc:29
#5  0xb6b5609a in art::HGraph::TransformToSSA (this=this@entry=0xb5cf0000) at art/compiler/optimizing/nodes.cc:137
#6  0xb6b598f0 in art::OptimizingCompiler::TryCompile (this=this@entry=0xb5c37bf8, code_item=code_item@entry=0xb2d196c0, access_flags=acce
    invoke_type=invoke_type@entry=art::kDirect, class_def_idx=class_def_idx@entry=0, method_idx=method_idx@entry=11, class_loader=class_lo
    dex_file=...) at art/compiler/optimizing/optimizing_compiler.cc:132
#7  0xb6bb1090 in art::OptimizingCompiler::Compile (this=0xb5c37bf8, code_item=0xb2d196c0, access_flags=65537, invoke_type=art::kDirect, c
    method_idx=11, class_loader=0x10007a, dex_file=...) at art/compiler/compilers.cc:150
#8  0xb6b2f6ce in art::CompilerDriver::CompileMethod (this=this@entry=0xb5c3cb00, code_item=0xb2d196c0, access_flags=65537, invoke_type=ar
    class_def_idx=class_def_idx@entry=0, method_idx=method_idx@entry=11, class_loader=class_loader@entry=0x10007a, dex_file=...,
    dex_to_dex_compilation_level=dex_to_dex_compilation_level@entry=art::kRequired, compilation_enabled=compilation_enabled@entry=true)
    at art/compiler/driver/compiler_driver.cc:2123
#9  0xb6b2ff20 in art::CompilerDriver::CompileClass (manager=<optimized out>, class_def_index=<optimized out>) at art/compiler/driver/comp
#10 0xb6b2525e in art::ParallelCompilationManager::ForAllClosure::Run (this=0xb5c3b390, self=0xb5c27400) at art/compiler/driver/compiler_c
#11 0xb6e45c18 in art::ThreadPool::Wait (this=0xb5cc1400, self=self@entry=0xb5c27400, do_work=do_work@entry=true, may_hold_locks=may_hold_
    at art/runtime/thread_pool.cc:182
```

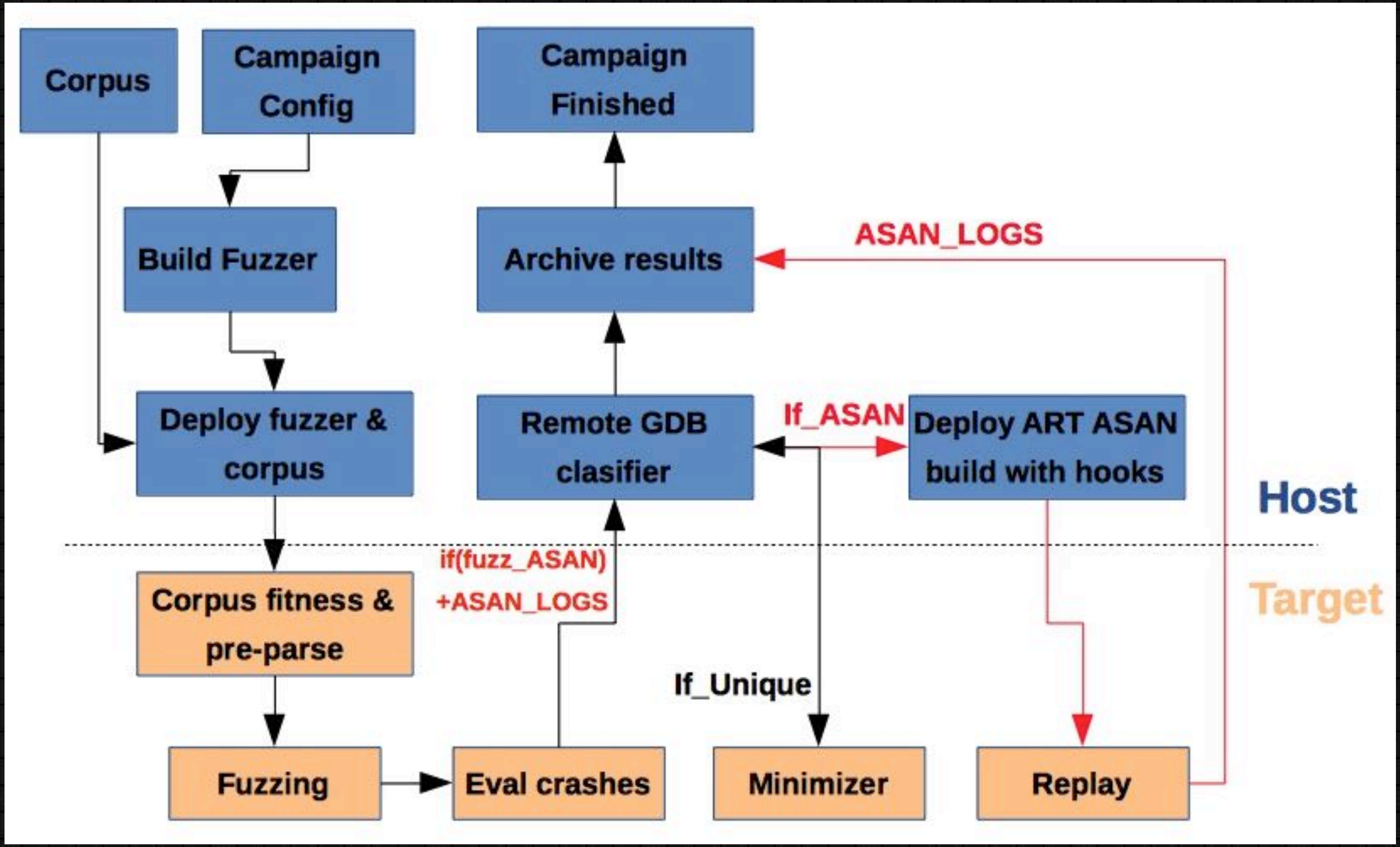
Major vs Minor Frame FPs

```
97          return ResourceMask(masks_[0] | other.masks_[0], masks_[1] | other.masks_[1]);
(gdb) bt
#0  0xb6b06cfc in Union (other=..., this=0xb335d778) at art/compiler/dex/quick/resource_mask.h:97
#1  SetupRegMask (reg=394, mask=0xb335d778, this=0xb83b7c58) at art/compiler/dex/quick/mir_to_lir-inl.h:146
#2  SetupResourceMasks (lir=0xb3156810, this=0xb83b7c58) at art/compiler/dex/quick/mir_to_lir-inl.h:212
#3  RawLIR (target=0x0, op4=0, op3=0, op2=<optimized out>, op1=<optimized out>, op0=<optimized out>, opcode=<optimized out>, dalvik_o
    this=0xb83b7c58) at art/compiler/dex/quick/mir_to_lir-inl.h:55
#4  NewLIR3 (src2=<optimized out>, src1=<optimized out>, dest=<optimized out>, opcode=<optimized out>, this=0xb83b7c58)
    at art/compiler/dex/quick/mir_to_lir-inl.h:114
#5  art::ArmMir2Lir::LoadBaseDispBody (this=0xb83b7c58, r_base=..., displacement=72, r_dest=..., size=art::kReference)
    at art/compiler/dex/quick/arm/utility_arm.cc:947
#6  0xb6af5ab0 in art::Mir2Lir::LoadRefDisp (this=<optimized out>, r_base=..., displacement=<optimized out>, r_dest=..., is_volatile=
    at art/compiler/dex/quick/mir_to_lir.h:1005
#7  0xb6b31da4 in art::Mir2Lir::GenIGet (this=this@entry=0xb83b7c58, mir=mir@entry=0xb314ec78, opt_flags=opt_flags@entry=0, size=size
    rl_dest=..., rl_obj=..., is_long_or_double=is_long_or_double@entry=false, is_object=is_object@entry=true) at art/compiler/dex/qui
#8  0xb6b5971c in art::Mir2Lir::CompileDalvikInstruction (this=this@entry=0xb83b7c58, mir=mir@entry=0xb314ec78, bb=bb@entry=0xb314eb8
    at art/compiler/dex/quick/mir_to_lir.cc:733
#9  0xb6b5a3ae in art::Mir2Lir::MethodBlockCodeGen (this=this@entry=0xb83b7c58, bb=bb@entry=0xb314eb80) at art/compiler/dex/quick/mir
#10 0xb6b5a53e in art::Mir2Lir::MethodMIR2LIR (this=0xb83b7c58) at art/compiler/dex/quick/mir_to_lir.cc:1248
#11 0xb6b2aeb8 in art::Mir2Lir::Materialize (this=0xb83b7c58) at art/compiler/dex/quick/codegen_util.cc:1044
#12 0xb6b97760 in art::CompileMethod (driver=..., compiler=0xb806e948, code_item=<optimized out>, access_flags=0, invoke_type=art::kV
    class_def_idx=class_def_idx@entry=258, method_idx=method_idx@entry=2344, class_loader=class_loader@entry=0x10007a, dex_file=...,
    llvm_compilation_unit=llvm_compilation_unit@entry=0x0) at art/compiler/dex/frontend.cc:776
```

Learning Phase



Execution Phase



Rule-sets Evolution

Rule-sets Level-1

- Create rule for every basic section
  - 16 in total (header & mapList are excluded)
  - Verify that input seeds contain examined section (fitness)
  - Random mutations within the section range
- Need to extract (fast) section ranges
  - Some exist in header (stringIds), some not (codeItems)
- Use DEX mapList
  - Entries contain start Off & Size in items
    - Benefit for ordering to avoid size calculations (end == next_start)
  - Pre-parse & store data for all input seeds
    - Workers fast resolve due to fork() model

DEX mapList Entries

▶ struct class_def_item_list dex_class_defs	12906 classes
▼ struct map_list_type dex_map_list	18 items
uint size	12h
▼ struct map_item list[18]	
▶ struct map_item list[0]	TYPE_HEADER_ITEM
▶ struct map_item list[1]	TYPE_STRING_ID_ITEM
▶ struct map_item list[2]	TYPE_TYPE_ID_ITEM
▶ struct map_item list[3]	TYPE_PROTO_ID_ITEM
▶ struct map_item list[4]	TYPE_FIELD_ID_ITEM
▶ struct map_item list[5]	TYPE_METHOD_ID_ITEM
▶ struct map_item list[6]	TYPE_CLASS_DEF_ITEM
▶ struct map_item list[7]	TYPE_ANNOTATION_SET_REF_LIST
▶ struct map_item list[8]	TYPE_ANNOTATION_SET_ITEM
▼ struct map_item list[9]	TYPE_CODE_ITEM
enum TYPE_CODES type	TYPE_CODE_ITEM (2001h)
ushort unused	0h
uint size	C832h
uint offset	1CEA70h
▶ struct map_item list[10]	TYPE_ANNOTATIONS_DIRECTORY_ITEM
▶ struct map_item list[11]	TYPE_TYPE_LIST
▶ struct map_item list[12]	TYPE_STRING_DATA_ITEM
▶ struct map_item list[13]	TYPE_DEBUG_INFO_ITEM
▶ struct map_item list[14]	TYPE_ANNOTATION_ITEM
▶ struct map_item list[15]	TYPE_ENCODED_ARRAY_ITEM
▶ struct map_item list[16]	TYPE_CLASS_DATA_ITEM
▶ struct map_item list[17]	TYPE_MAP_LIST

Learning Phase1 Results

	Quick			Optimizing		
Ruleset	**Lines**	**Functions**	**Branches**	**Lines**	**Functions**	**Branches**
Original Seeds	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%

Code Coverage for 5K iterations / rule

Learning Phase1 Results

	Quick			Optimizing		
Ruleset	**Level1**	**Level2**		**Level1**	**Level2**	
	PASSED	**HARD FAIL**	**SOFT FAIL**	**PASSED**	**HARD FAIL**	**SOFT FAIL**
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%

DEX verification success ratio for 5K iterations / rule

Phase1 Observations

- Best results from sections with Data type items
  - codeItems, debugInfo, encodedArray, annotationItems
- Bad results from sections with Index and/or Offset type items
  - stringIdItems, typeIdItems, methodIdItems
- Avg. results from sections with mixed type items
  - classDefItems
- Failed so far with annotation related sections

Phase1 Observations

- Locating less valuable targets (priority = low)
  - debugInfo: Are not parsed by the OAT compiler
    - Used by debugger & ELFWriter if "—include-debug-symbols"
  - encodedArrayItems: values to initialize static fields
    - Invoked during class initialization
    - CompilerDriver initialize classes, although not directly affecting compilation parameters
  - Strings must be explicitly sorted
    - Fuzzing stringIds & stringData items requires re-sorting
    - Noticeable performance overhead

Designing Learning Phase2

- Need to improve verifier success ratio
  - Upgrade rule intelligence
  - For section items with members of type:
    - Index: In-range mutation of IDXs of matching reference type
    - Offset: In-range mutation for referencing data section
    - Metadata: Create enumeration pools of valid data for each type
- Introduce structural mutations for data items
  - Instructions inside code_items
  - Class data encoded_method, encoded_field, etc.

Designing Learning Phase2

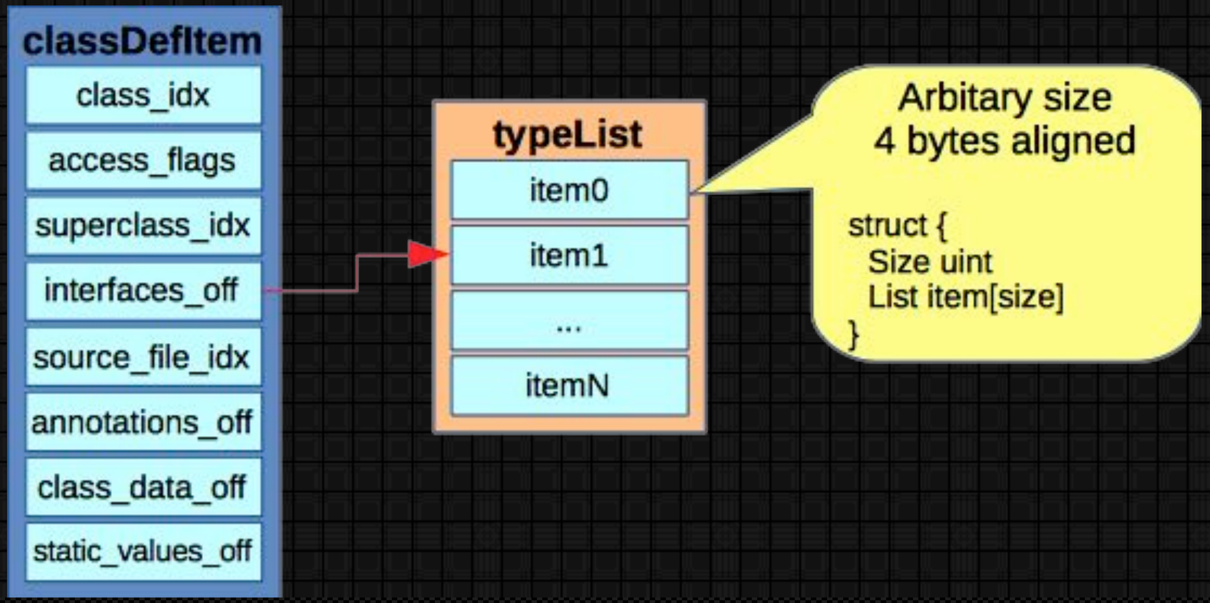
- Focus on code_items for maximum compiler stressing
- Dedicated rules for code_items fuzzing
  - Random fuzzing within instructions range
  - Modify instructions opcode
  - Shuffle instructions within code_item
  - Modify branches offset
  - Modify register numbers

Designing Learning Phase2

- Accurate ways to detect verifier L2 hard fails ratio
  - Use single class corpus
    - Code-coverage elitism (Top-500) of split original corpus
    - Class / Method not found error treated as soft
  - Campaign's L2 hard hit counter will reflect rejection % for rule
- Backwards chain basic rules across sections
  - Force mangled Data items picked always by some Off
  - Force mangled Off items picked always by some Idx
  - Attempt to examine mangled blobs under more contexts
  - Less performance cost in case of seeds with small #classes

Off + Data Mangle Challenges

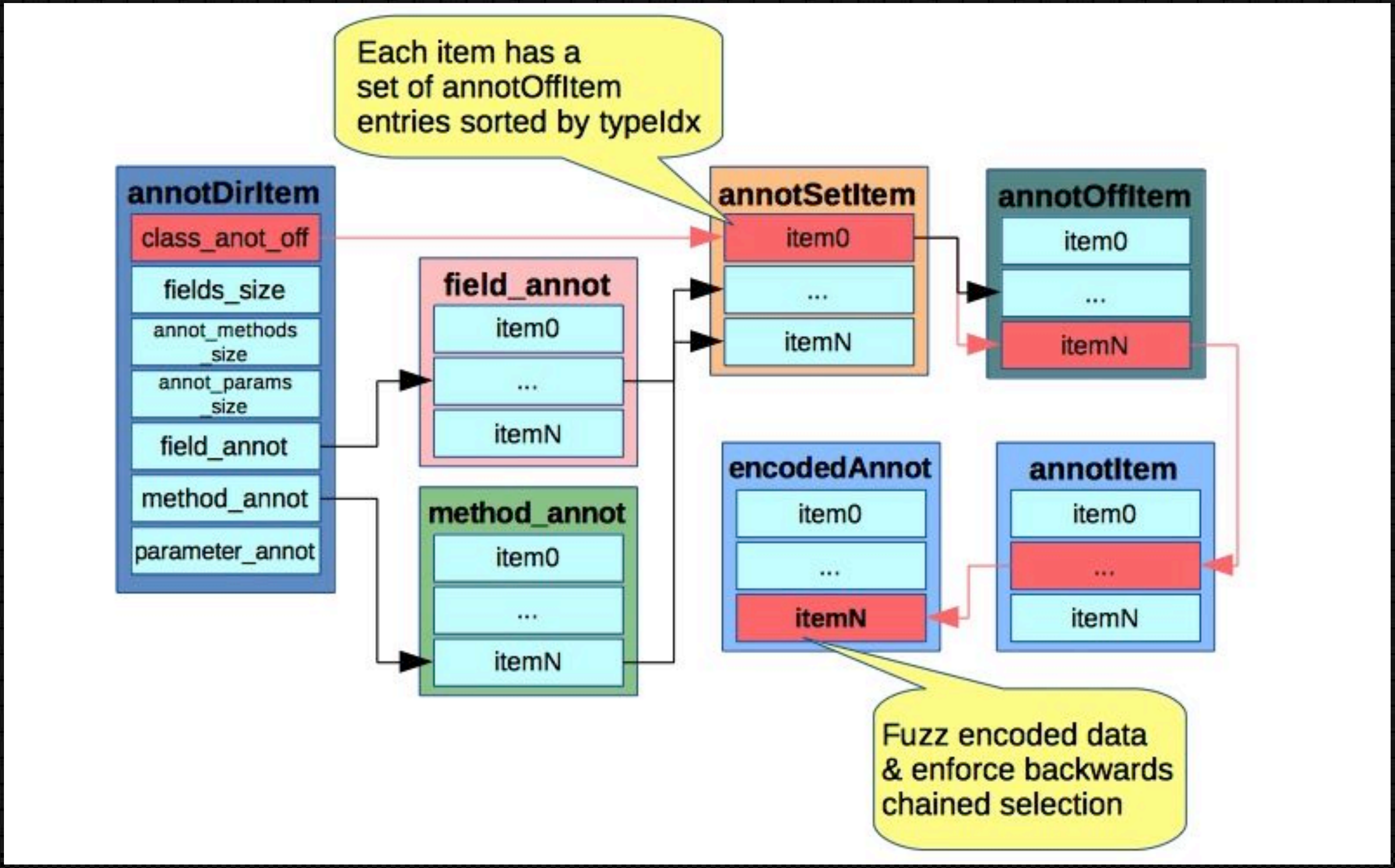
- In-range mutation will most likely fail validation if not pointing at the beggining of encoded item
- Items in data sections follow strict structural rules



Off + Data Mangle Challenges

- During seeds init phase (pre-parsing):
  - Calculate the number of items (count) in each data section
    - Specially for code items extract total number of Instrs / file
    - Better random distribution when instr fuzzing
  - Store at seeds metadata
- When fuzzing, for each worker process:
  - RNG uses target data section items count instead of size
  - Picked item IDs are sorted & passed to mangle routine
  - Mangle routine scans once applying mutations for marked items

Chaining Rules



Learning Phase2 Results

- Skipped evolution for less-valuable targets
- Elitist evolution of top5 rules from phase1
- Improved & chained annotation rules

	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%

VFY-L1 success ratio for 5K iter. / rule-group (random inner)

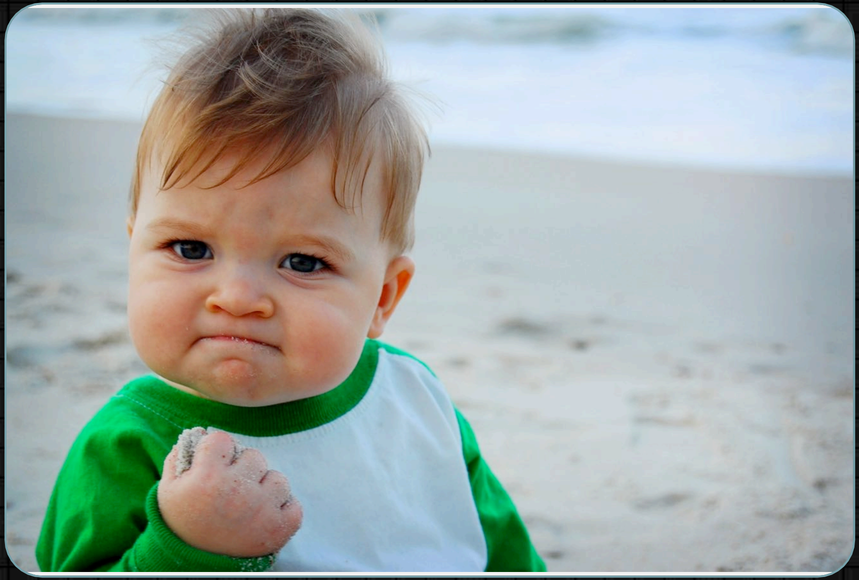
Learning Phase2 Results

- Verifier level2 hard fails are more accurately analyzed through single class seeds

Ruleset	Quick			Optimizing		
	Phase1		Phase2	Phase1		Phase2
	MultiClass	SingleClass	SingleClass	MultiClass	SingleClass	SingleClass
protoIdItems	0.00%	0.00%	0.49%	0.00%	0.00%	0.91%
fieldIdItems	0.06%	22.74%	38.30%	0.06%	17.99%	35.90%
methodIdItems	0.32%	22.85%	47.76%	0.33%	24.64%	45.12%
classDefItems	0.02%	0.74%	15.98%	0.02%	0.70%	17.38%
codeItems	1.11%	86.56%	15.60%	1.08%	86.33%	15.34%
annotations_chain	-	-	1.30%	-	-	1.68%

VFY-L2 hard fail ratio for 5K single class iter. / rule-group (random inner)

Fuzzing Results



OS Versions

- Android 5.1.x Release Build
  - Nexus4, Nexus5, Nexus6
- ART master branch #8e8bb8a (April 16, 2015)
  - Nexus 5, Nexus 6
  - Coverage & ASAN builds using same commit
- Device specific crash triggers
  - Nexus 4 vs Nexus 5/6
    - Different base libc allocator (dlmalloc vs jemalloc)
  - Nexus 5 (2GB RAM) vs Nexus 6 (3GB RAM)
    - Small differences in heap layout affecting fps (non-ASAN only)

5.1.x Unique Crashes

- OPTIMIZING crashes not including QUICK
  - Compiler failover increases analysis effort
  - Many QUICK bugs discovered via OPTIMIZING fuzzing
- Need reliable way to avoid backend failover

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

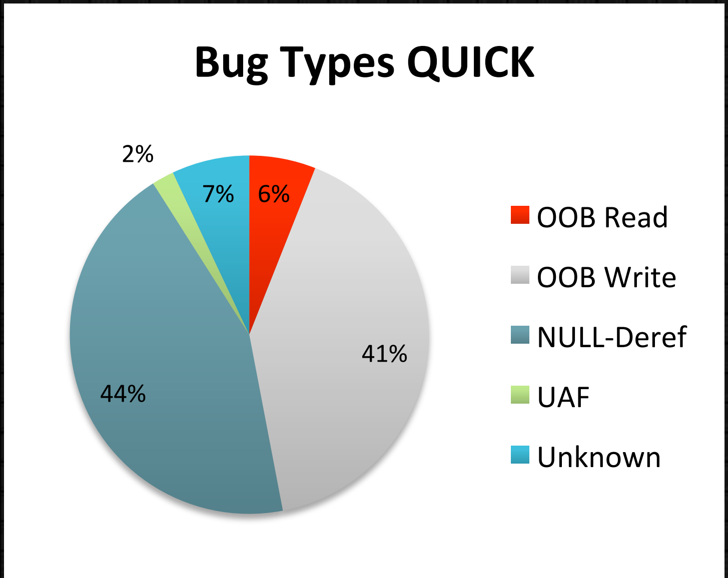
Master Unique Crashes

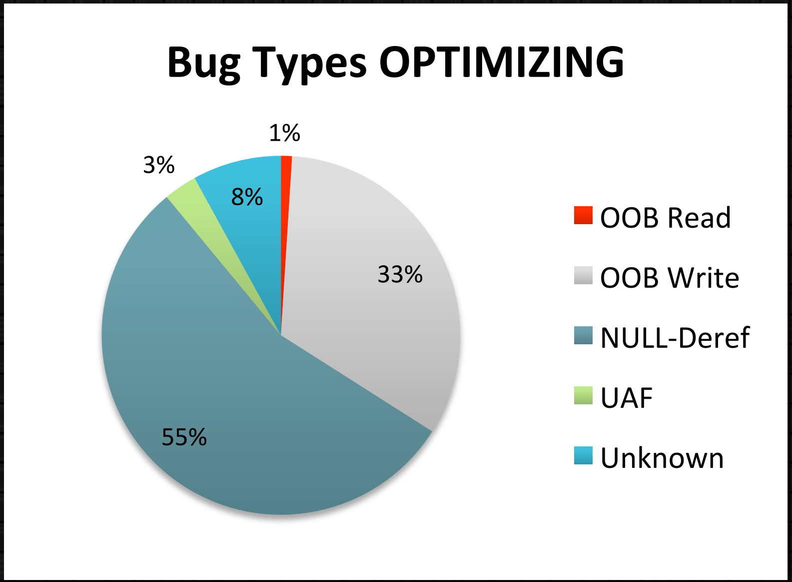
- Not all 5.1.x bugs are triggered in master (possibly fixed)
- ASAN crashes additional to non-ASAN master target
- OPTIMIZING crashes not including QUICK
  - Compiler failover increases analysis effort
- Increased # of bugs outside "art/compiler"

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

General Statistics

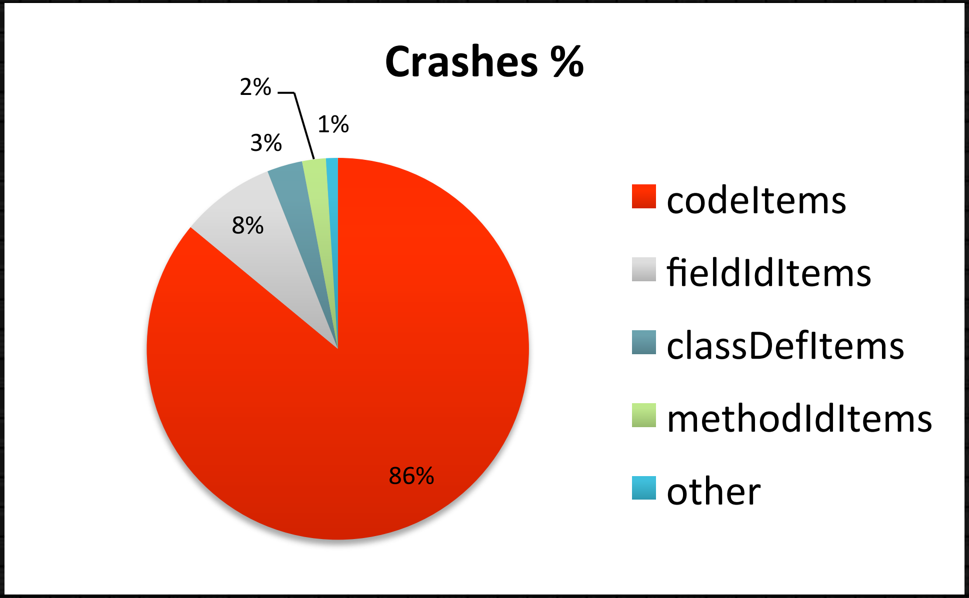
- From instrumentation & manual analysis
- Includes both 5.1.x & master





General Statistics

- Discovered crashes % for current DEX rules
- Other includes chained rules



Summary

- ART is a very complex component
  - Requires security testing from many angles
  - Large number of execution / configuration parameters
- Mutation rules evolution
  - Level1: Honor range of DEX basic sections
  - Level2: Honor structural dependencies of item indexes
  - Combine level2 rules into more complex chains
- Feedback evolution must consider DEX verifier success / fail results

Next Steps

- Analyze discovered bugs
  - Non-interesting cases will find their way to report
- DEX fuzzer optimizations
  - Improve rule chains intelligence
  - Annotations have been poorly covered
- Cont. with Runtime init & exec fuzzing phases
  - Prototype ELF OAT fuzzer using Melkor under alpha state
  - ART Image file format fuzzer under development

Next Steps

- Fuzzing framework optimizations
  - libbacktrace / libunwind integration for real-time unique crashes
  - Performance improvements (e.g. pre-fork server with Runtime initialized). You know the cool stuff lcamtuf blogs about.
- Examine alternative fuzzing techniques – Improve feedback analysis automation
- Better integration with ASAN & other instr. tools
  - Hopefully AOSP will start supporting ASAN for ART
  - Examine clang SanitizerCoverage as a faster gcov alternative

References

- Matteo Franchin - ART's Quick Compiler: an unofficial overview
- Tavis Ormandy – Making Software Dumber
- 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
- Michał Zalewski - AFL Fuzzer:
http://lcamtuf.coredump.cx/afl/technical_details.txt
- LLVM LibFuzzer:
http://llvm.org/docs/LibFuzzer.html
- Alejandro Hernández - Melkor ELF Fuzzer:
https://github.com/IOActive/Melkor_ELF_Fuzzer

Questions?