Android Native Library Fuzzing

Paolo Celada

February - July, 2022

Politecnico di Torino

hexhive

EPFL

Agenda

Dynamic analysis of native components in Android applications

- Introduction
- Previous Approaches
- AFL++ in Android
- Harness
- Android Native Fuzzing Framework
- Results
- Future Work

Introduction

Android, NDK, JNI and fuzz testing

Introduction

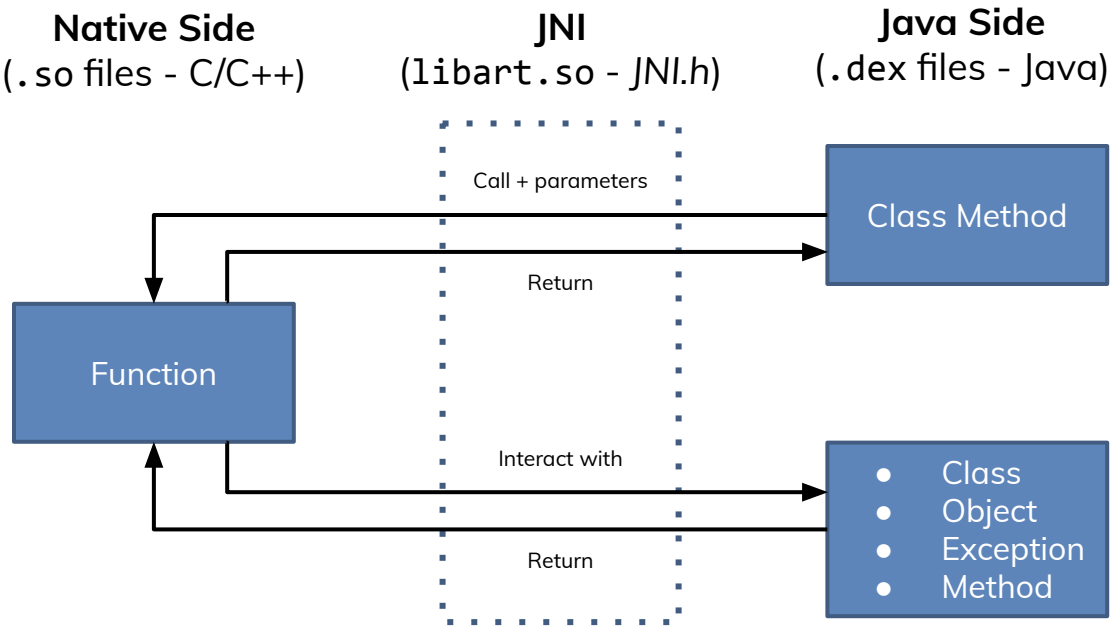
- Android is the most popular OS for mobile devices
  - 2.8 billions users
  - 4000+ devices
  - 18 billions applications
- Applications are developed with the SDK in Java/Kotlin, and the NDK in C, C++ (Native) → **37.2%** [14]
- **Java is “secure”:**
  - JVM/ART usage
  - Pointer removal
  - Memory safety
  - Exception handling
- **Native with Java is:**
  - **Fast:** android limited HW
  - **Convenient:** native library reuse
  - **But… highly insecure:**
    - memory/temporal safety violations
    - format string vulnerabilities
    - type confusion
    - CWE-111 (*direct use of unsafe JNI*)

  → **Testing is key (fuzzing)**
- **How?**

  The **Java Native Interface** (JNI)

The Java Native Interface (JNI)

A “foreign function interface programming framework”



Usage

From Java to Native

Load library, declare methods as ***native*** and interact using JNI functions

(Example: `void native_method(JNIEnv *env, jobject obj, ...args...)`)

From Native to Java

Create and load JVM and interact with it using JNI functions

`JNI.h`: Types and Functions

- **JNI Types:**
  - Primitive types (`int` → `jint`, `float` → `jfloat`, ...)
  - Reference types: array, classes, instances handled via `JNIEnv` functions only
- **JNINativeInterface (`JNIEnv`):**
  - Type conversions (`GetStringUTFChars`, `NewStringUTF`, `GetIntArrayRegion`, ...)
  - Java methods calls (`GetMethodID`, `CallCharMethod`, ...)
  - Object interactions (`NewObject`, `GetIntField`, `CallStaticObjectMethod`, ...)
  - Exception handling (`ThrowNew`, ...)
- **JNIInvokeInterface(`JavaVM`):**
  - VM-related operations (`GetEnv`, `DestroyJavaVM`, `AttachCurrentThread`, ...)

Native Functions Naming Conventions

Pattern-defined	Dynamically-defined
• **Generated by *javac*** • **Construction pattern** • **Symbols exported** **Example:** `Java_com_name_jni_package_JNI_nativeMethod`	• Manually registered by developer, dynamically linked • **Preferred name** • **Construction steps** 1. **`JNI_OnLoad`** 2. `JNINativeMethod` struct 3. **`RegisterNatives`** • **Only `JNI_OnLoad` exported**

JNI Problem Evaluation

- Downloaded all F-droid applications (open source) → 250GB
- 3832 apps (not counting versions)
- 340 apps with native

All use the JNI (to some extent)

Note:
`void native_method(JNIEnv *env, jobject obj, ...args...)`

JNI Method	# Apps using it
GetStringUTFChars	238
FindClass	231
NewStringUTF	207
ReleaseStringUTFChars	201
GetMethodID	186
GetArrayLength	179
DeleteLocalRef	167
GetFieldID	145
NewGlobalRef	143
GetObjectClass	137
NewObject	127
GetStaticMethodID	123
NewByteArray	118
CallObjectMethod	116
DeleteGlobalRef	113
NewIntArray	113
CallVoidMethod	111
SetObjectArrayElement	110
NewObjectArray	109

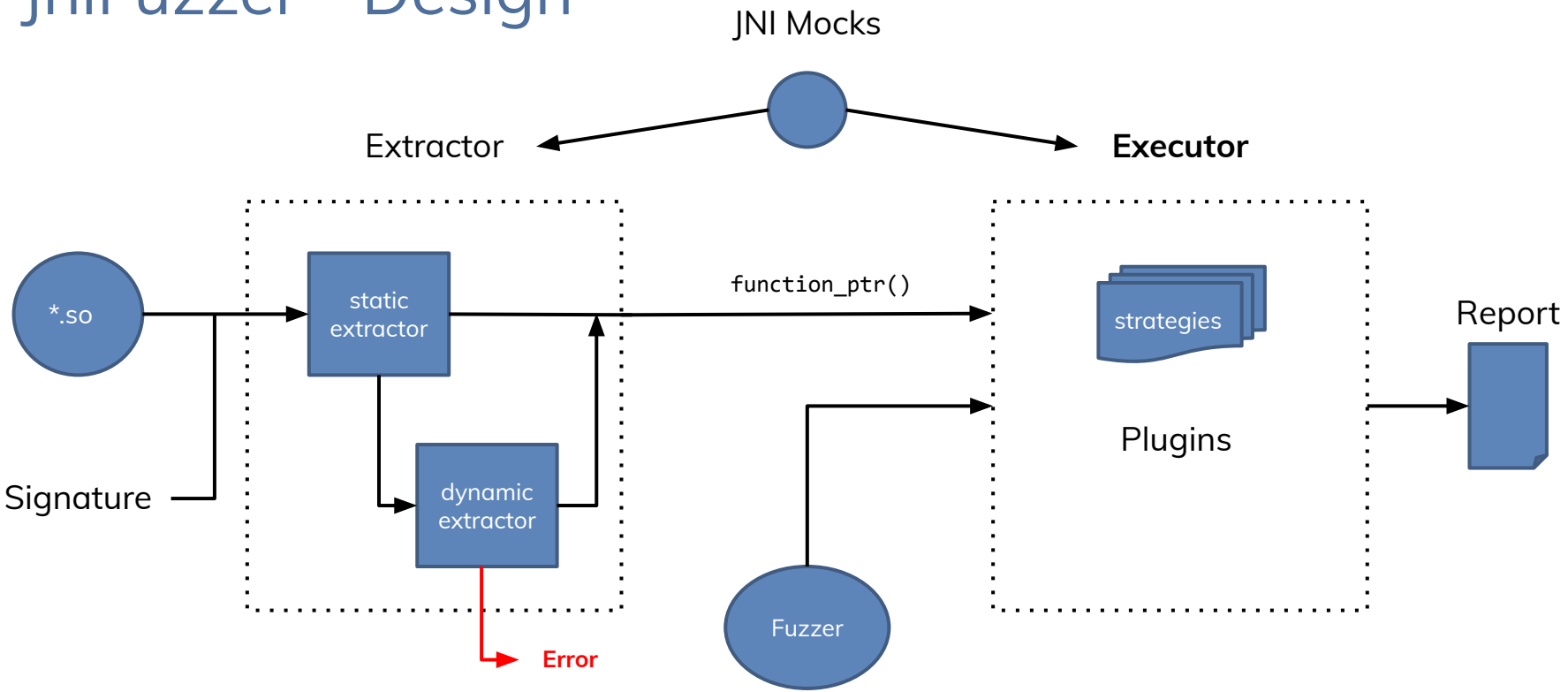
Previous Approaches

JniFuzzer

Previous Approaches

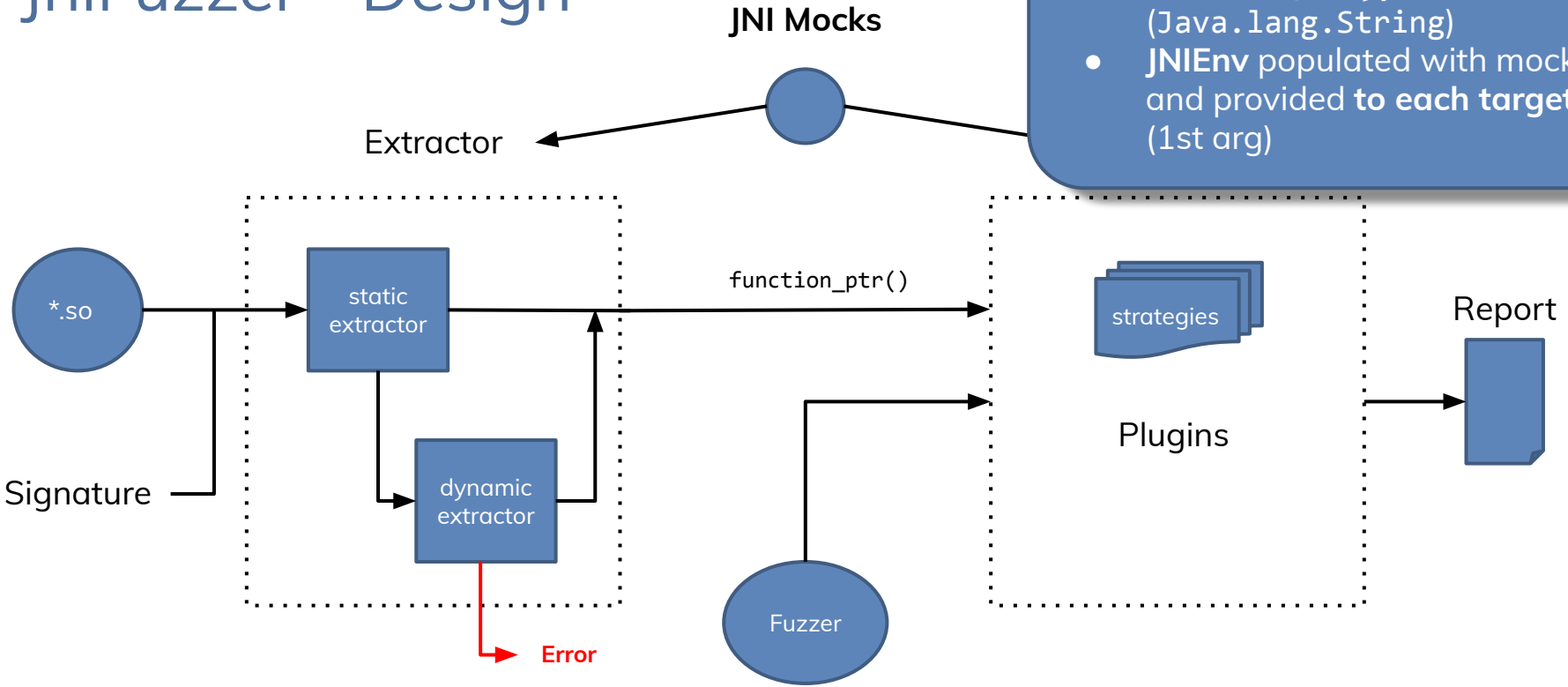
Target Java - Static	Target Java - Dynamic
• Droidsafe [1] • Flowdroid [2] • ICCTA [3] • Chex [4] • Amandroid [5]	• Memory Leak Fuzzer [6] • DroidFuzzer [7] • Non-Crashing Logic Bugs Fuzzer [8]
Target Native - Static	**Target Native - Dynamic**
• StubDroid [9] • JuCify [10] • Native-to-Java Callback Analysis [11]	• JniFuzzer [12]

JniFuzzer - Design

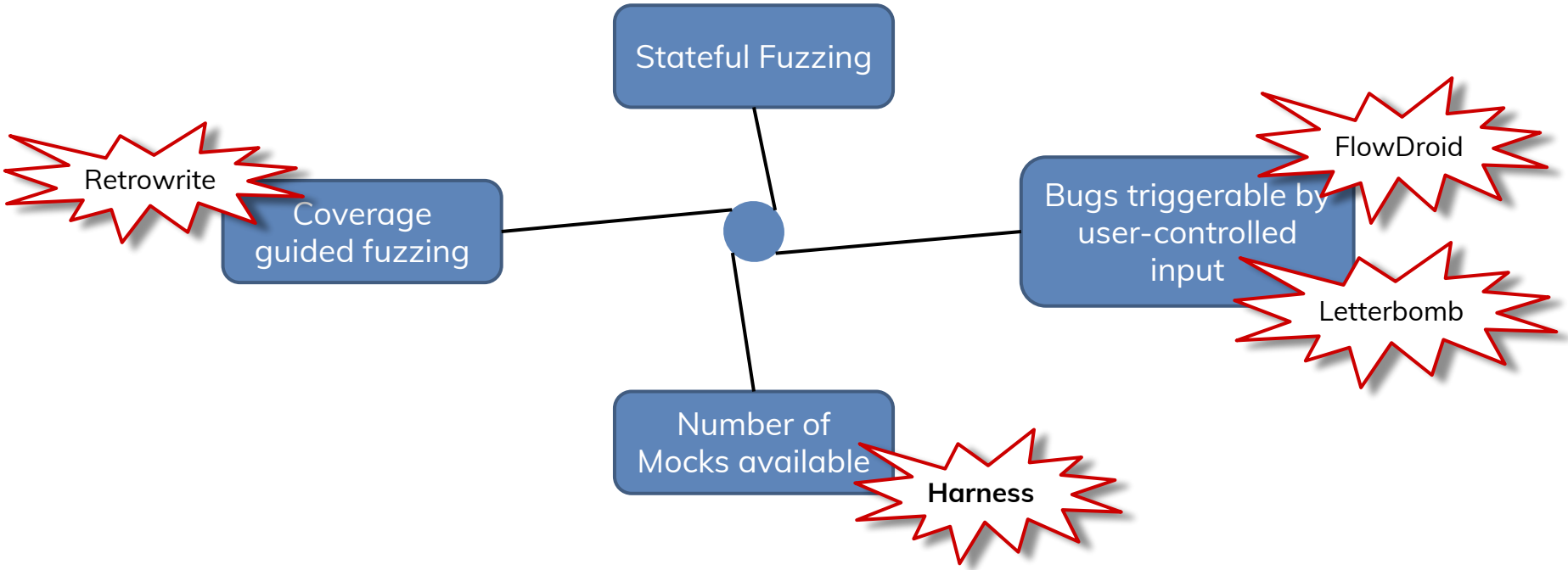


JniFuzzer - Design

- **Mocks of JNI functions** (subset)
- **Mocks of JNI types** (`Java.lang.String`)
- **JNIEnv** populated with mocks and provided **to each target** (1st arg)



JniFuzzer - Limitations



AFL++ in Android

Patches and Installation

AFL++ Build

Prerequisites

- Rooted device
- Termux
- Packages: *make*, *libandroid-shmem* and *ndk-sysroot*
- *Clang-v13* (install from .deb files, not termux)

AFL++ Build

Prerequisites

- Rooted device
- Termux
- Packages: *make*, *libandroid-shmem* and *ndk-sysroot*
- *Clang-v13* (install from .deb files, not termux)

Patches*

- **POSIX compliance issue** (in `src/afl-ld-lto.c`): `index()` function unavailable in Bionic libc → `strchr()`
- **LLVM symbols**: needed by AFL++ compiler pass → `LD_PRELOAD libLLVM-13.so`
- **Afl-cc symbols**: missing compiler flag to `afl-compiler-rt.o` required to link its symbols
- **MMAP symbols**: Bionic doesn't have MMAP symbols → force usage of *shm* for shared memory op.
- **Compilation test**: use `--afl-CLASSIC` mode to pass it (not strictly required)
- **Bug in AFL++**: found with ASAN, prevent regular AFL++ startup due to wrong length passed to *memchr*

*Relative to AFL++ release 4.00c of January 26th. 2022

AFL++ Build and Performance

Prerequisites

- Rooted device
- Termux
- Packages: *make*, *libandroid-shmem* and *ndk-sysroot*
- *Clang-v13* (install from .deb files, not termux)

Patches*

- **POSIX compliance issue**
- **LLVM symbols**
- **Afl-cc symbols**
- **MMAP symbols**
- **Compilation test**
- **Bug in AFL++**

Performance

On a Google Pixel 4:

- Octa-core Snapdragon 855 Qualcomm
- Most perf. core (2.84GHz)
- AFL++ test-perf.

2900 exec/sec

*Relative to AFL++ release 4.00c of January 26th. 2022

Harness

Design and Usage

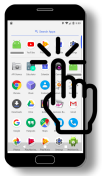
Key Idea / Inspiration

How does Android start an app?

Boot device

Start Zygote process

- preloads potentially needed resources
- efficient app startup
- listens to socket for incoming requests



User clicks on app icon

ActivityManagerService

Zygote socket

Fork Zygote state

- **provide to the app an already initialized VM**
- Dalvik VM or ART
- *`Zygote.forkAndSpecialize()`*

App starts

Key Idea / Inspiration

How does Android start an app?

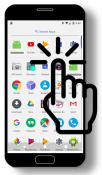
Problem

- **Android apps** are highly **multithreaded**
- **Fork propagates only main thread**
- Forking app state results in SIGSEGV when invoking Java code
- **This doesn't happen when loading a custom JVM**

Boot device

Start Zygote process

- preloads potentially needed resources
- efficient app startup
- listens to socket for incoming requests



User clicks on app icon

ActivityManagerService

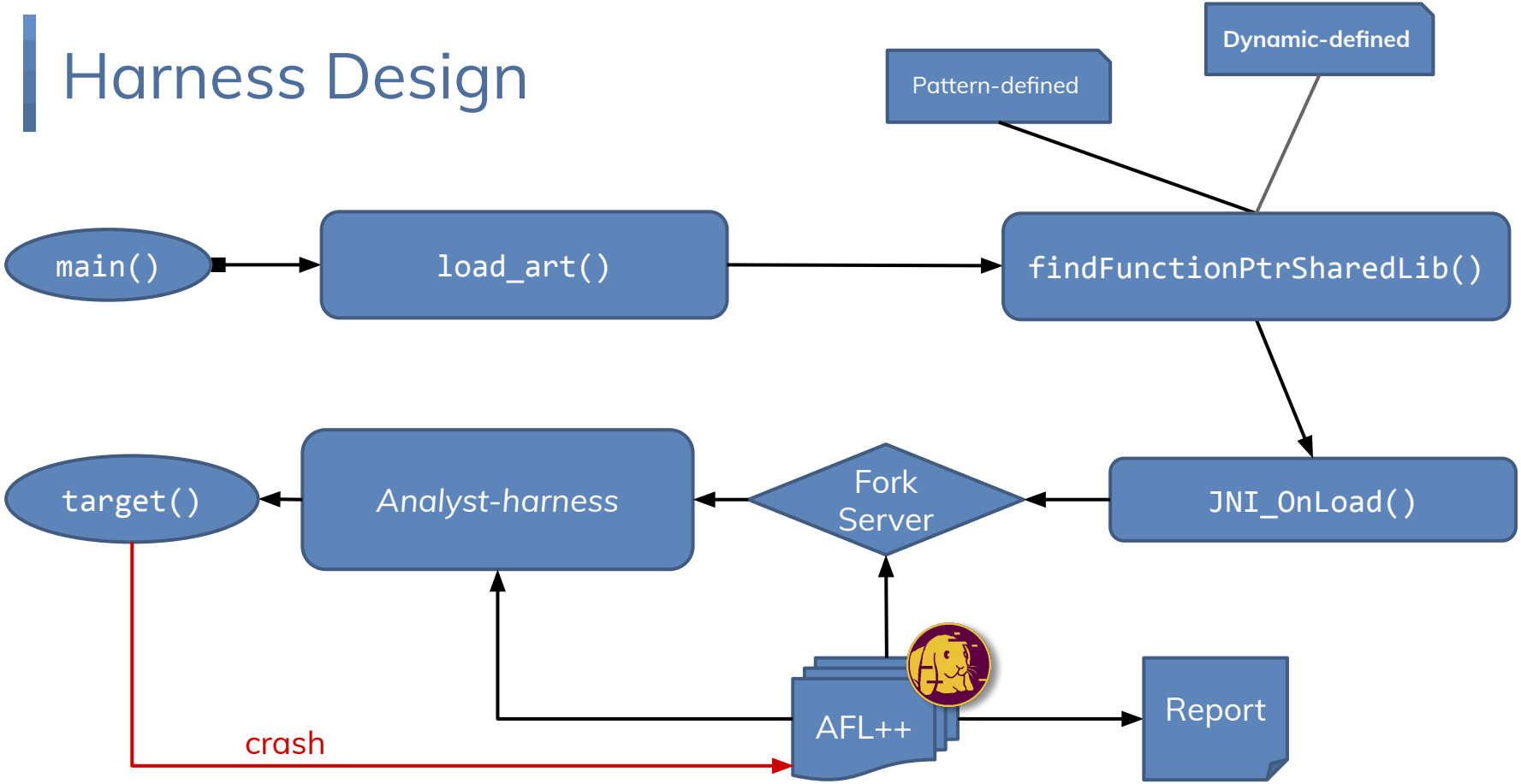
Zygote socket

Fork Zygote state

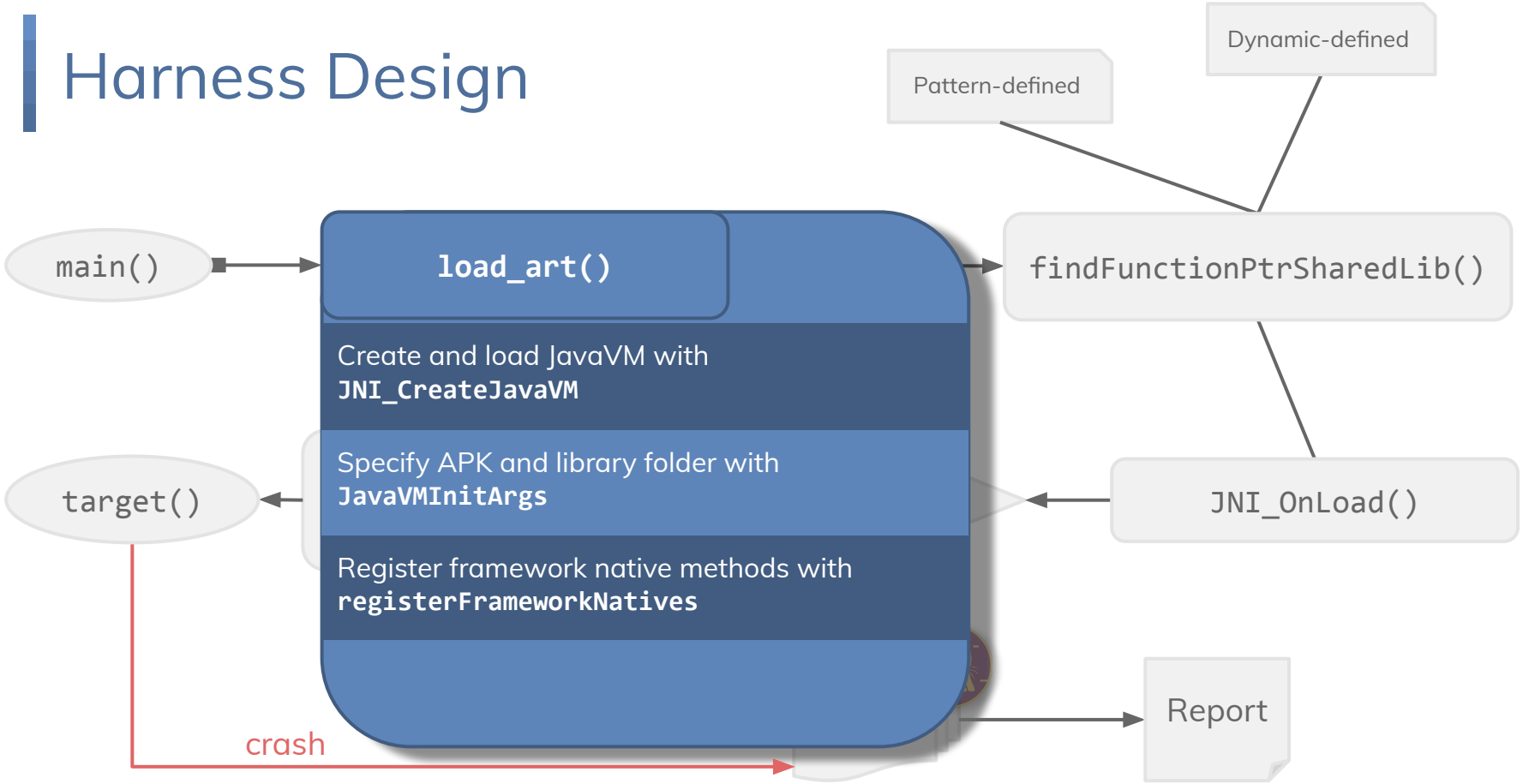
- **provide to the app an already initialized VM**
- Dalvik VM or ART
- *Zygote.forkAndSpecialize()*

App starts

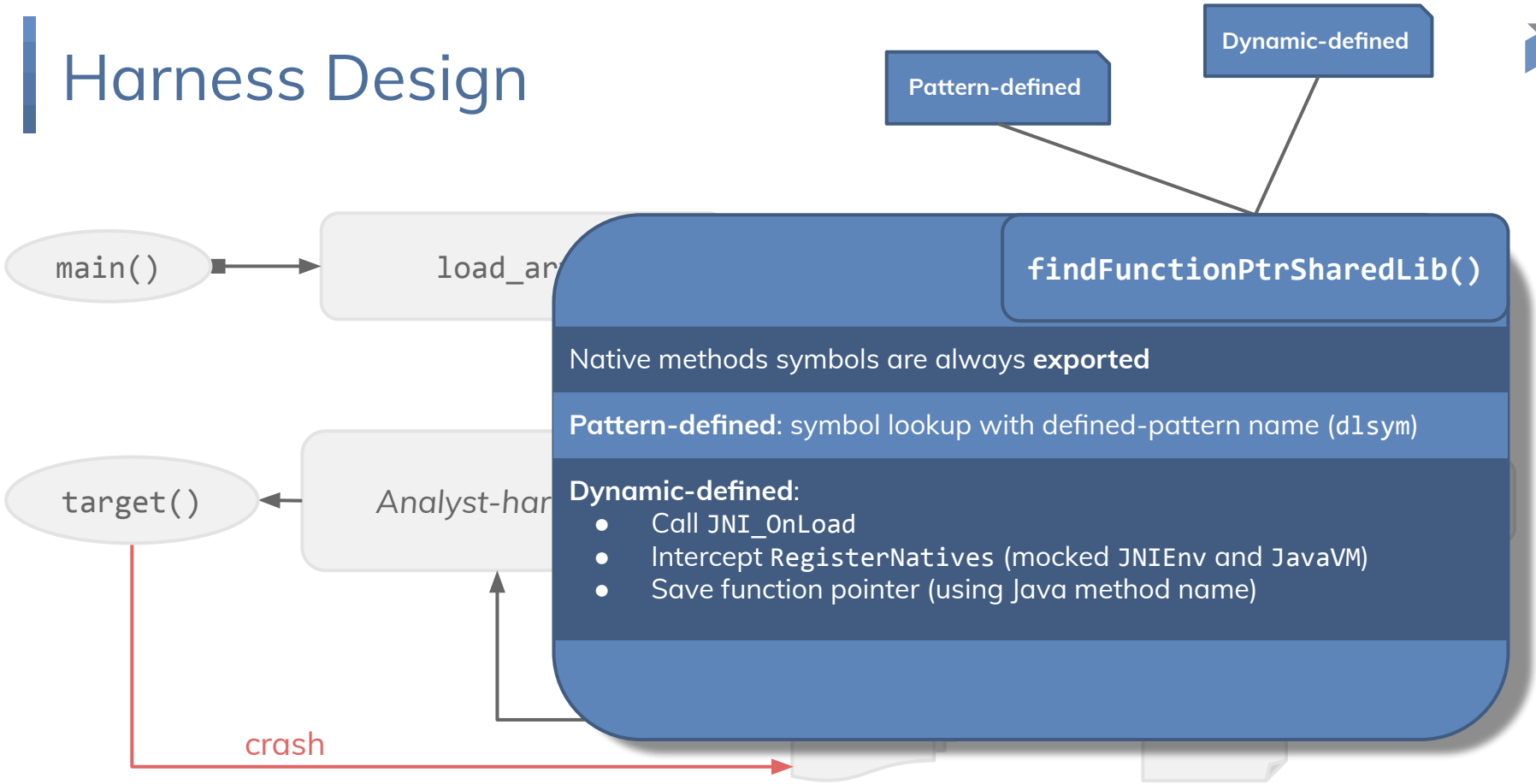
Harness Design



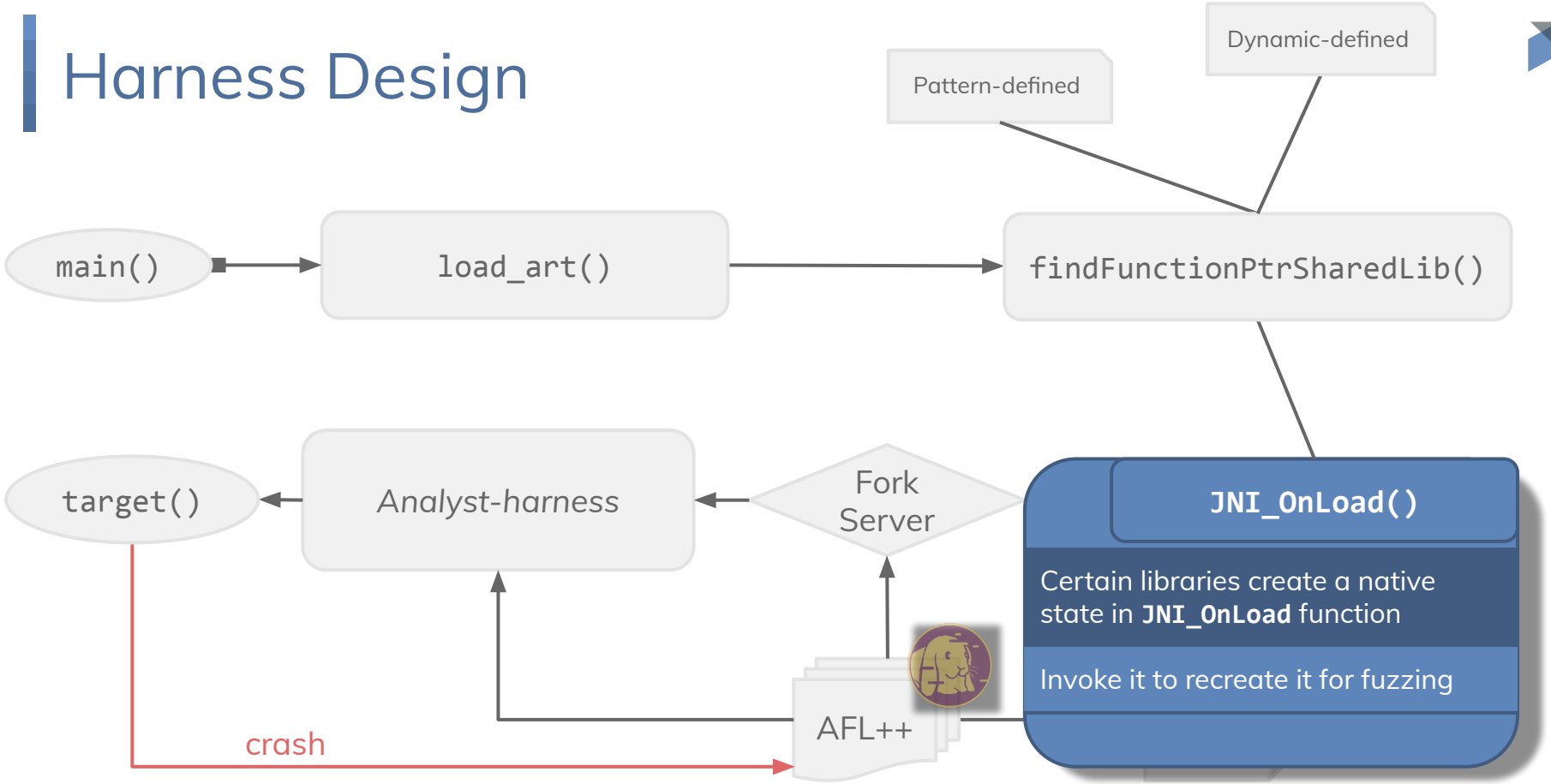
Harness Design



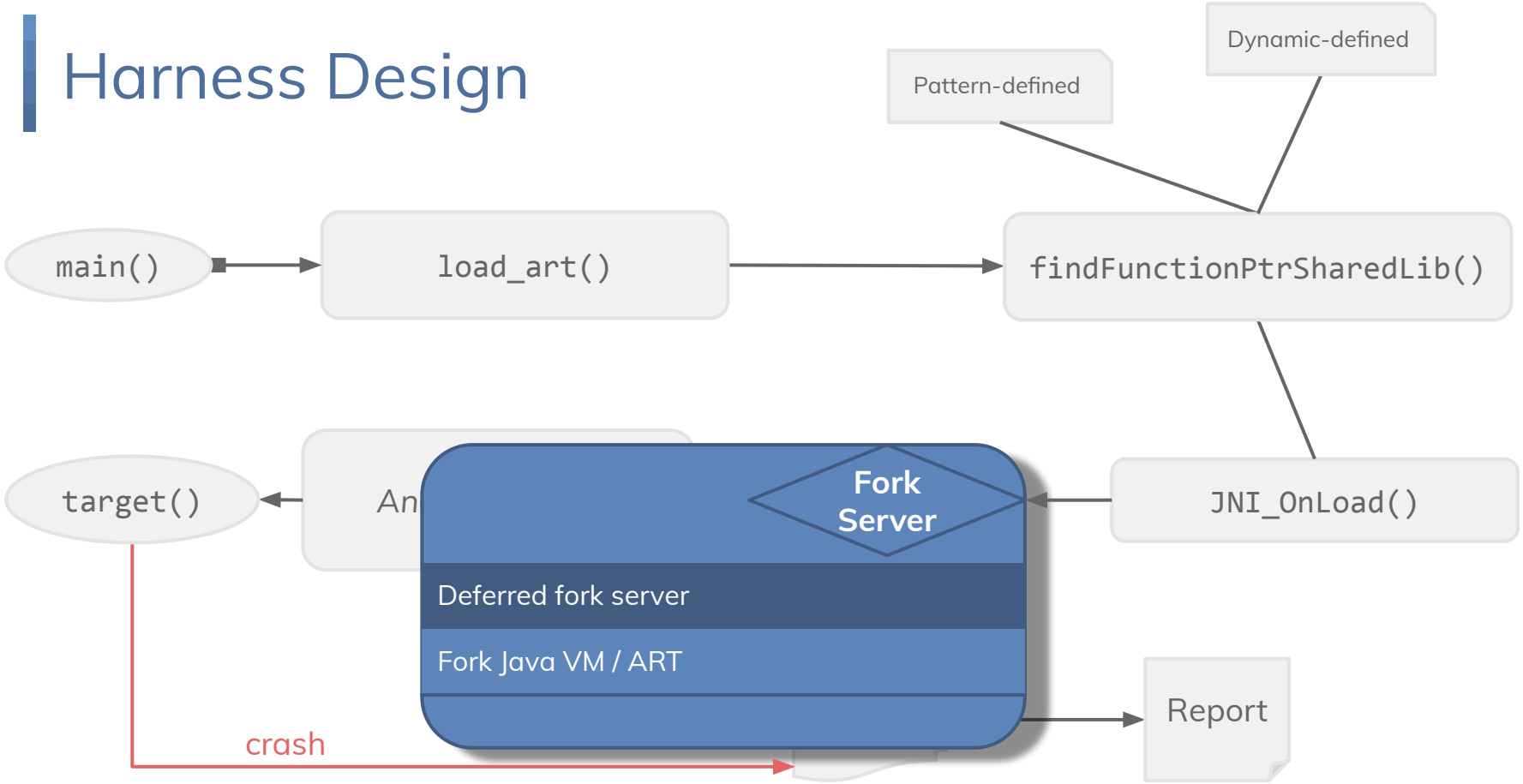
Harness Design



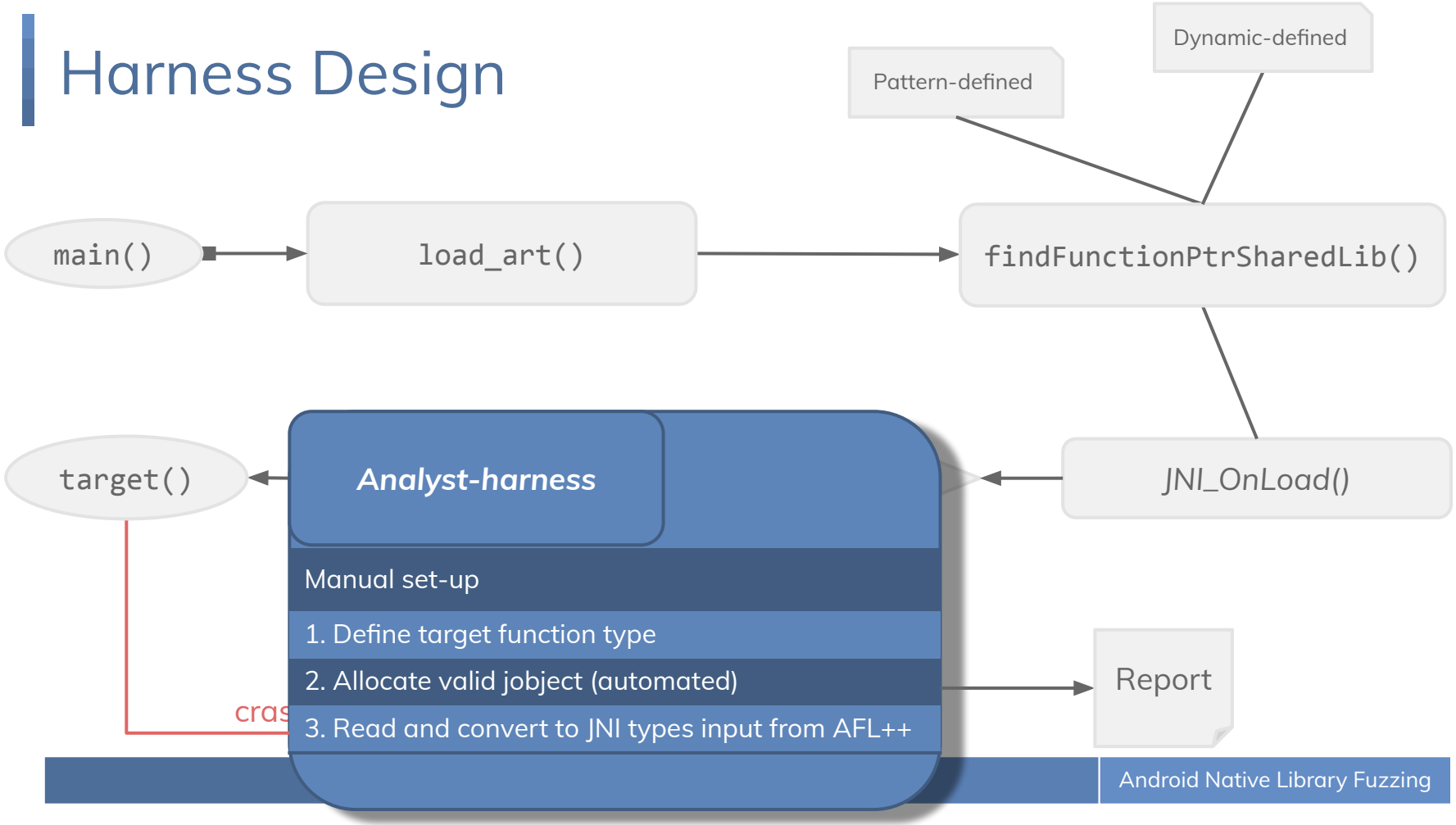
Harness Design



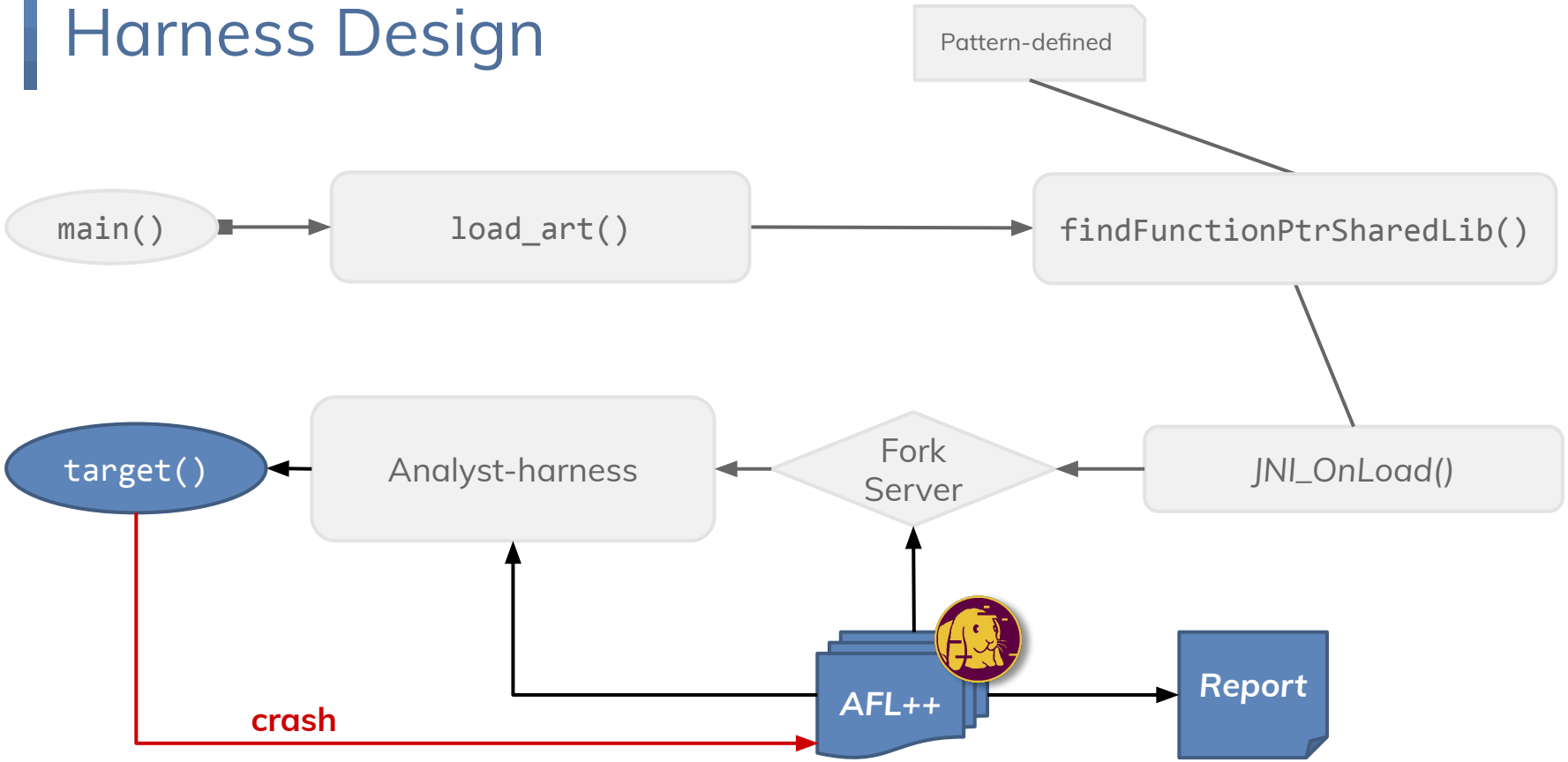
Harness Design



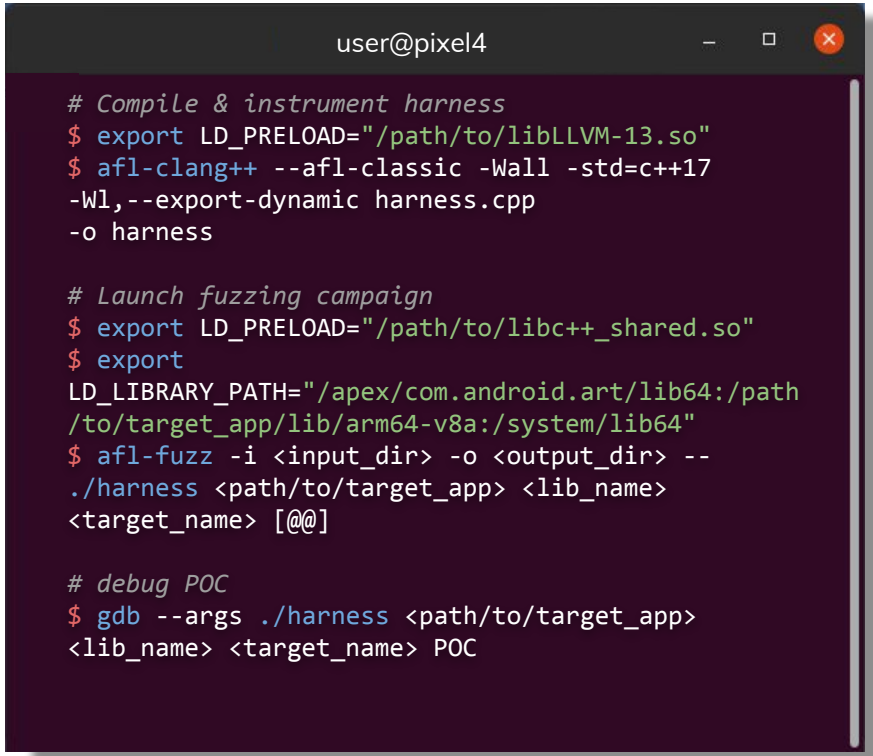
Harness Design

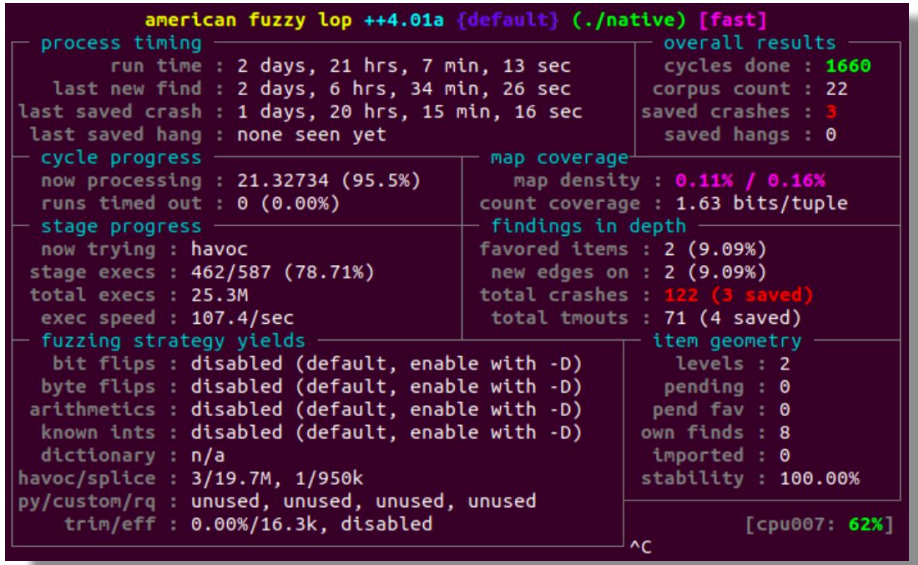


Harness Design



Harness Usage and Performance





Performance:

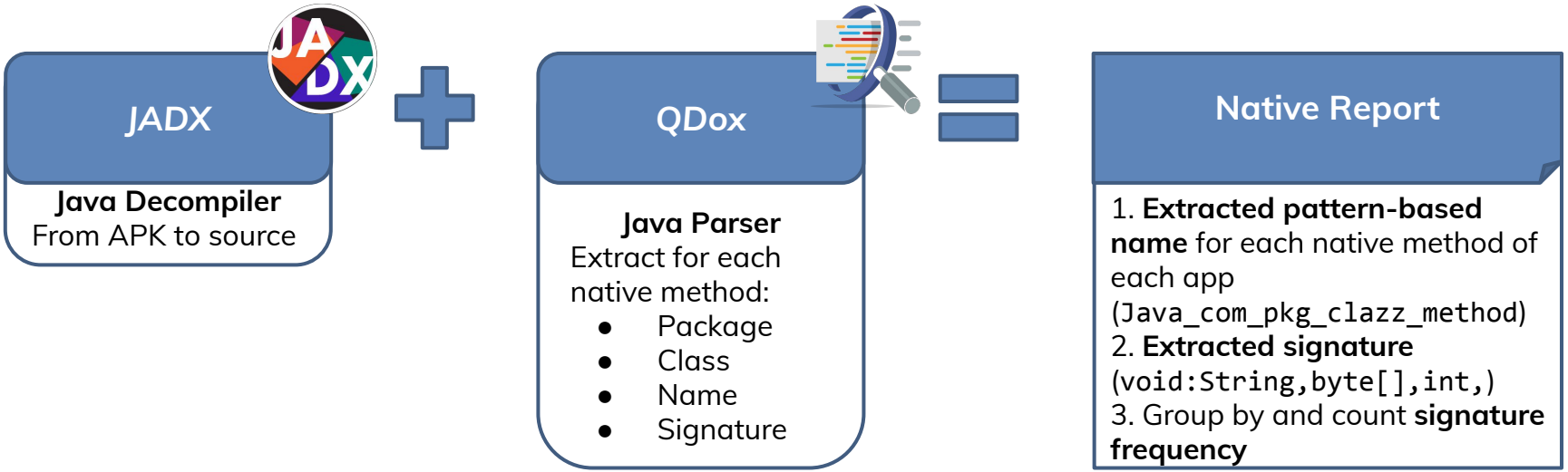
- Google pixel 4: max of 120 execs/sec
- Google pixel 4 (no fork server): 3 execs/sec
- Samsung A40: max of 47 execs/sec

Android Native Fuzzing Framework

Native Extractor, Fuzzing Drivers and Phone Cluster Manager

Native Methods Signature Extractor

Java native methods extractor and signature analysis



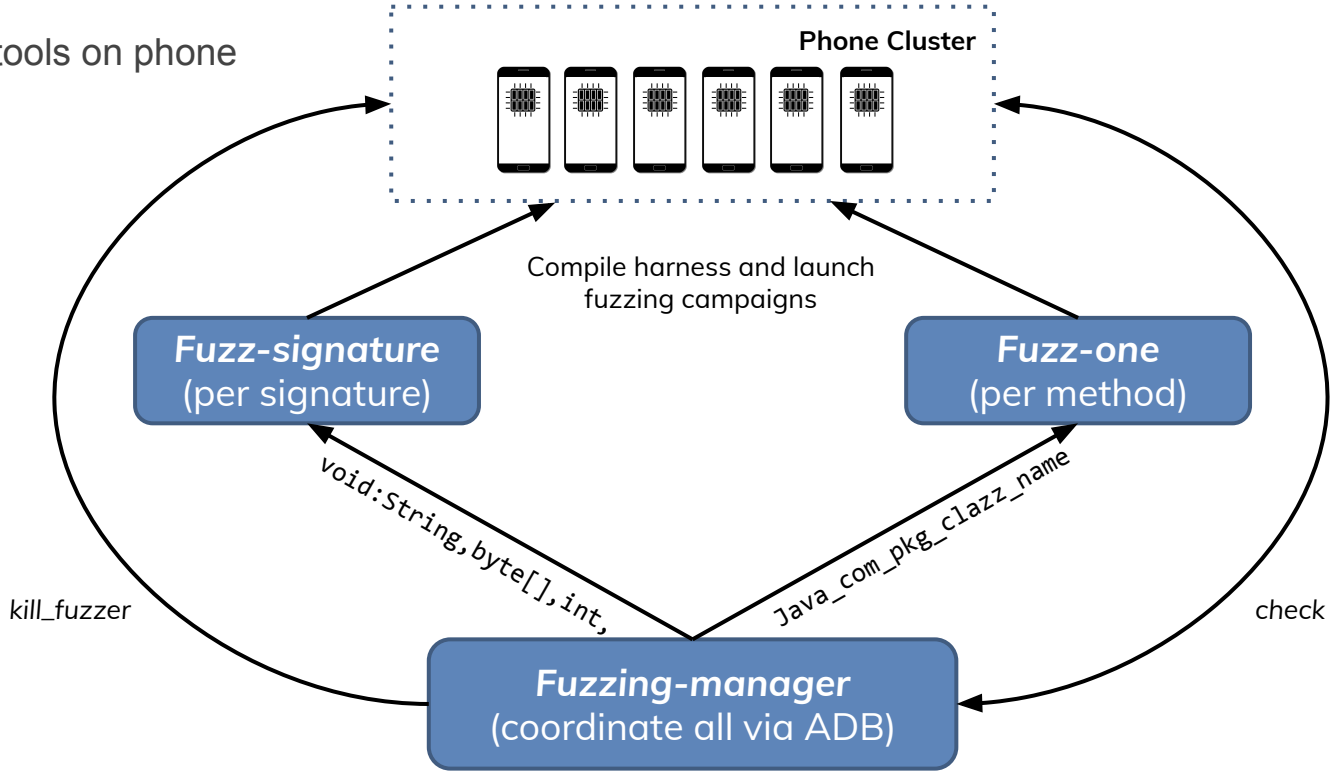
Fuzzing Drivers

Automated fuzzing of a set of Android applications

	Fuzz-One	Fuzz-Signature
Input	Pattern-based method name (`Java_com_pkg_clazz_method`)	Method signature from extractor (`void:String,byte[],int,`)
Output	Fuzzing campaign results	Fuzzing campaign results (for each method)
Features	• AFL++ like set-up • Fuzz campaign duration • From file or *stdin*	• AFL++ debug mode • Parallel fuzzing (up to N cores)
Tools	AFl++Patched, wait, timeout, others…	
Usage	`$ ./fuzzing_one.sh <method-chosen> <time-to-fuzz> <input-dir> <output-dir> <read-from-file[0|1]> <AFL_DEBUG[0|1]> <parallel-fuzzing[0|N]>`	`$ ./fuzzing_driver.sh <signature-chosen> <time-to-fuzz> <input-dir> <output-dir> <read-from-file[0|1]> <AFL_DEBUG[0|1]> <parallel-fuzzing[0|N]>`

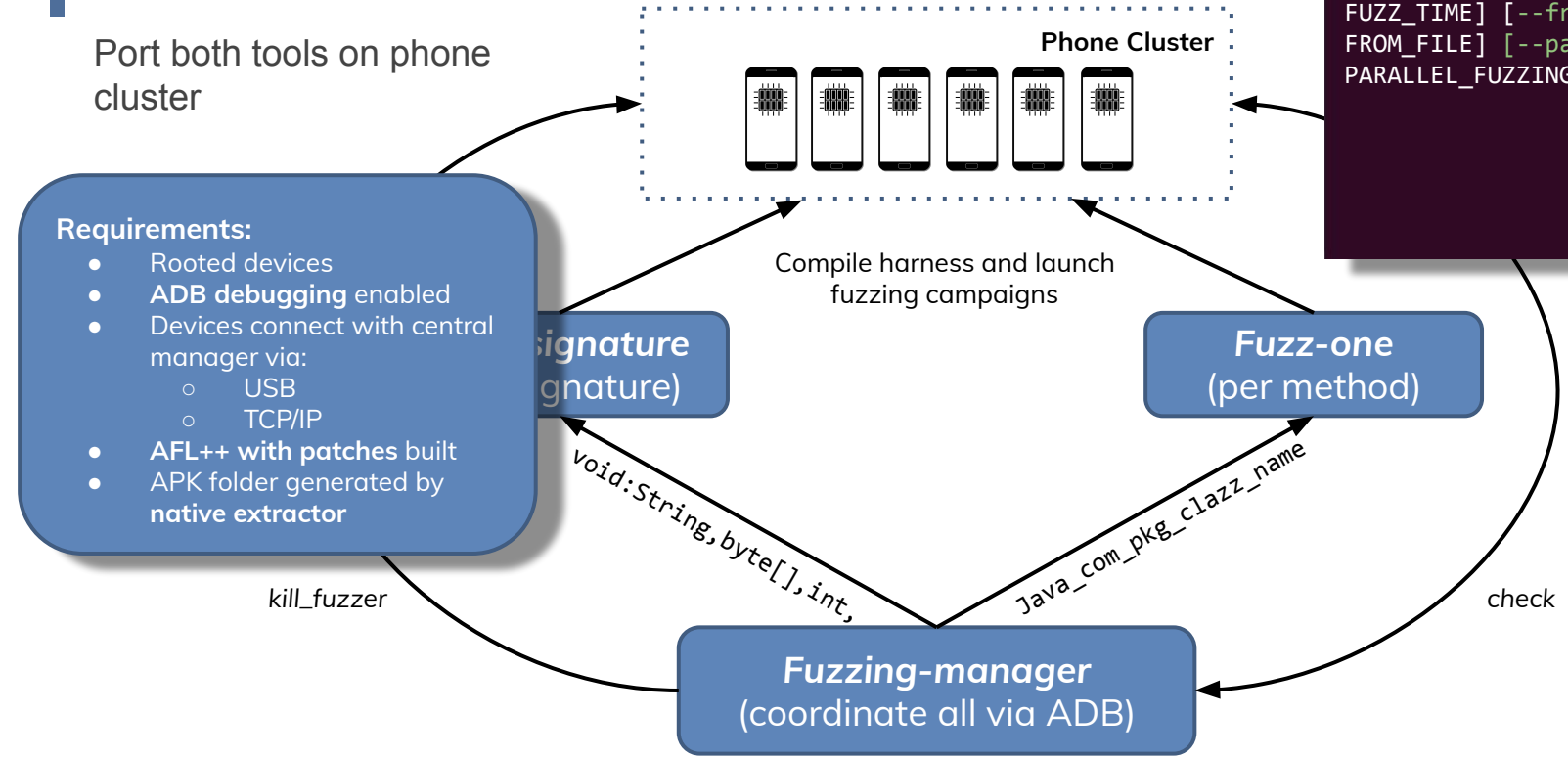
Phone Cluster Manager

Port both tools on phone cluster



Phone Cluster Manager

Port both tools on phone cluster



manager@cluster

```
$ python fuzzing_manager.py [-h]
--action {fuzz_signature,
fuzz_one, check, kill_fuzzer}
[--target TARGET] [--fuzz_time
FUZZ_TIME] [--from_file
FROM_FILE] [--parallel_fuzzing
PARALLEL_FUZZING]
```

Results

Performance and use

Devices

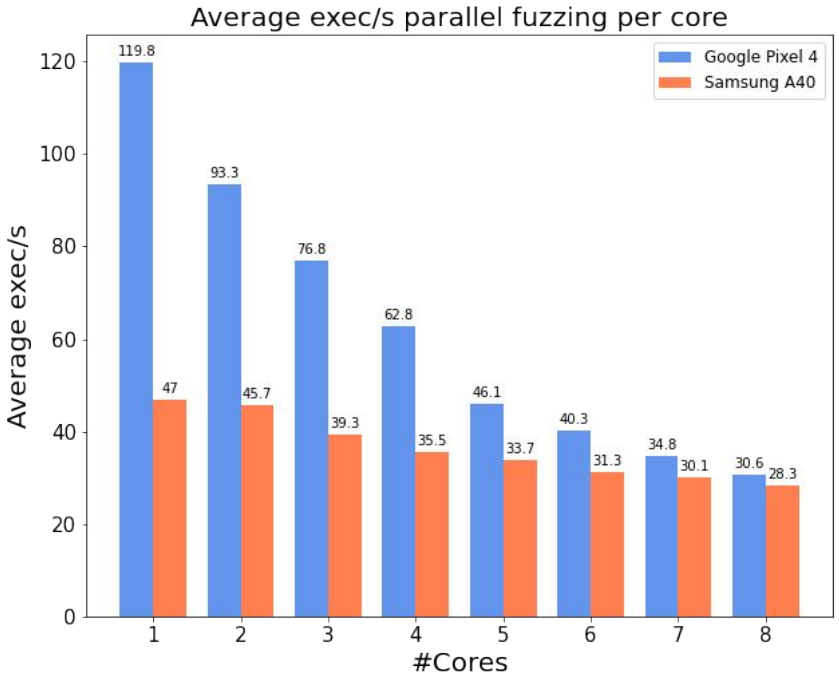
Google Pixel 4

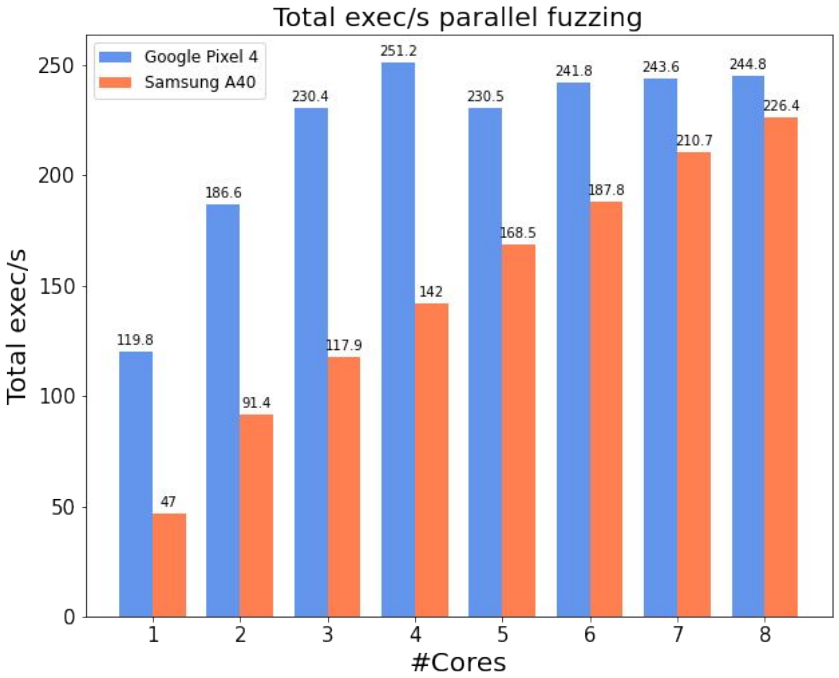
- Qualcomm SM8150 Snapdragon 855
- Octa-core (1x2.84 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.78 GHz Kryo 485)

Samsung A40

- SAMSUNG Exynos 7 Octa 7904
- Octa-core (2x1.77 GHz Cortex-A73 & 6x1.59 GHz Cortex-A53)

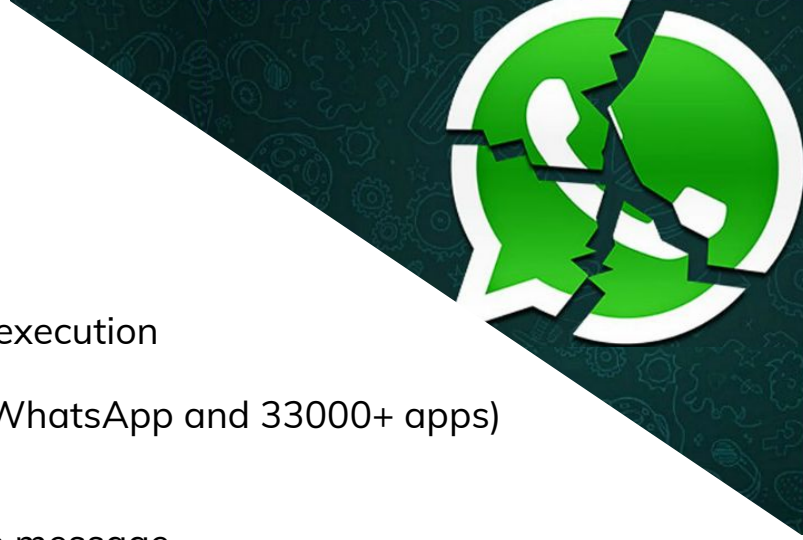
Framework Performance*





*Tested on a modified version of the HelloJNI-Callback toyapp provided by Android Studio

Bug Reproducibility

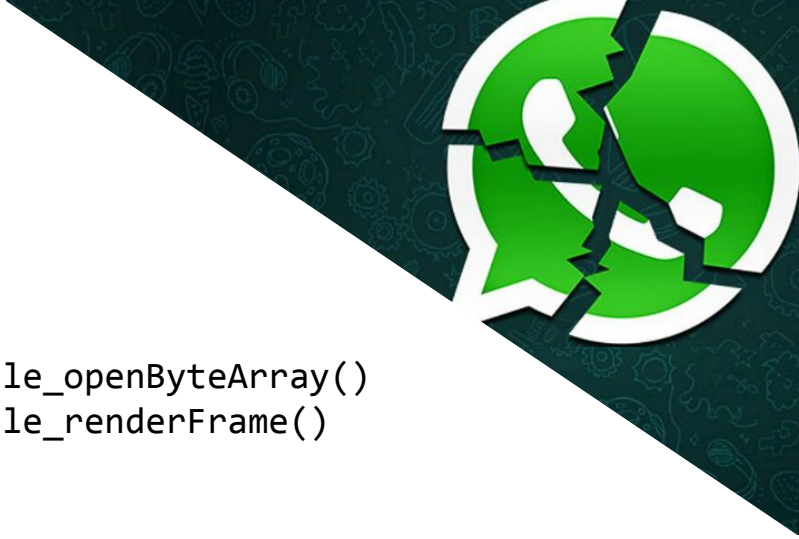


"CVE-2019-11932"

- **What**? double free vulnerability, allowing remote code execution
- **Where**? Android-Gif-Drawable[15] GIF parsing library (WhatsApp and 33000+ apps)
- **How**?
    1. Attacker sends corrupted GIF via Whatsapp message
    2. Victim downloads it into gallery
    3. Victim clicks on 📎 to send new file
    4. The GIF is loaded for preview, triggering the double free

Corrupted GIF: GIF with 2 consecutives frames with size 0
Vulnerable code: use of realloc with size 0 → free

Bug Reproducibility



"CVE-2019-11932"

- **Harness (very important)**:
  1. `Java_pl_droidsonroids_gif_GifInfoHandle_openByteArray()`
  2. `Java_pl_droidsonroids_gif_GifInfoHandle_renderFrame()`
- **Fuzzing**:
  - For 48 hours
  - Double free produces crash because FORTIFY activated
- **Extra**: reproduced heap buffer overflow (caused by corrupted GIF file with `image_height > canvas_height` (or width...) → bug fixed with v1.2.20

Dataset

AndroZoo dataset (APKs):

- Dexdate > 01/01/2021
- Maximum size = 10MB

25,988 APK

3,743 native APK

Fuzzing Results

"Automated fuzzing following a signature-based approach"

Signature	Testable Ratio	Testable Percentage	Fuzzing Duration [h]	#Cores	#Crashes
`void:String,`	42/238	17.6%	2	4	3
`long:String,`	6/12	50.0%	2.5	6	0
`String:String,`	14/60	23.3%	2	6	1
`boolean:String,`	8/39	20.5%	2	8	0
`int:String,`	16/46	34.8%	2	8	4
`void:String,String,`	12/34	35.3%	2.5	8	2
`long:String,String,`	1/1	100.0%	2	8	0
`int:String,String,`	6/8	75.0%	2	8	0
`boolean:String,String,`	4/8	50.0%	2	8	0
`String:String,String,`	1/12	8.33%	2	8	0
`boolean:String,String,String,`	1/6	16.0%	2	8	0
`void:String,String,String,`	3/10	30.0%	2	8	0
Total	114/474	24.0%	-	-	10

Bugs Discovered

Bug #1

- **What?** Off-by-one stack buffer overflow
- **Where?** `Java_net_sourceforge_zbar_Image_setFormat` in 3 APKs (*Chimpa Bazaar*, *Onix Worker* and *Barcode And QR Code Generator*)
- **How?** Call to `GetStringUTFRegion` with invalid target size (input unchecked)
- **Triggerable from Java?** No, string is hardcoded

Bug #2

- **What?** Implementation bug (not conform to specifications)
- **Where?** Harness (actually any library using…)
- **How?** Call to `NewStringUTF` with non UTF-8 or modified UTF-8 encoded string → free string, does not invalidate pointer, return valid Java object → use after free (`malloc`)
- **Triggerable from Java?** No, strings in Java are UTF-16, but…

Bugs Discovered

Bug #3

- **What?** Stack buffer overflow
- **Where?** 4 native methods handling OPUS audio files (in *Live Microphone To Speaker* app - 100K+ downloads)
- **How?** Call to GetStringUTFRegion with invalid target size (input unchecked)
- **Triggerable from Java?** Yes, using a OPUS file with name of sufficient length

Bugs Discovered

Bug #4

- **What?** Stack buffer overflow
- **Where?** `Java_Runtime_Native_init` in game PnuYozhika 3
- **How?** Call to `__strcpy_chk` with invalid target size (input unchecked, but caught using FORTIFY)
- **Triggerable from Java?** No

Bug #5*

- **What?** Stack buffer overflow
- **Where?** `Java_bestdict_common_code_BisObject_GetSound` in 71 dictionary applications (most used is a Thai dictionary with 1M+ downloads)
- **How?**
  - V.18: unchecked use of `sprintf`
  - V.19+: unchecked use of `memcpy`
- **Triggerable from Java?** Yes, with corrupted word dictionary

*Following a per-library analysis

Future Work

Future Work

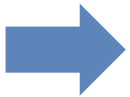
- **Performance** → caching mechanism and deferred fork server
- **Stateful Fuzzing** → source-to-sink static analysis (SE) to generate:
  - Native calls sequence
  - Parameters constraints
- **Binary Instrumentation** → coverage guided fuzzing using:
  - Static rewriting tools (RetroWrite)
  - Dynamic rewriting tools (ARM CoreSight)

AFL++ CoreSight

"Achieve grey-box fuzzing dynamically"

Motivation

- **Harness** fuzzes each library in a **black-box** fashion
- Application's are **closed source**
- Static rewriting tools not available (e.g. RetroWrite)
- QEMU slow (2-5x)



AFL++ CoreSight

- Leverage CoreSight ARM process's feature
- Capture branch executions at runtime
- Outperform QEMU (no VM)
- Design:
  - Coresight-trace
    - Trace source
    - Trace sink
    - Trace link
  - Coresight-decoder

AFL++ CoreSight - Build

- **Capstone disassembler**
- ***Coresight-trace* and *Coresight-decoder*:**
  - Bionic missing `pthread_setaffinity_np` → substitute with `sched_setaffinity`
- **Patched glibc**
  - Why? To provide the fork server
  - How? With *patchelf*
  - Why not?
    - The GNU C library requires GCC compiler (> v6.2)
    - NDK offers only LLVM/Clang (GCC v4.9 until NDK r17), same for cross-compiler
  - Alternative? **Musl** → too many compilation dependencies missing

Bibliography

Bibliography

[1] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, “Information flow analysis of android applications in droidsafe.”, NDSS, 2015, p. 110

[2] 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”, Acm Sigplan Notices, vol. 49, no. 6, 2014, pp. 259–269

[3] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “Iccta: Detecting inter-component privacy leaks in android apps”, 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 280–291

[4] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: statically vetting android apps for component hijacking vulnerabilities”, Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 229–240

[5] F. Wei, “Sankardas roy, xinming ou, et almbox. 2014. amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps”, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM

Bibliography

[6] H. Shahriar, S. North, and E. Mawangi, “Testing of memory leak in android applications”, 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, 2014, pp. 176–183, DOI 10.1109/HASE.2014.32

[7] H. Ye, S. Cheng, L. Zhang, and F. Jiang, “Droidfuzzer: Fuzzing the android apps with intent filter tag”, Proceedings of International Conference on Advances in Mobile Computing and Multimedia, New York, NY, USA, 2013, p. 68?74, DOI 10.1145/2536853.2536881

[8] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, “Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs”, Proc. ACM Program. Lang., vol. 5, oct 2021, DOI 10.1145/3485533

[9] S. Arzt and E. Bodden, “Stubdroid: Automatic inference of precise data-flow summaries for the android framework”, 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 725–735, DOI 10.1145/2884781.2884816

[10] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, “Jucify: A step towards android code unification for enhanced static analysis”, 2021, DOI 10.48550/ARXIV.2112.10469

Bibliography

[11] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, “Identifying java calls in native code via binary scanning”, Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 2020, p. 388?400, DOI 10.1145/3395363.3397368

[12] C. Rizzo, “Static flow analysis for hybrid and native android applications”. PhD thesis, Royal Holloway, University of London, 2020. (unpublished)

[13] A. Moroo and Y. Sugiyama, “Armored coresight: Towards efficient binary-only fuzzing”, Ricerca Security, November 10, 2021. https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html

[14] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. De Geus, C. Kruegel, and G. Vigna, “Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy”, 02 2016, DOI 10.14722/ndss.2016.23384

[15] Koral, “Android-gif-drawable.” https://github.com/koral--/android-gif-drawable.git, 2013

Appendix

Native Functions Naming Conventions

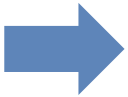
Pattern-defined	Dynamically-defined

● **Generated by *javac*** when compiling
● **Construction pattern** (“_” separated)
  ○ *Java*
  ○ *Package name*
  ○ *Class name*
  ○ *Method name* (Java side)
  ○ *Mangled argument signature*
● **Symbols exported**
● JVM performs dynamic symbol lookup
 	
Example
● Java name: `nativeMethod`
● Native name:
`Java_com_name_jni_package_JNI_nativeMethod`
	● Developer in charge of **manually register** Java methods with Native functions
	● Preferred native name
	● **Construction steps**
	  1. Implement natively **`JNI_OnLoad`** (defined in `jni.h`)
	  2. Define `JNINativeMethod` struct, with for each entry:
	  - Java method name (hard-coded string)
	  - Java method signature (*smali*)
	  - Native function address
	  3. Call `JNIEnv` function **`RegisterNatives`** with struct as parameter
	● **Only `JNI_OnLoad` exported**
	● JVM calls JNI_OnLoad as soon as the library is loaded

Native Methods Extractor Success Rate

Run extractor on:

- 3,734 APKs
- 17M+ Java files
- 275,171 native methods (non unique)



Unable to parse 743 Java files (0.0042%), missing 15 native methods overall

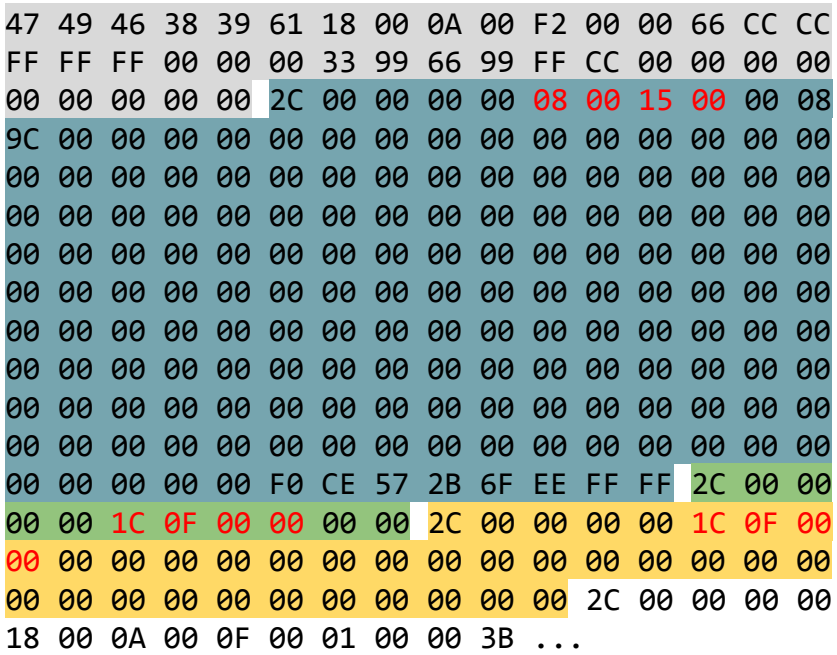
Threads on JNI_CreateJavaVM()

```
[#0] Id 1, Name: "main", stopped 0x5555564ec4 in main (), reason: SINGLE STEP
[#1] Id 2, Name: "Jit thread pool", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#2] Id 3, Name: "Runtime worker ", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#3] Id 4, Name: "Runtime worker ", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#4] Id 5, Name: "Runtime worker ", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#5] Id 6, Name: "Runtime worker ", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#6] Id 7, Name: "Signal Catcher", stopped 0x7fbdc70978 in __rt_sigtimedwait (), reason: SINGLE STEP
[#7] Id 8, Name: "HeapTaskDaemon", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#8] Id 9, Name: "ReferenceQueueD", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#9] Id 10, Name: "FinalizerDaemon", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
[#10] Id 11, Name: "FinalizerWatchd", stopped 0x7fbdc2034c in syscall (), reason: SINGLE STEP
```

```
gef➤  bt
#0  0x0000007fbe0dd34c in syscall () from /apex/com.android.runtime/lib64/bionic/libc.so
#1  0x0000007d2d43f930 in art::ConditionVariable::WaitHoldingLocks(art::Thread*) () from /apex/com.android.art/lib64/libart.so
#2  0x0000007d2d859a30 in art::ThreadPool::GetTask(art::Thread*) () from /apex/com.android.art/lib64/libart.so
#3  0x0000007d2d858c94 in art::ThreadPoolWorker::Run() () from /apex/com.android.art/lib64/libart.so
#4  0x0000007d2d8587a4 in art::ThreadPoolWorker::Callback(void*) () from /apex/com.android.art/lib64/libart.so
#5  0x0000007fbe141d50 in __pthread_start(void*) () from /apex/com.android.runtime/lib64/bionic/libc.so
#6  0x0000007fbe0e228c in __start_thread () from /apex/com.android.runtime/lib64/bionic/libc.so
```

```
gef➤  bt
#0  0x0000007fbe12d978 in __rt_sigtimedwait () from /apex/com.android.runtime/lib64/bionic/libc.so
#1  0x0000007fbe0eec7c in sigwait () from /apex/com.android.runtime/lib64/bionic/libc.so
#2  0x0000007d2d811b90 in art::SignalCatcher::WaitForSignal(art::Thread*, art::SignalSet&) () from /apex/com.android.art/lib64/libart.so
#3  0x0000007d2d810788 in art::SignalCatcher::Run(void*) () from /apex/com.android.art/lib64/libart.so
#4  0x0000007fbe141d50 in __pthread_start(void*) () from /apex/com.android.runtime/lib64/bionic/libc.so
#5  0x0000007fbe0e228c in __start_thread () from /apex/com.android.runtime/lib64/bionic/libc.so
```

WhatsApp Vulnerability - Corrupted GIF



Legenda:

- Frame #1
- Frame #2
- Frame #3
- Image Height and width

Signature Analysis

Frequency	Signature	Frequency	Signature
2487	void:	283	int:int
1343	int:	233	void:Object,
921	String:	212	void:int,int,
721	void:int,	211	long:
640	void:String,	206	long:long,
583	boolean:	179	long:int,
523	Dialog:Bundle,	172	void:DialogInterface,
509	void:long	160	void:a,
437	void:Bundle,	142	void:long,long,
436	void:View,	138	Void:CharSequence,int,int,int,
416	void:Context,	132	String:String,
393	void:boolean,	126	boolean:Object,
339	int:long,	117	void:RecyclerView.b0,int,