Securing Native Libraries on Android with LFI

Zachary Yedidia (Stanford), Nathan Egge (Google), Matthias Blume (Google), Daniel Moghimi (Google), Úlfar Erlingsson (Google), **Tal Garfinkel** (UCSD)

Android is Critical Infrastructure

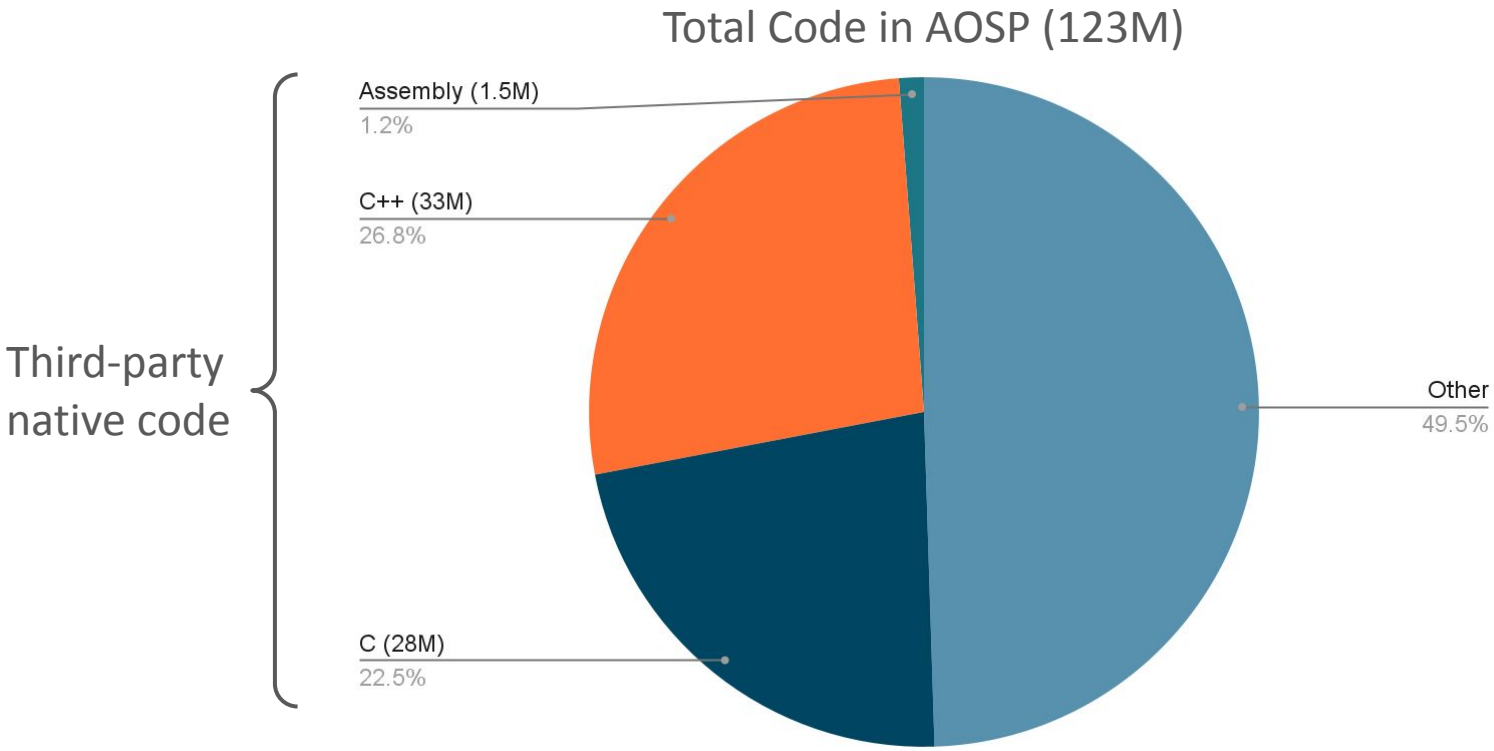
3.5 billion devices and growing

45% of global OS market

72% of global smartphone market (over 3B devices)

>45% of tablets

Half of Android Open Source Project (AOSP) is Unsafe Third-Party Code

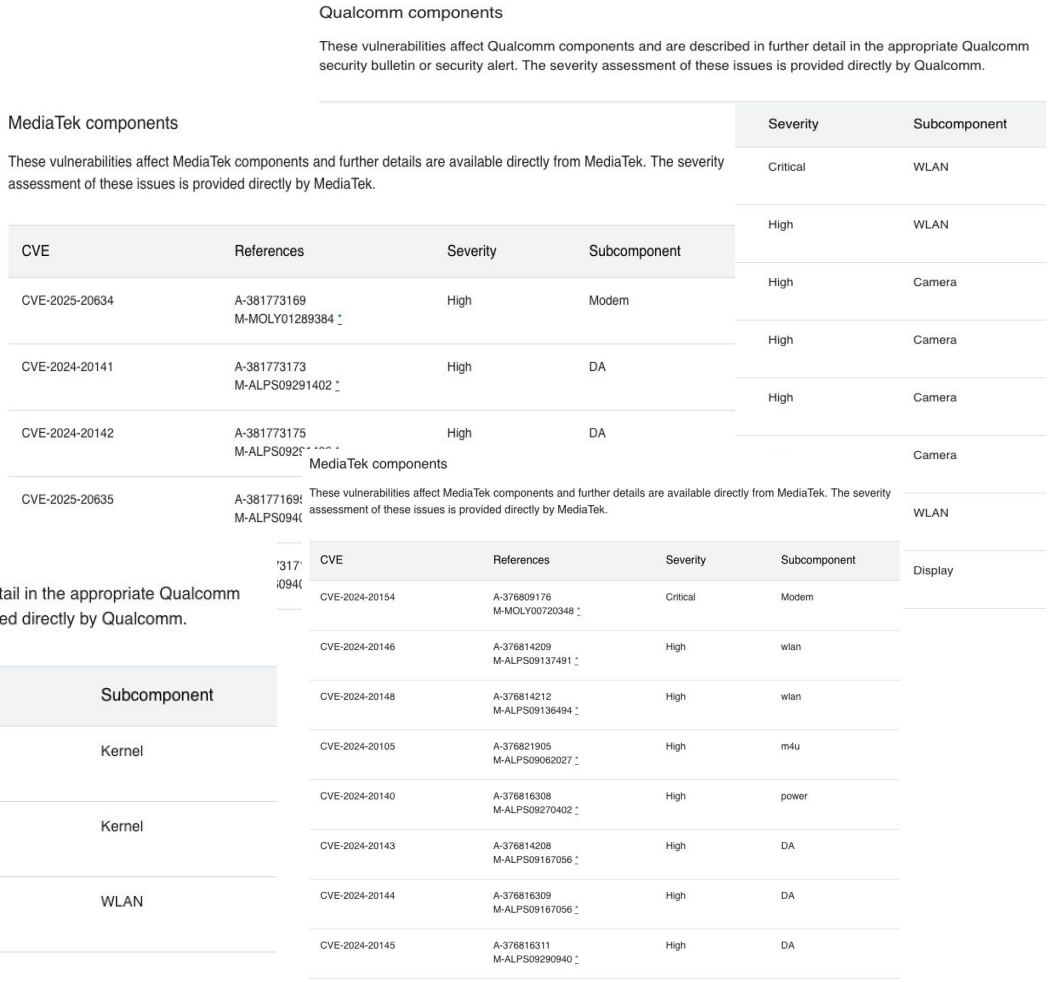


Third-Party Device Drivers

Often exploitable

Frequently changing (new code)

→ Privilege escalation!, RCE



Qualcomm components

These vulnerabilities affect Qualcomm components and are described in further detail in the appropriate Qualcomm security bulletin or security alert. The severity assessment of these issues is provided directly by Qualcomm.

CVE	References	Severity	Subcomponent
CVE-2024-21464	A-350500921 QC-CR#3445828	High	Kernel
CVE-2024-45553	A-372003017 QC-CR#3845043 [2]	High	Kernel
CVE-2024-45558	A-372002616 QC-CR#3852338	High	WLAN

MediaTek components

These vulnerabilities affect MediaTek components and further details are available directly from MediaTek. The severity assessment of these issues is provided directly by MediaTek.

CVE	References	Severity	Subcomponent
CVE-2025-20634	A-381773169 M-MOLY01289384	High	Modem
CVE-2024-20141	A-381773173 M-ALPS09291402	High	DA
CVE-2024-20142	A-381773175 M-ALPS092	High	DA
CVE-2025-20635	A-38177169 M-ALPS094		

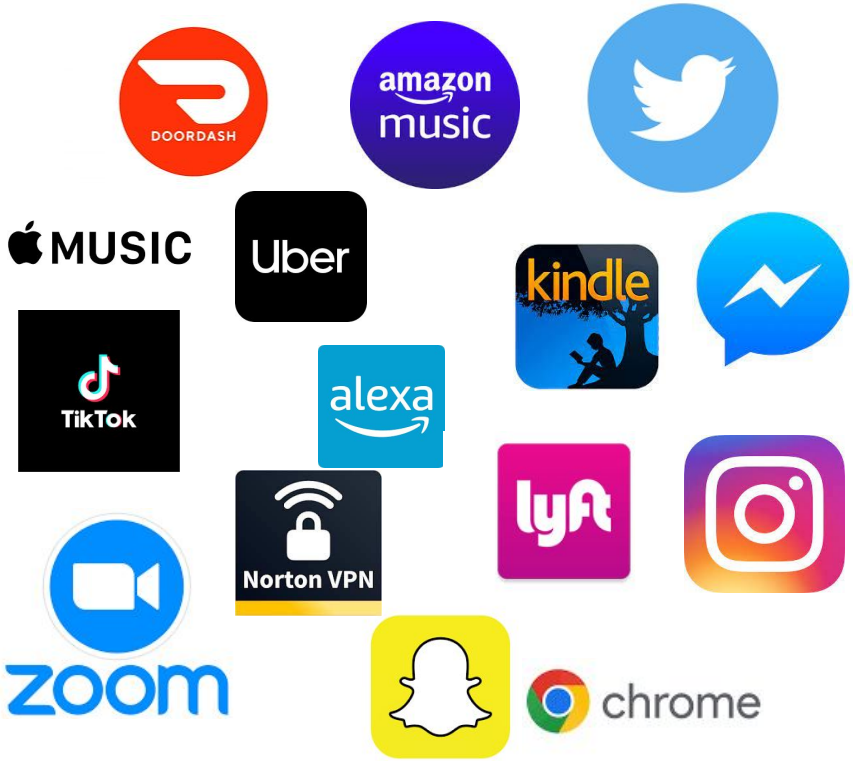
Severity	Subcomponent
Critical	WLAN
High	WLAN
High	Camera
High	Camera
High	Camera
	Camera
	WLAN
	Display

MediaTek components

These vulnerabilities affect MediaTek components and further details are available directly from MediaTek. The severity assessment of these issues is provided directly by MediaTek.

CVE	References	Severity	Subcomponent
CVE-2024-20154	A-376809176 M-MOLY00720348	Critical	Modem
CVE-2024-20146	A-376814209 M-ALPS09137491	High	wlan
CVE-2024-20148	A-376814212 M-ALPS09136494	High	wlan
CVE-2024-20105	A-376821905 M-ALPS09062027	High	m4u
CVE-2024-20140	A-376816308 M-ALPS09270402	High	power
CVE-2024-20143	A-376814208 M-ALPS09167056	High	DA
CVE-2024-20144	A-376816309 M-ALPS09167056	High	DA
CVE-2024-20145	A-376816311 M-ALPS09290940	High	DA

Lots of (Unpatched) third-party native libraries in popular Apps



Frequent vulnerabilities

App Name	Vul Lib Version	Vul Announced	TTRP (Days)	TTAF (Days)
Xbox	XML2-2.7.7	2014-11-04	12	1956
Apple Music	XML2-2.7.7	2014-11-04	12	1704
TikTok	GIFLib-5.1.1	2015-12-21	87	1429
Zoom Meetings	OpenSSL-1.0.0a	2010-08-17	91	1323
Amazon Alexa	OpenSSL-1.0.1s	2016-05-04	12	1086
Amazon Kindle	Libpng-1.6.34	2017-01-30	330	1019
StarMaker	FFmpeg-3.2	2016-12-23	4	1001
eBay	OpenCV-2.4.13	2017-08-06	41	905
Fitbit	SQlite3-3.20.1	2017-10-12	12	902
Uber	OpenCV-2.4.13	2017-08-06	41	830
Snapchat	SQlite3-3.20.1	2017-10-12	12	670
Discord	GIFLib-5.1.1	2015-12-21	87	665
Lyft	OpenCV-2.4.11	2017-08-06	41	662
Twitter	GIFLib-5.1.1	2015-12-21	87	457
Instagram	FFmpeg-2.8.0	2017-01-23	2	267

Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code

Current Defenses are insufficient

Incomplete Mitigations

ASLR, Stack canaries, CFI, bounds checks

Routinely bypassed

Completeness vs performance trade-off

Rewrite code in safe language (Rust)?

Huge engineering cost and time: rewrite, retest, support

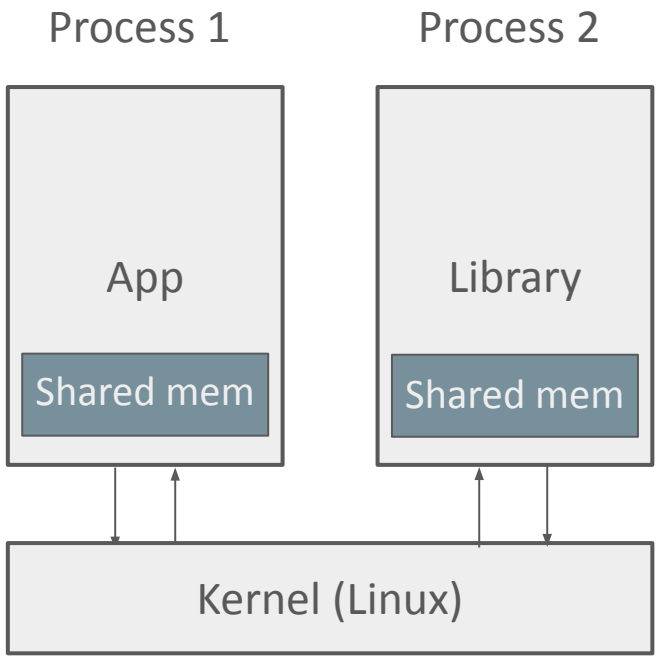
Billions of lines of existing code you don’t own

Process-Based Sandboxing: Heavy weight

High IPC Costs: 1000s of cycles for a context switch

Can cause up to 50% overhead on Android media workloads (Audio/Video codecs)

“A key limitation is that the process is the smallest unit of isolation, but processes are not cheap. Especially on Android, using more processes impacts device health overall” - Google Chrome Memory Safety



Our Approach: In-Process Sandboxing (“Compartments”)

Retrofitting Fine Grain Isolation in the Firefox Renderer

Extended Version

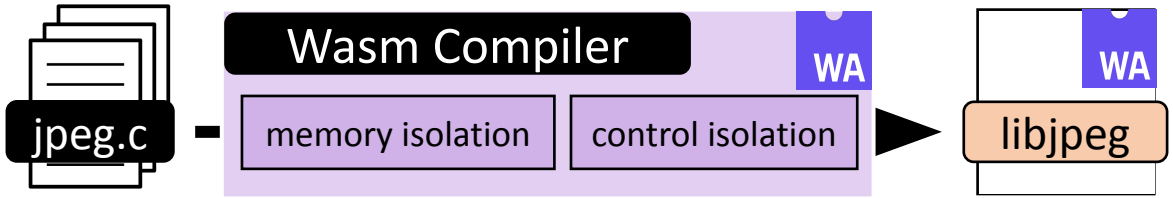
Shravan Narayan[†] Craig Disselkoen[†] Tal Garfinkel[*] Nathan Froyd[◇] Eric Rahm[‹]
Sorin Lerner[†] Hovav Shacham[*,†] Deian Stefan[†]
[†]UC San Diego [*]Stanford [◇]Mozilla [*]UT Austin

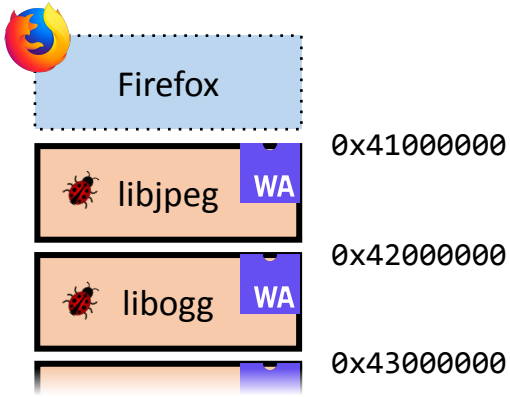
The Road to Less Trusted Code

Lowering the Barrier to In-Process Sandboxing

TAL GARFINKEL, SHRAVAN NARAYAN, CRAIG DISSELKOEN, HOVAV SHACHAM, AND DEIAN STEFAN

Hardening Firefox Render (RLBox + WebAssembly)





Securing Firefox with WebAssembly

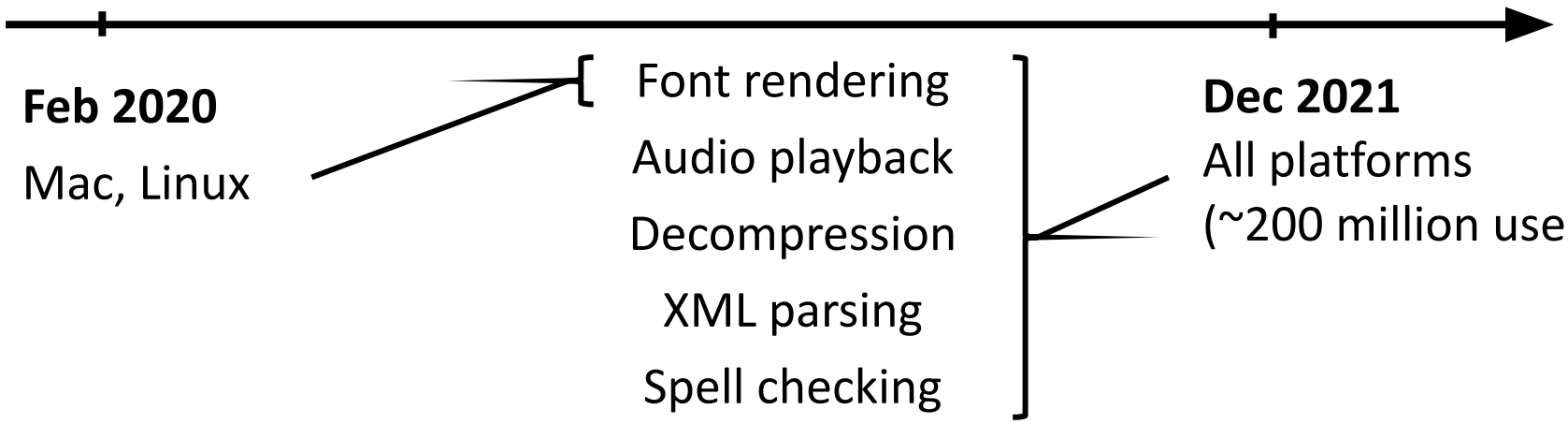
By Nathan Froyd

Posted on February 25, 2020 in Featured Article, Firefox, Rust, Security, and WebAssembly

Protecting the security and privacy of individuals is a central tenet of Mozilla's mission, and so we constantly endeavor to make our users safer online. With a

https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

Deployed in Firefox since 2021



Worked at Scale, Modest effort per-library

Sandboxed libraries

Image and font rendering

Audio video playback

Decompression

XML parsing

Spell checking

Libraries are unmodified!

Automatic security checks
dozens to hundreds per library

Remaining data validation
on average, 2–4 lines of code

Time: a few days per library

Wasm is very limiting…

Compatibility: ❌

Very limited SIMD/intrinsics

No hand-written assembly

No dynamic code generation (JIT)

libc compat (WASI-libc – no glibc, bionic, etc.)

Limited POSIX

ABI differences (different pointer sizes)

Breaks other mitigations (e.g. PAC/MTE)

Performance: ❌

Wasm2c on Arm64:

37.5% Geomean on SPEC 2017

SIMD/hand-written assembly 😞

Dynamic code generation (JIT) 😞

Compatibility and Performance not top priorities for Wasm

Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing

Zachary Yedidia
Stanford University

Low-Level SFI is better (for library sandboxing)

Software-based Fault Isolation (SFI)[1]: an idea from 1993.

High-level SFI (WebAssembly)

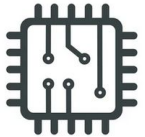
Targets portable IR

Strengths: Platform Independence

Low-level SFI (Original SFI, LFI)

Works directly on machine code.

Strengths: compatibility, performance, simplicity



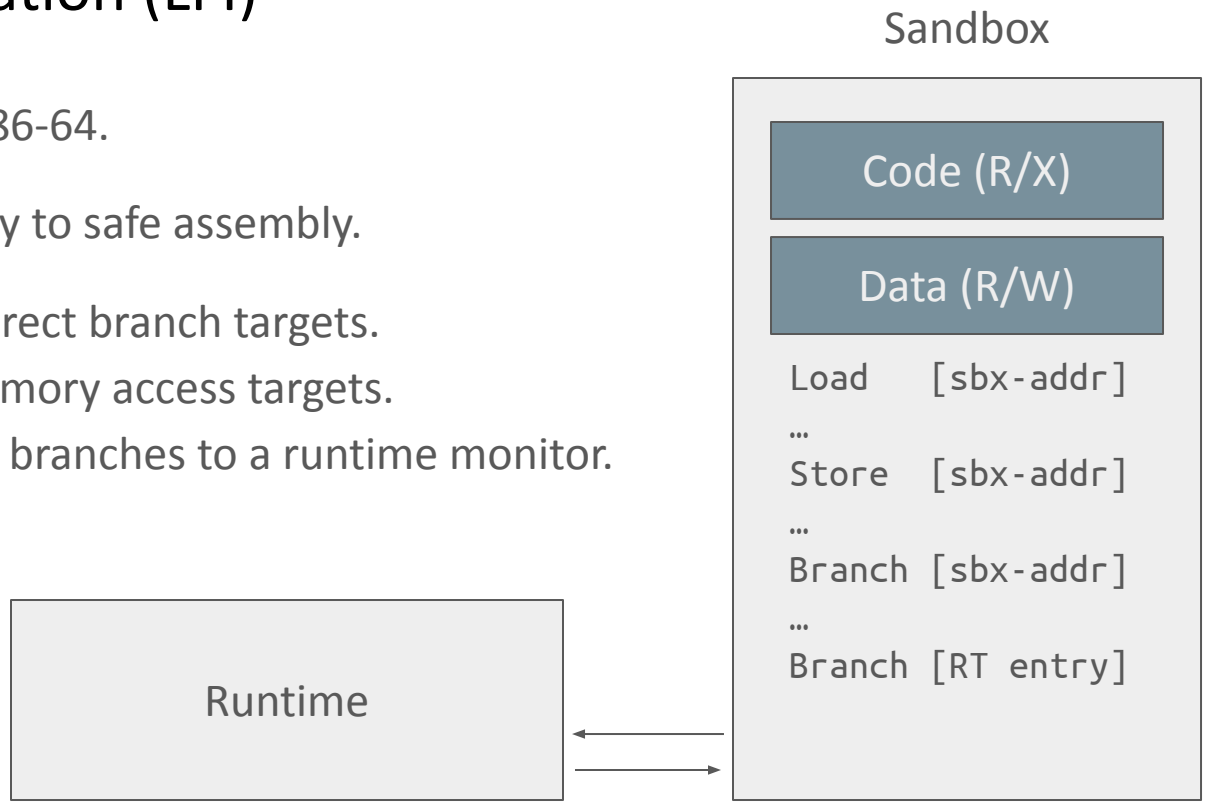
[1]Wahbe et al. (1993). “Efficient Software-Based Fault Isolation.”

Lightweight Fault Isolation (LFI)

Low-level SFI for Arm64 and x86-64.

How it works: rewrite assembly to safe assembly.

- **Control flow**: rewrite indirect branch targets.
- **Loads/stores**: rewrite memory access targets.
- **System calls**: rewrite into branches to a runtime monitor.



LFI: Why It’s Good

Performance: Typically ~6-8% overhead (prior SFI systems: 15-20% for R/W sandboxing).

As low as 1.5% geomean overhead on SPEC2017 for W-only Arm64.

Flexible and more potential for optimization exists.

LFI: Why It's Good

Performance: Typically ~6-8% overhead (prior SFI systems: 15-20% for R/W sandboxing).

As low as 1.5% geomean overhead on SPEC2017 for W-only Arm64.

Flexible and more potential for optimization exists.

Compatibility: LP64 ABI, standard libc (musl, bionic, glibc, etc.), Linux syscall interface.

Security: Machine code verifier → compiler can be untrusted.

Simplicity: Rewriter (~2K LOC), verifier (~400 LOC), runtime (~5K LOC).

We own it: can evolve it for our use-case.

LFI: How It Works

LFI Rewriter (compiler modification)

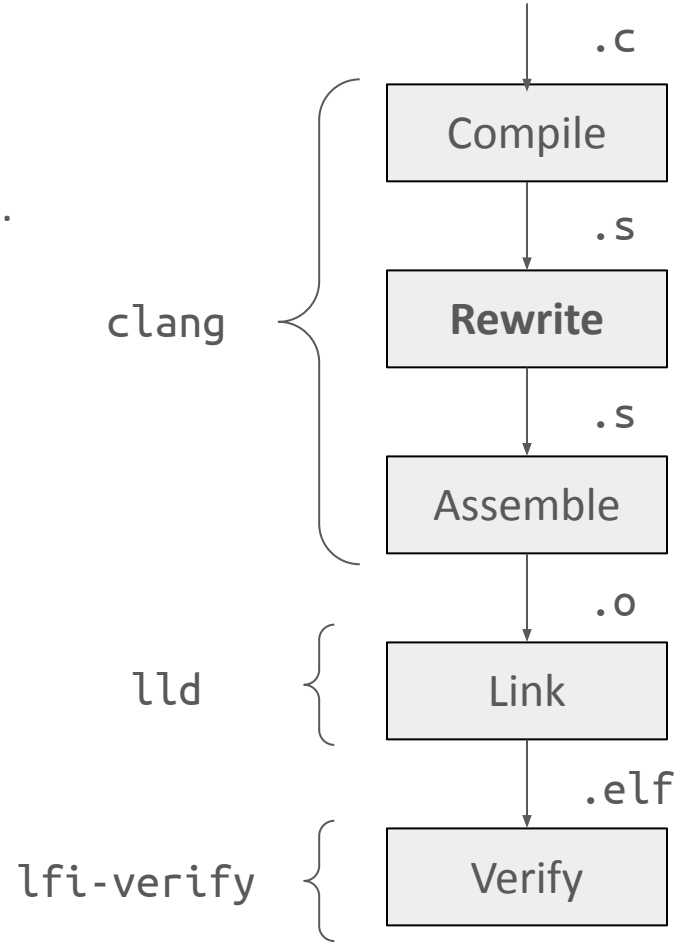
Rewriter plugs into an existing compiler (e.g., LLVM).

Other compiler stages remain unchanged*.

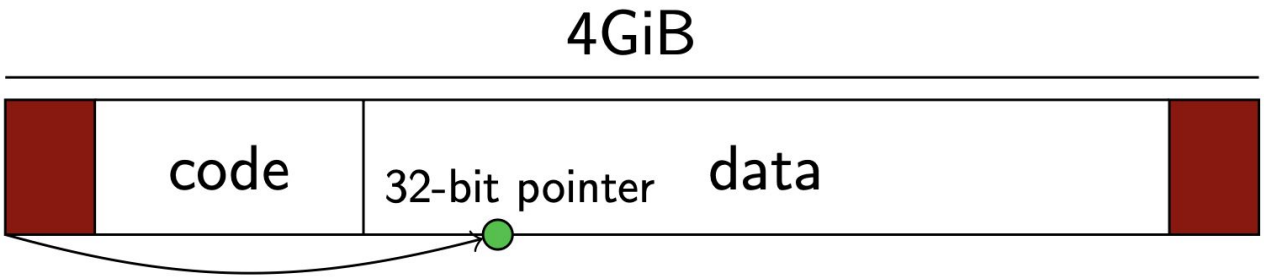
*only need to reserve a few registers.

All sandboxed code goes through the rewriter:

Libc, dynamic linker, libc++, compiler-rt, …



LFI Execution Environment



Runtime enforces strict `W^X`.

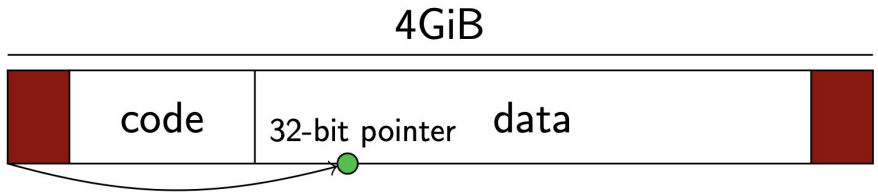
Dynamic codegen can be supported: Run verifier before `mprotect(PROT_EXEC)`.

48-bit address space: supports up to 64K sandboxes.

Rewriting (Masking) Memory Accesses

Reserve x21: sandbox base.

Reserve x18: any sandbox address.




```
ldr x0, [x1]
```

→

```
ldr x0, [x21, w1, uxtw]
```

Mask with addressing mode

```
ldp x0, x1, [x2]
```

→

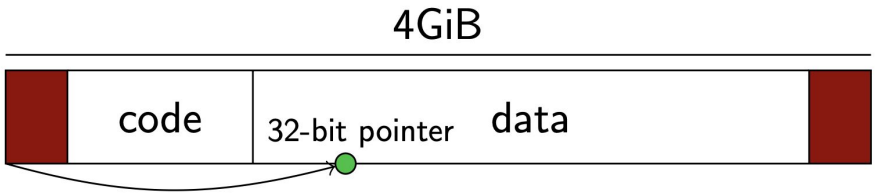
```
add x18, x21, w2, uxtw
ldp x0, x1, [x18]
```

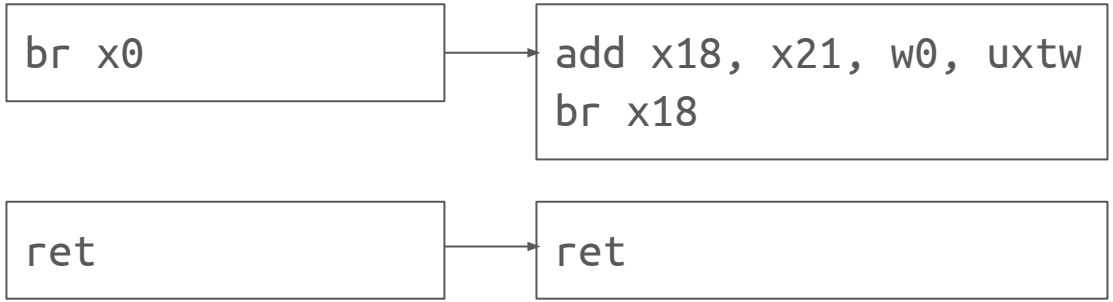
Mask with guard instruction

Rewriting Control Flow

1. Mask all modifications to `x18`, `x30`
2. Rewrite indirect branches to target `x18`.

(Returns target `x30` by convention)





Key: Arm64 instructions are fixed-width!

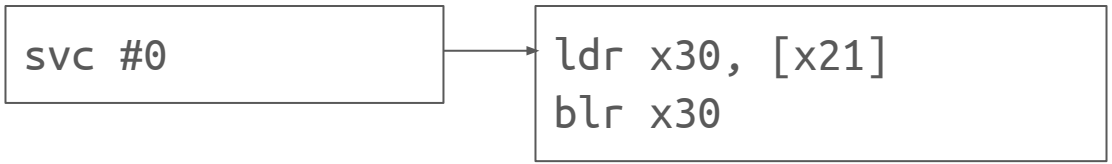
Safe as long as each individual instruction is safe.

Rewriting System Call Instructions

Runtime entrypoints: placed in the first page of the sandbox (read-only).

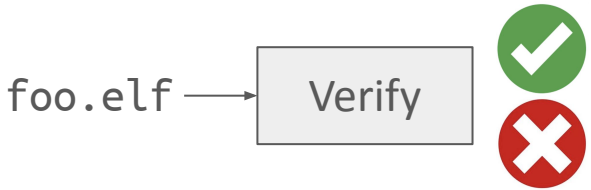
→ x21 already points here!

Examples: syscall request, thread-local storage (TLS) load, TLS store.



LFI Verifier: check if program was correctly rewritten

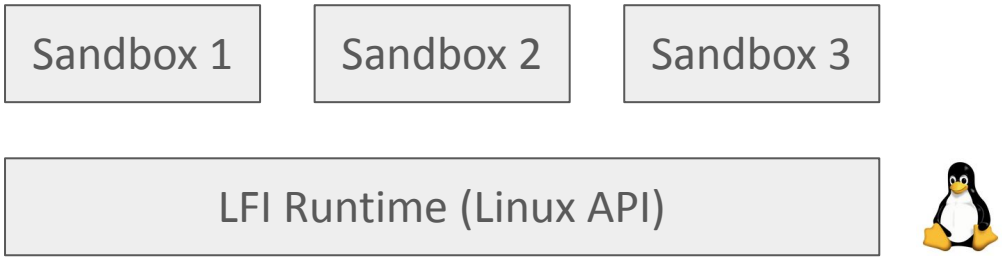
Enables: supply chain (build) safety, safe closed-source libraries, verifies rewriter.



Fast (500+ MiB/s), **Simple** (~400 LoC), **Easy to fuzz/formally verify**

LFI Runtime

LFI runtime: virtualizes user mode



LFI runtime: Handles system calls that come from the sandbox.

Implements a **Linux API** (similar to WASI, QEMU-user, Blink emulator).

Many system calls are passed through with simple checks.

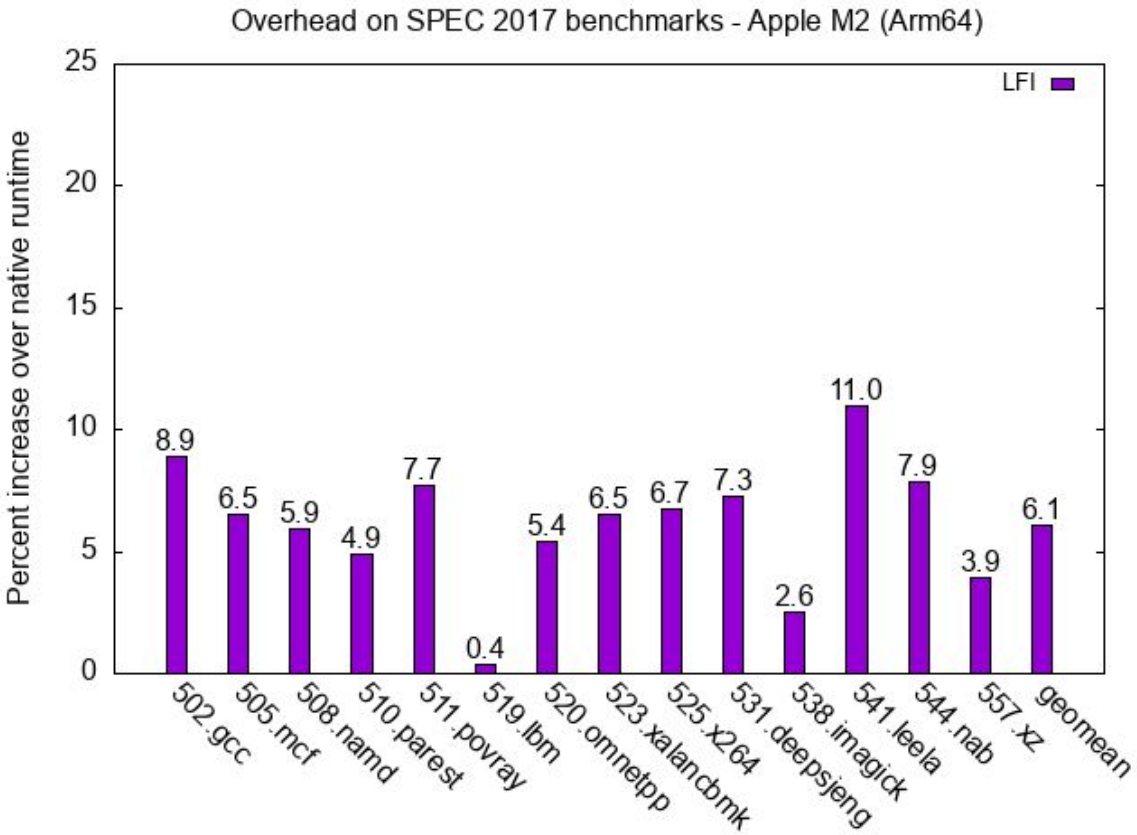
Putting it all together

```
$ git clone https://github.com/lua/lua
$ cd lua
$ make CC=aarch64-lfi-linux-musl-clang
...
$ lfi-run ./lua
Lua 5.5.0  Copyright (C) 1994-2025 Lua.org, PUC-Rio
> print('hello world')
hello world
```

```
0000000000023900 <lua_xmove>:
   23900: eb01001f    cmp     x0, x1
   23904: 54000360    b.eq    0x23970 <lua_xmove+0x70>
   23908: 8b2042b2    add     x18, x21, w0, uxtw
   2390c: f9400a48    ldr     x8, [x18, #0x10]
   23910: 7100045f    cmp     w2, #0x1
   23914: cb22d108    sub     x8, x8, w2, sxtw #4
   23918: f9000a48    str     x8, [x18, #0x10]
```

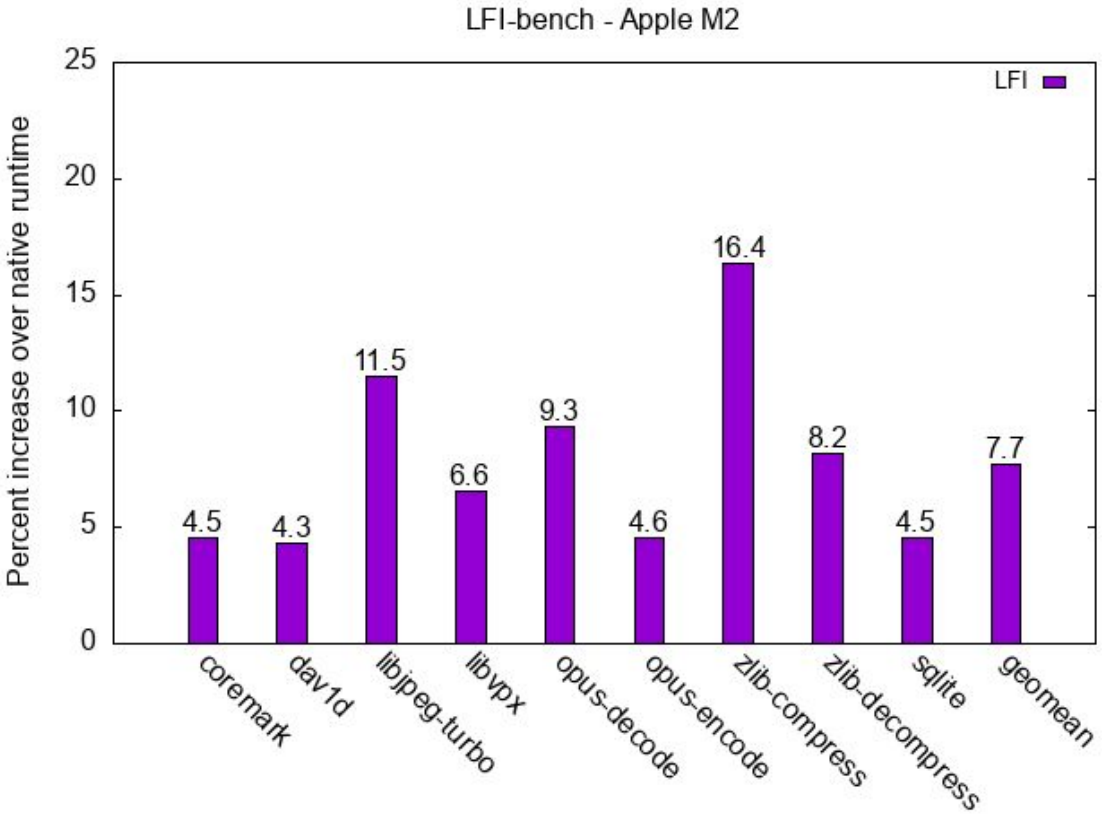
LFI Performance

LFI Overhead: SPEC 2017



Geometric mean: 6.1%

LFI Overhead: Codecs/Libraries



Geometric mean: 7.7%

LFI (Stores-Only) Sandboxing

Stores-only sandboxing: rewrite stores → integrity only.

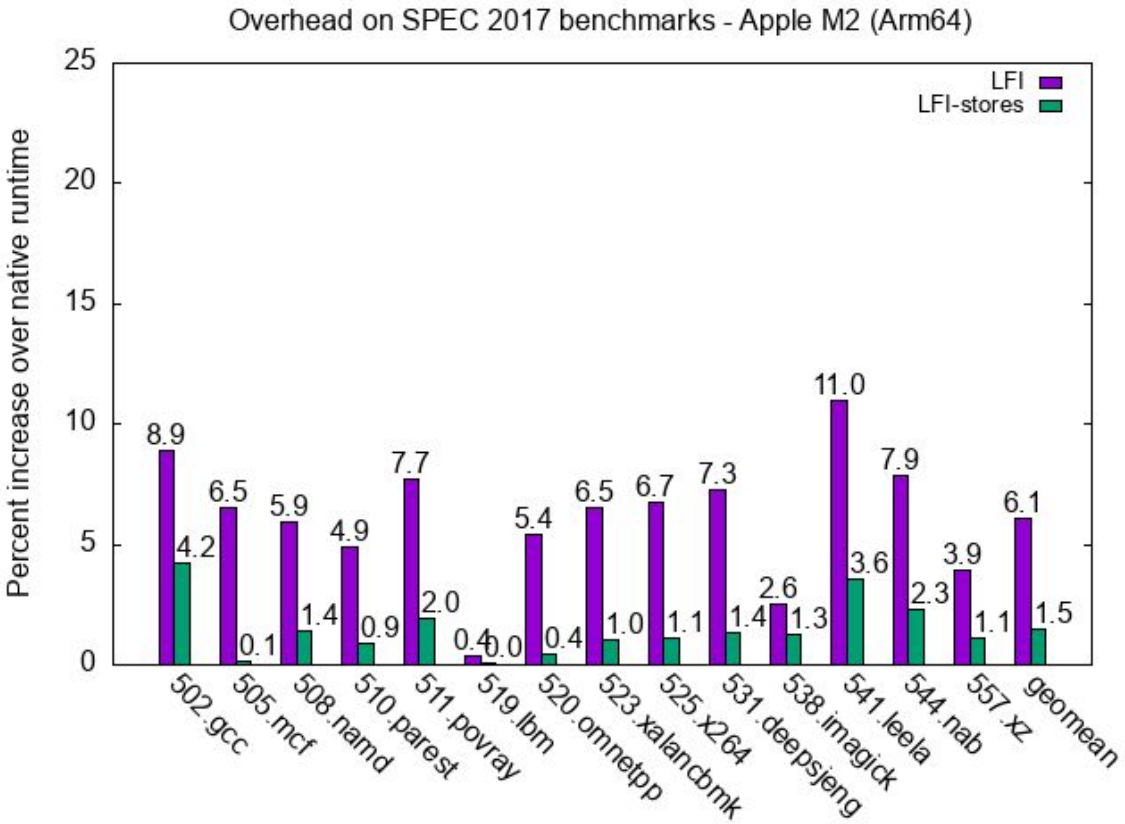
Untrusted code can read your data (but cannot write).

Reduces performance overhead: 1.5%

```
ldr x0, [x1]
```

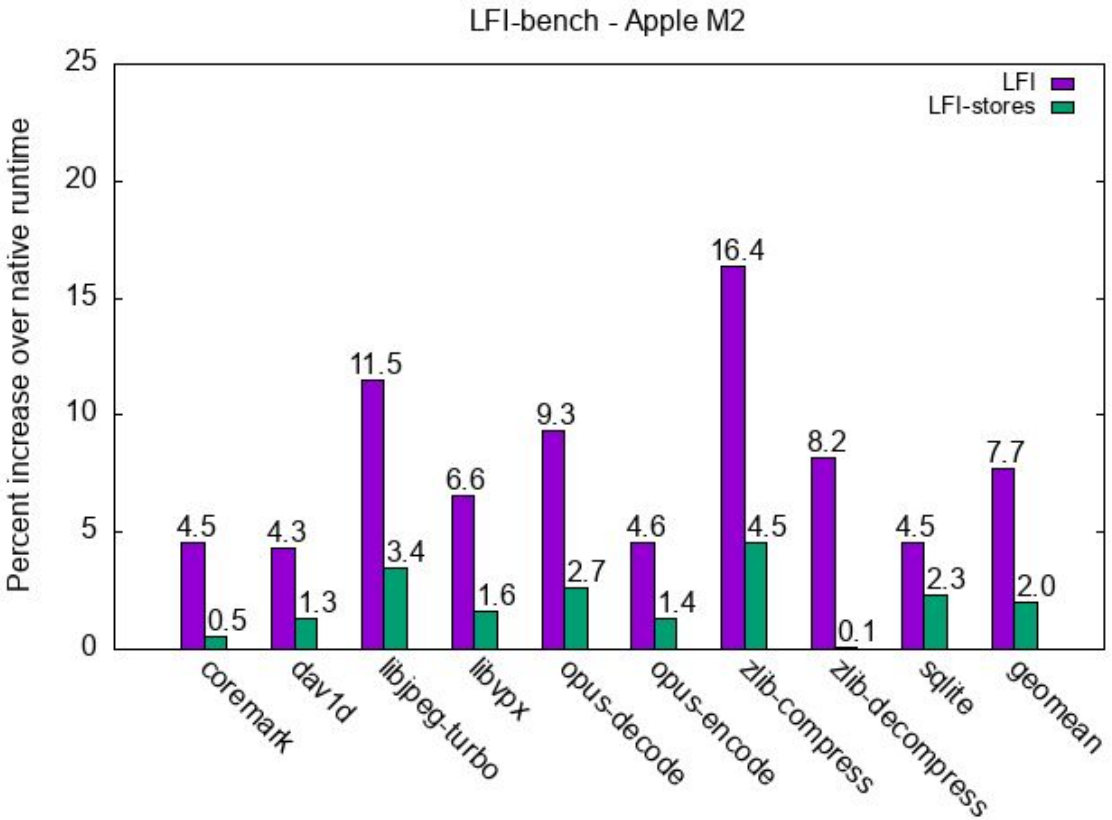
```
str x0, [x1]
```

LFI (Stores-Only) Overhead: SPEC 2017



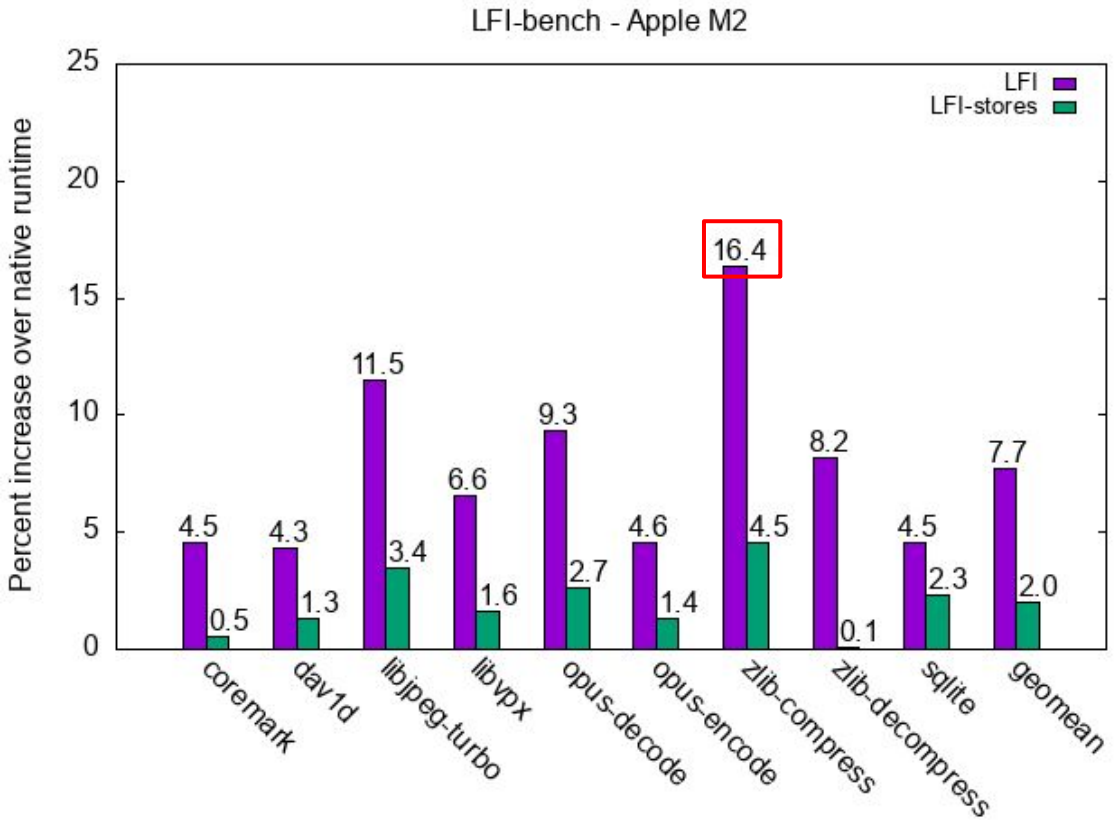
Geometric mean: 1.5%

LFI (Stores-Only) Overhead: Codecs/Libraries



Geometric mean: 2.0%

LFI (Stores-Only) Overhead: Codecs/Libraries



Geometric mean: 2.0%

Room for further optimizations!

Example: `zlib` with full LFI ⟶ 16% slowdown.

Small targeted optimization ⟶ 1% slowdown.

Approach: guard on pointer modifications, not use.

Further performance improvements:

Better **automatic** optimizations.

Manual optimization is also possible.

```
deflate_slow(deflate_state *s, int flush)

        ldr     w8, [x19, #5900]
        ldr     w10, [x19, #164]
        ldr     w11, [x19, #172]
        ldr     x12, [x19, #5888]
        add     w13, w8, #1
        mvn     w10, w10
        str     w13, [x19, #5900]
        ldr     w13, [x19, #180]
        add     w10, w11, w10
        strb    w10, [x12, x8]
        ldr     w8, [x19, #5900]
        ldr     x12, [x19, #5888]
```

Debuggable performance: Easy to identify difference between native and LFI code.

Benchmark: Sandbox Transitions

Microbenchmark: time to perform loop iteration that calls `add(int, int)` in a shared lib.

Platform	Instructions	Cycles	Time
Native	11	3	1.1ns
LFI	56	31	10.4ns

Arm Cortex X1

For reference (on Linux):
System call: 100-200ns
Context switch: 1,000-3,000ns

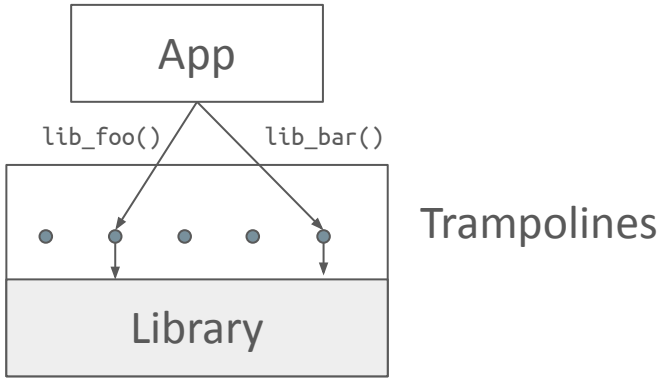
Library Sandboxing on Android

How do we sandbox a Library?

1. **Build** the library with the LFI compiler.
2. **Adapt** host application code.

   Allocate buffers in sandbox.

   Perform input validation.



Trampolines: transition into sandbox (save callee-saved regs, switch sp, x21, x18).

Case Study: Sandboxing libdav1d

dav1d is an AV1 decoder (used by YouTube through ExoPlayer)

Goals: fastest software decoder, cross-platform, small binary size.

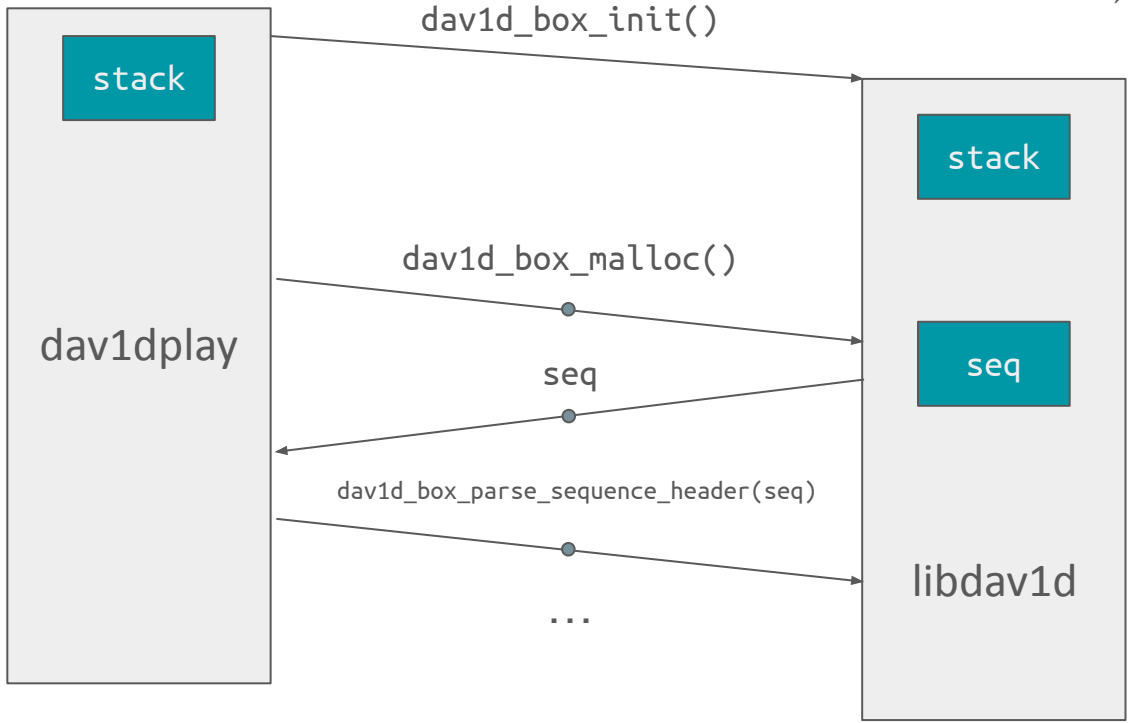
How? ***LOTS*** of hand-written assembly.

```
Totals grouped by language (dominant language first):
asm:          234001 (85.33%)
ansic:         40075 (14.61%)
sh:              161 (0.06%)
```

Performance Critical: “jank”, low end devices — good candidate for LFI!

Case Study: Sandboxing libdav1d

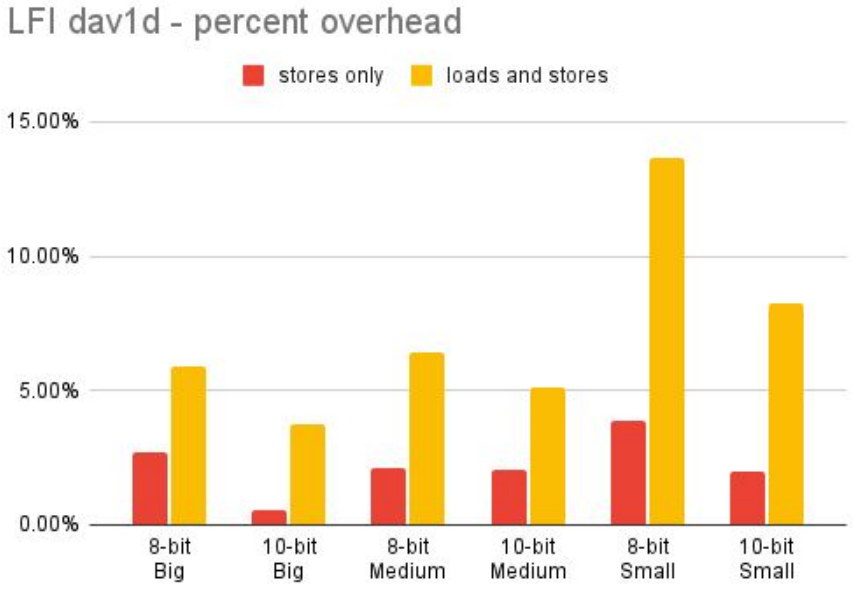
Loads sandbox executable via `mmap`
→ Linked with library

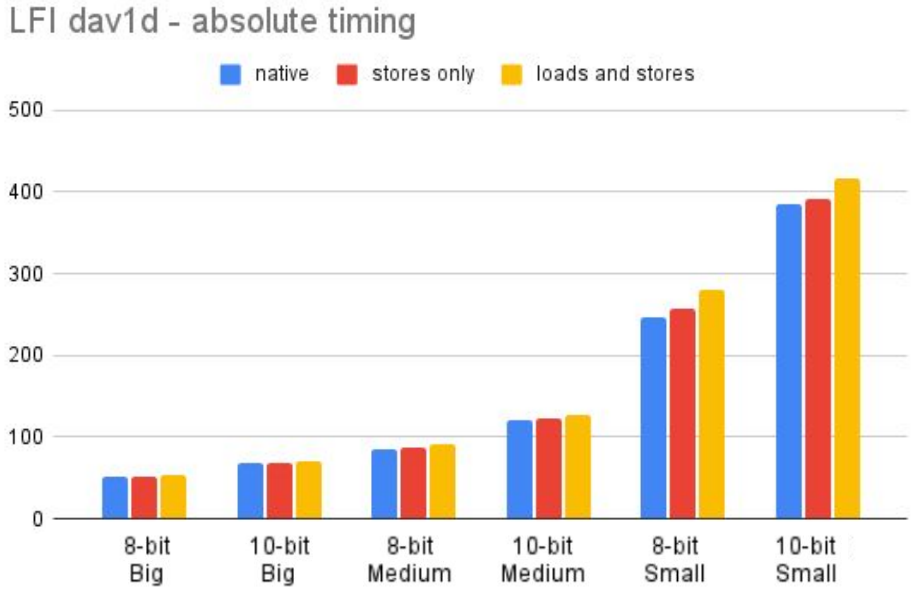


examples/dav1dplay.c +48 −32

`libdav1d`: no modifications!

LFI dav1d Performance on Pixel 9 Pro



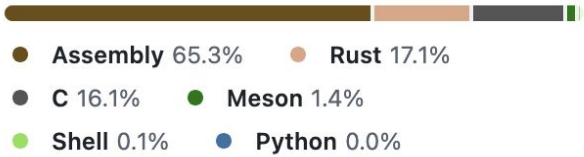


Comparison: Rewriting in Rust (rav1d)

memorysafety / **rav1d**

Security:
Still **LOTS** of assembly.
700+ unsafe blocks in Rust.

Languages



Performance:
11.5% overhead vs dav1d (on Pixel 8)
Engineering effort:
Hard fork – not maintained by dav1d team
- dav1d has new assembly not in rav1d

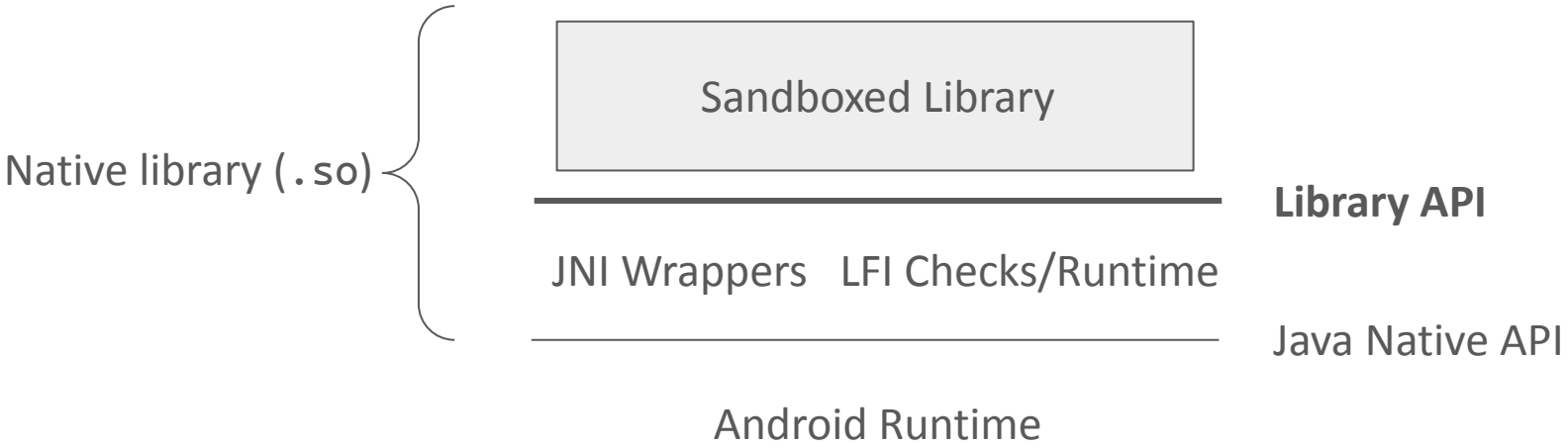
Significant effort:
- dav1d 2699 commits
- rav1d 8000+ commits (over a year)

Rust

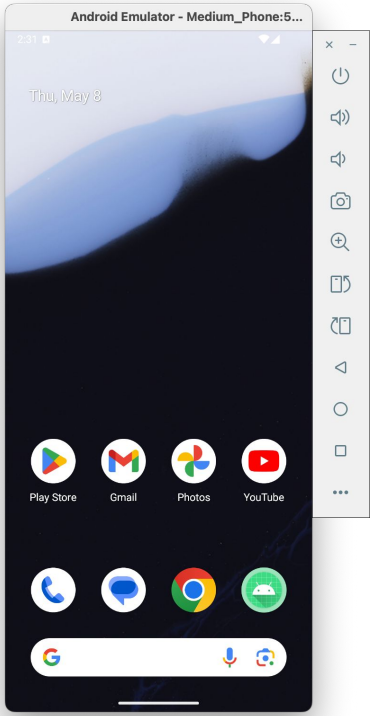
dav1d asm

Sandboxing an Android Native Library

Android apps are written in Java and access native code via Java Native Interface (JNI).



Demo: Android ExoPlayer with Sandboxed libvpx/libopus




```
--------- beginning of main
05-06 02:38:00.653  3370  3435 I liblfi  : new LFI sandbox at 7200000000
05-06 02:38:00.742  3370  3435 I liblfi  : new LFI sandbox at 6a00000000
05-06 02:38:00.874  3370  3435 I vpx_jni : vpx-lfi: initialized sandboxed decoder: 0x7220225230
05-06 02:38:00.879  3370  3452 I vpx_jni : vpx-lfi: (re)allocated sandboxed buffer 0x7220225440
05-06 02:38:00.880  3370  3452 I vpx_jni : vpx-lfi: (re)allocated sandboxed buffer 0x72203d7100
05-06 02:38:00.884  3370  3435 I opus_jni: opus-lfi: initialized sandboxed decoder: 0x6a20109030
05-06 02:38:00.890  3370  3453 I opus_jni: opus-lfi: (re)allocated sandboxed inputBuffer 0x6a200e5080
05-06 02:38:00.890  3370  3453 I opus_jni: opus-lfi: (re)allocated sandboxed outputBufferData 0x6a20133040
05-06 02:38:00.890  3370  3453 I opus_jni: opus-lfi: (re)allocated sandboxed inputBuffer 0x6a200e5250
05-06 02:38:00.891  3370  3453 I opus_jni: opus-lfi: (re)allocated sandboxed inputBuffer 0x6a200e5420
05-06 02:38:00.891  3370  3453 I opus_jni: opus-lfi: (re)allocated sandboxed inputBuffer 0x6a200e5420
```

Ongoing Work

Deploying LFI in AOSP

Android Clang/LLVM integration → upstream in Clang/LLVM

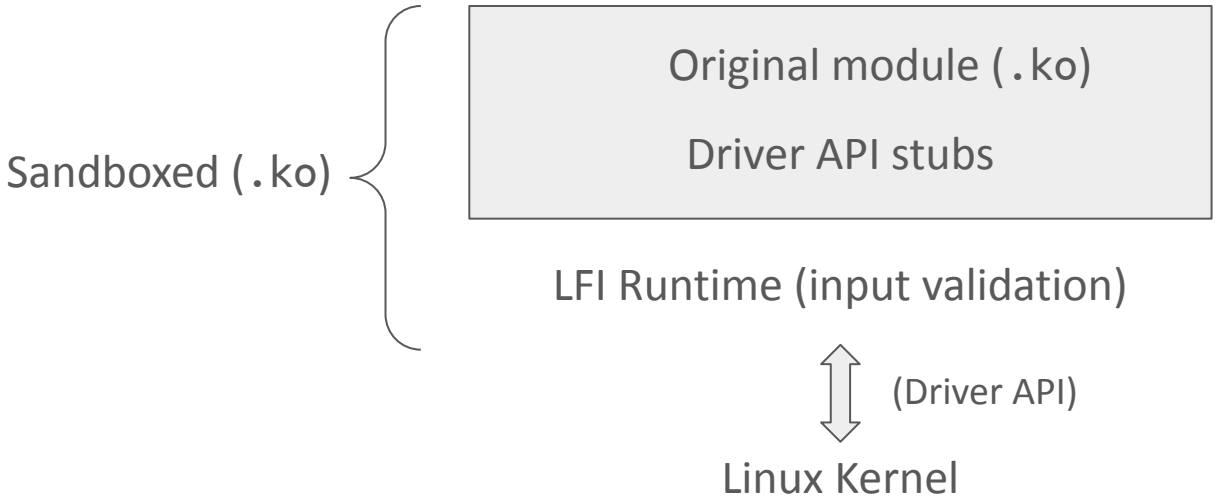
Sandboxing native libraries used by Java Libraries/Frameworks in AOSP

(Jetpack Media3, Android Media Framework, android.graphics, libcore, etc.)

Ship it

Isolating Third-Party Device Drivers

Problem: third-party device drivers → Local privilege escalation, RCE!



Sandboxing Native Libraries in Android Apps

Problem: Many Android apps include vulnerable native libraries.

Developer experience: Can we get to “push-button sandboxing”?

Little-to-no engineering effort required.

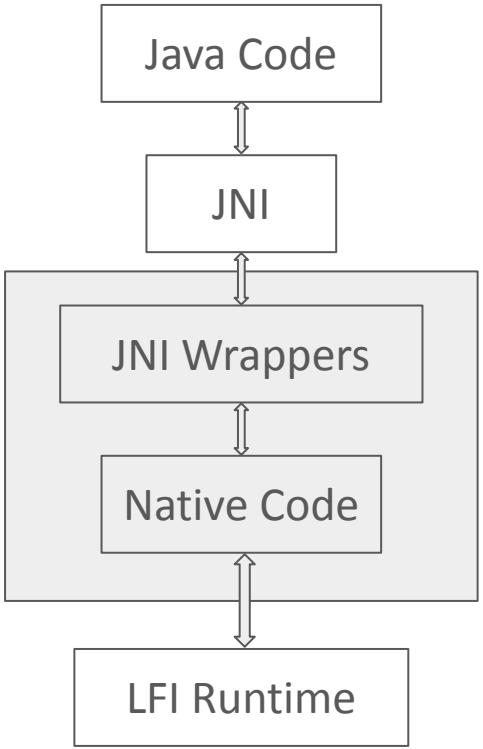
Key: apply sandboxing at the JNI boundary.

android studio

Android NDK

The Android NDK is a toolset that lets you implement parts of your app in native code, using languages such as C and C++. For certain types of apps, this can help you reuse code libraries written in those languages.

Google Play



Ask us about your use case!

C/C++ libraries

Device drivers

Thank you!