東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

The Challenge of Loading 4kB-Aligned ELFs on 16kB Devices

Kalesh Singh <kaleshsingh@google.com>
Juan Yescas <jyescas@google.com>

東京 2025
LINUX
PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

The 16KB Transition Delivers Significant Performance Gains

The move to 16kB pages delivers substantial, measurable performance and efficiency gains for mobile systems.

4x
reduction in page faults

0.8s
faster boot time

-3.16%
faster app launch time (average)

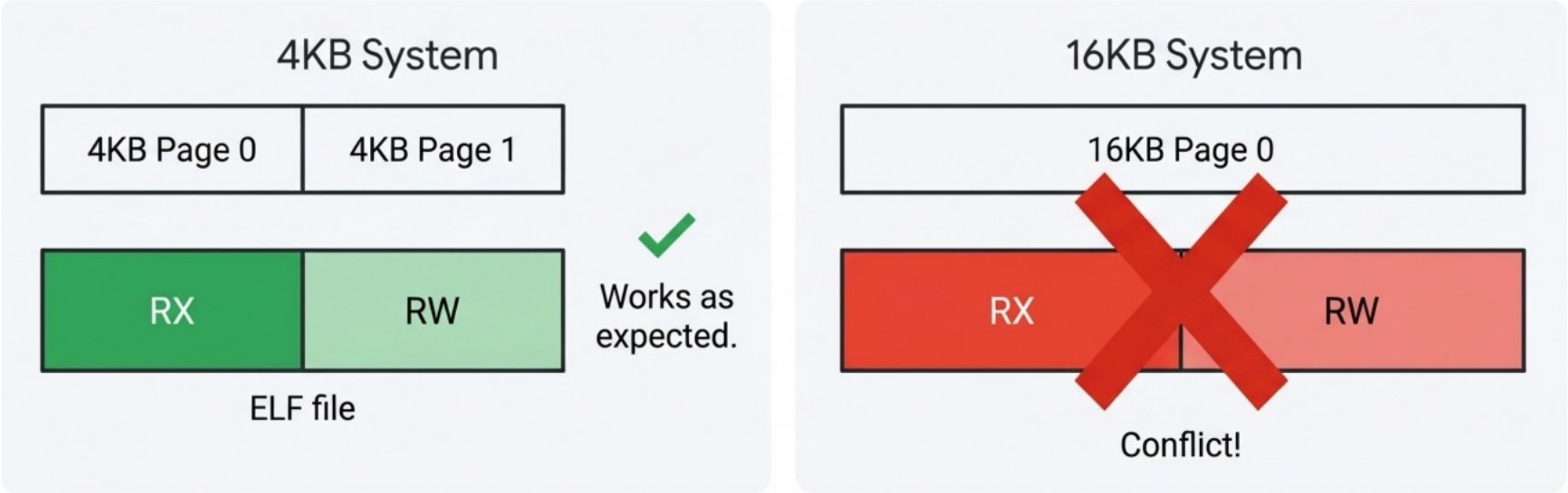
- -17% for Google Search
- -30% for Google News

4.56%
reduced power consumption (average)

Other industry benchmarks (Geekbench, GFXbench, Speedometer) show between **2%-10%** performance improvements.

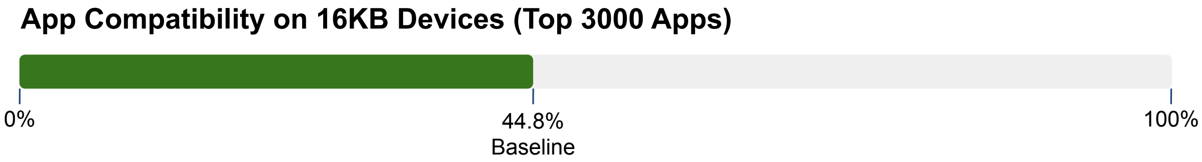
The Fundamental Conflict: 4kB Segments on a 16kB Page

Hardware memory permissions (Read, Write, Execute) are applied to entire pages. A single 16kB page can only have one set of permissions.



The loader must choose one permission for the entire 16kB page, either forcing an insecure RWX mapping or a crash.

東京 2025
Linux Plumbers Conference
TOKYO, JAPAN / DEC. 11-13, 2025



Our Strategy: User-Space Compatibility Loading

Goal

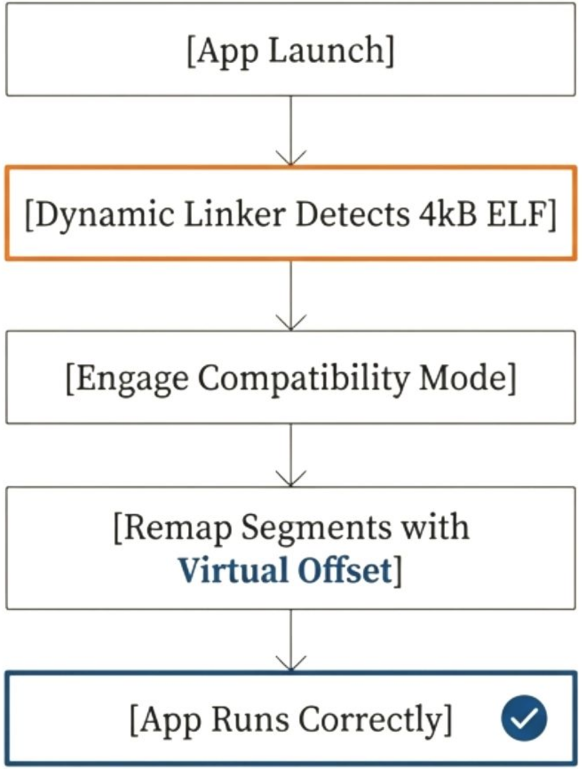
Enable unmodified 4kB-aligned ELF shared libraries to load and execute correctly on 16kB systems without creating insecure RWX mappings.

Mechanism

Implemented within the bionic dynamic linker, this compatibility layer intercepts the loading process for incompatible ELFs.

Core Technique

The dynamic linker consolidates adjacent ELF's segments (permissions) to reduce the layout to a single permission boundary and then aligns the final permission boundary with a 16kB page boundary.



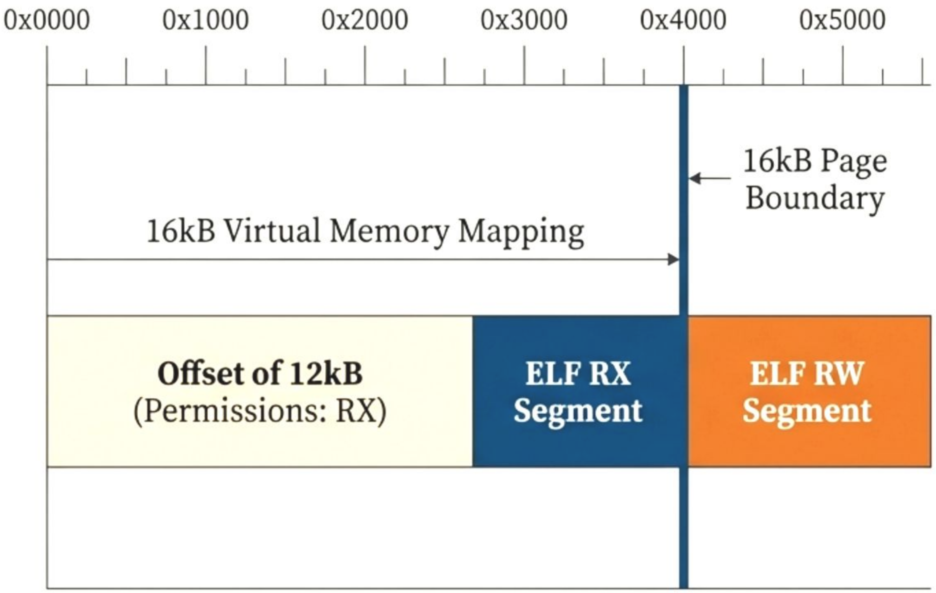
Solution Layer 1: The Simple Shift (RX RW Consolidation)

Regular ELF Layouts

Most non-custom ELFs can be reduced to a simple layout (one executable region followed by one writable region), we solve the alignment problem with a virtual offset.

Process Steps

1. Dynamic Linker creates a single anonymous memory mapping.
2. It copies the ELF contents into this mapping (using 4KB segment alignments).
3. It calculates an offset (e.g., 12kB) to push the permission boundary onto a 16kB page boundary.



The permission change from RX to RW now happens exactly on a hardware-enforced page boundary.

App Compatibility on 16KB Devices (Top 3000 Apps)

A Plot Twist: The Problem Is Not Just "Legacy" ELFs

Discovery

We found that even ELFs built with `-z max-page-size=16384` could fail to load.

Root Cause

A long-standing bug in GNU ld and LLVM lld linkers. The `PT_GNU_RELRO` segment's end was only guaranteed to be aligned to the common page size (4kB), not the specified max page size.

Impact

This creates the exact same permission conflict: a read-only `RELRO` region and a read-write data region land on the same 16kB page.

Ecosystem Scope: This wasn't a niche issue; it affected a huge proportion of apps in the ecosystem, even those built with with 16KB segment alignment.

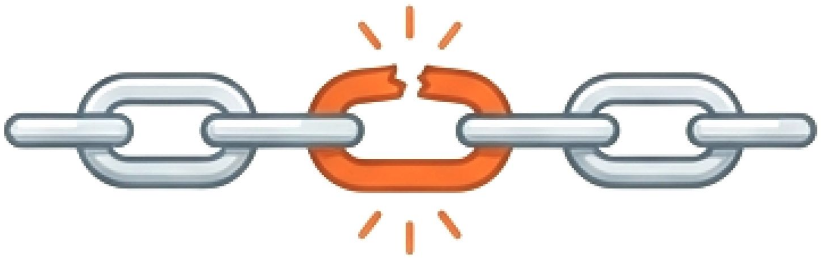
Sourceware Bugzilla: [Bug 28824]

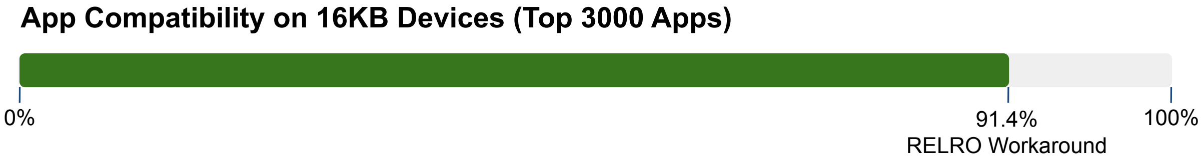
LLVM Issues: [#65002]

RELRO Alignment Workaround

Our Workaround

The bionic loader now explicitly detects this specific RELRO misalignment bug and forces the ELF into our compatibility mode, even if its segments appear 16kB-aligned.





Complex ELF Layouts & Sub-Page Protection

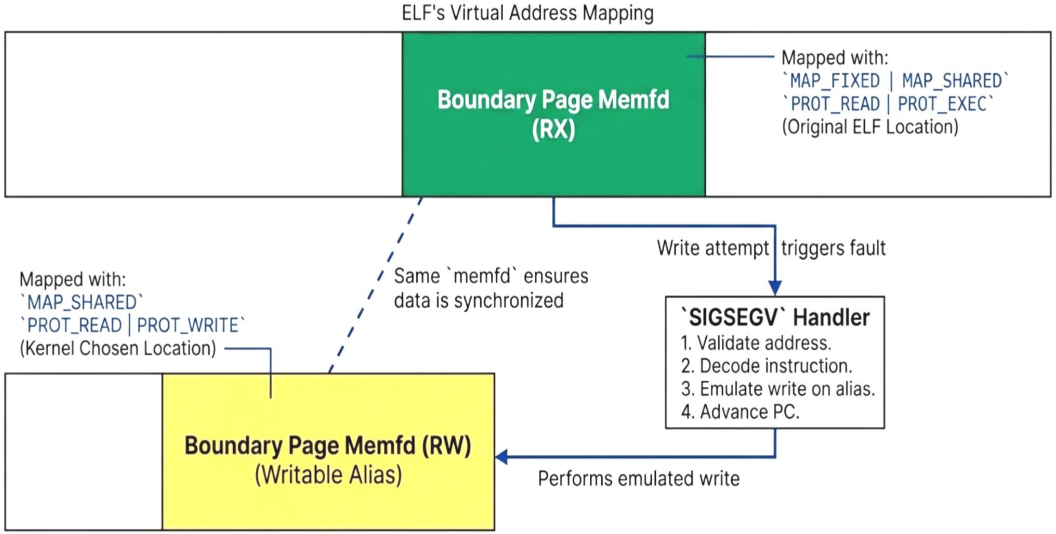
For complex ELF layouts that couldn't be simplified to a single RX|RW boundary, we designed a mechanism to emulate memory protection at a sub-page granularity.

Concept

Manage permissions at a sub-page level in software for complex layouts.

Mechanism

Use **`memfd`** to create dual mappings of 'boundary pages': a fixed **`RX`** mapping at the original address, and a writable **`RW`** alias elsewhere. A **`SIGSEGV`** handler catches writes to the **`RX`** page page and emulates them on the **`RW`** alias.



This technique is conceptually similar to how some JITs manage memory without persistent RWX pages.

The Pivot to A Pragmatic RWX Fallback

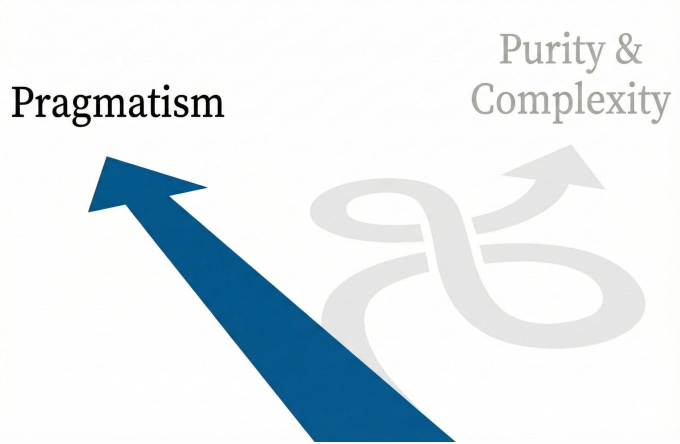
The Sub-Page Write Protection solution was ultimately shelved. While technically sophisticated, it introduced significant risk and complexity for limited real-world benefit.

Reasons for the Pivot

Immense Complexity: The `SIGSEGV` handler required a full `aarch64` instruction decoder (`VIXL`), creating a large and fragile maintenance burden.

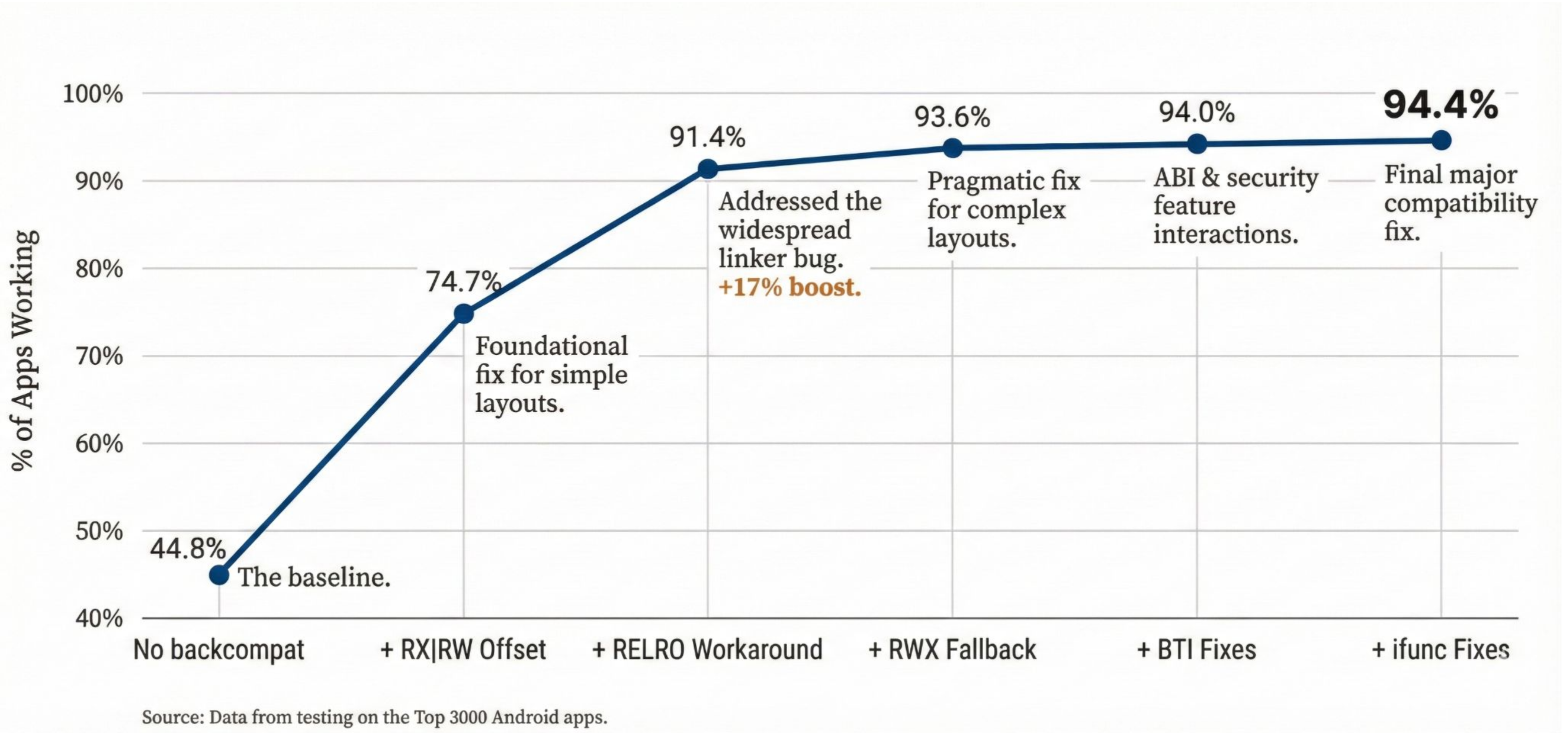
Performance Overhead: Microbenchmarks showed emulated writes on boundary pages were `~15` times slower than direct writes (`341 ns` vs `23 ns`).

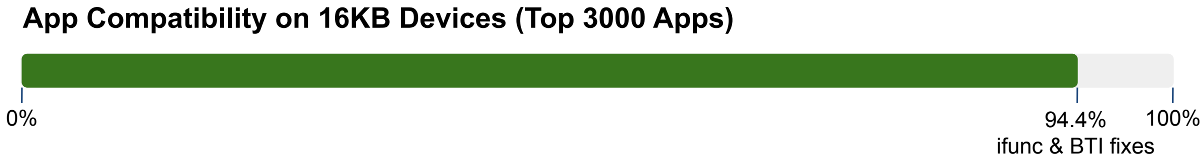
Limited Security Gain: The Android Security team concluded that since apps can already call `mprotect()` to create `RWX` mappings, and `ART JIT` already produces functionally equivalent memory, the complex `W^X` emulation did not significantly improve the security posture of the device.



App Compatibility on 16KB Devices (Top 3000 Apps)

App Compatibility Over Time





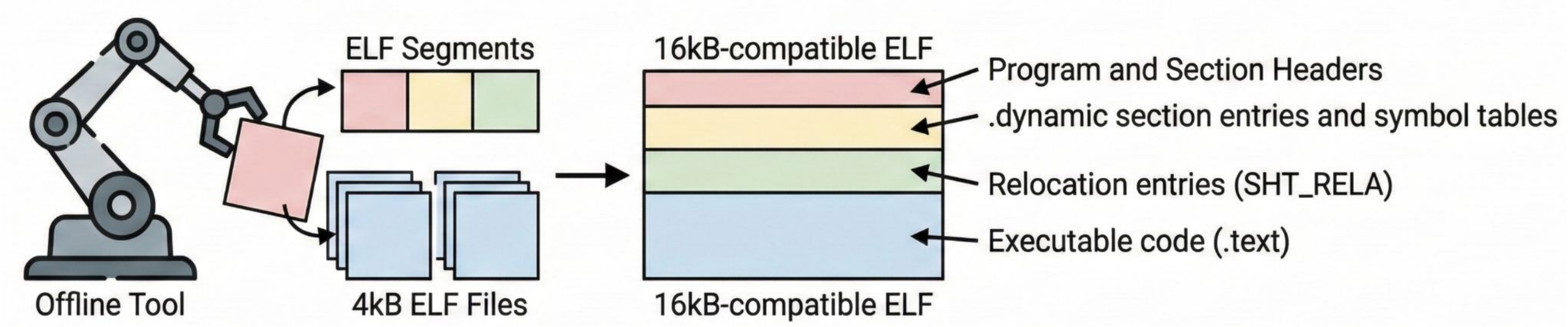
Alternative Path Explored: Offline ELF Patching

Before settling on a dynamic loading solution, we investigated an offline tool to patch 4kB ELF files to make them 16kB-compatible.

The Approach

The tool would re-layout ELF segments and meticulously patch all absolute and **PC-relative** addresses in:

- Program and Section Headers
- `.dynamic` section entries and symbol tables
- Relocation entries (`SHT_RELA`)
- Executable code (`.text`) sections



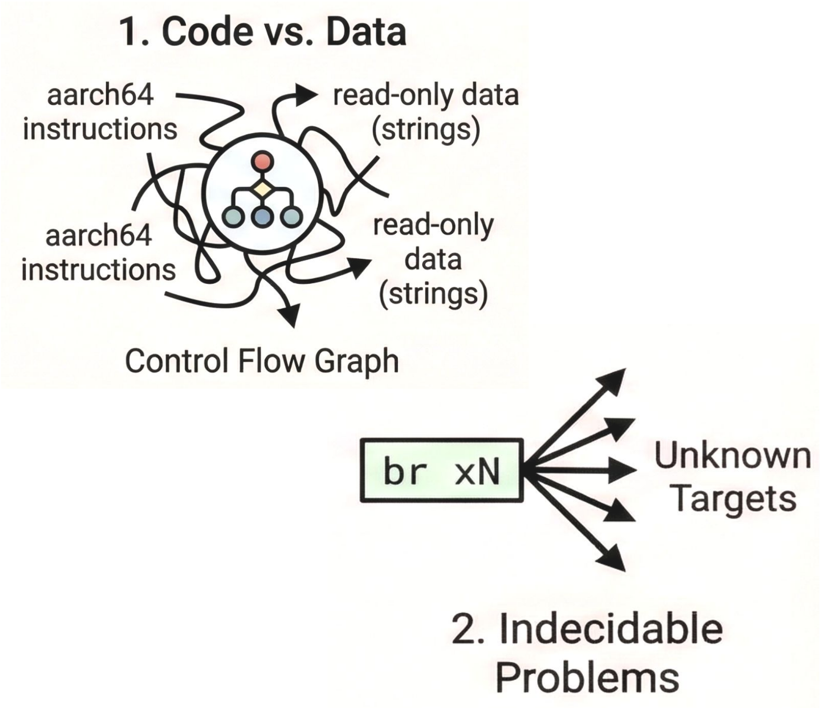
Alternative Path Explored: Offline ELF Patching

Before settling on a dynamic loading solution, we investigated an offline tool to patch 4kB ELF files to make them 16kB-compatible.

Why It Failed

The critical blocker was safely patching executable code.

- **Code vs. Data:** It's impossible to perfectly distinguish executable aarch64 instructions from interleaved read-only data (e.g., `strings placed via inline assembly`) without a flawless control flow graph.
    - **Intractable Problems:** Statically analyzing all possible targets of indirect branches (`br xN`), used for switch statements and vtables, is not a generally solvable problem.
- **Fragility:** The analysis was brittle and would break with new compiler versions or optimizations. The requirement for 100% correctness was unattainable.



The Final 6%: Outstanding Challenges

While we've achieved ~94% compatibility, the remaining failures are concentrated in a few difficult areas that our current solutions don't address

Two key areas:

1. **The Obfuscation Frontier:** Heavily obfuscated ELFs and custom loaders.

2. **Limitations of Compat Loading:** A few apps use the ELF mappings in ways that are inherently incompatible with the current compat loading approach.

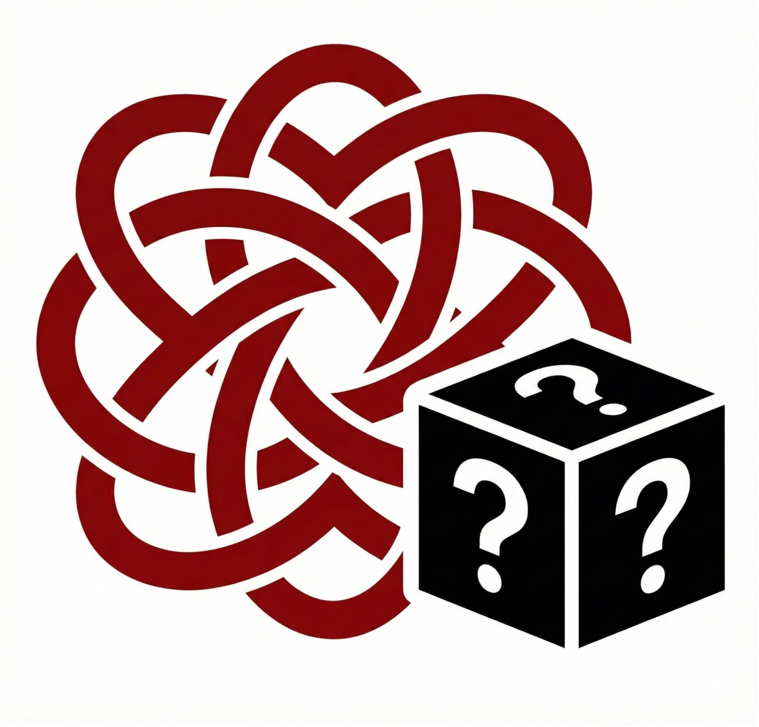
Challenge 1: The Obfuscation Frontier

A significant portion of the remaining incompatible apps use **custom loaders, packers, or obfuscation techniques**.

Why It's Difficult

- These techniques intentionally violate standard ELF conventions.
- They are **"black boxes"** that defy our static analysis and standard compatibility heuristics.

This is especially common in certain app ecosystems (e.g., apps in Asia).



Challenge 2: : Limitations of Compat Loading (Broken Assumptions)

Symptom

Apps that need to inspect their own ELF memory layout may fail.

Cause

Our compatibility mode uses an anonymous mapping and applies virtual memory offsets. The app's library is no longer a simple file-backed mapping at offset 0.

Core Problem

A classic mismatch between the Memory Management (MM) view and the Virtual File System (VFS) view of the process.

A Concrete Example

Some apps try to `openat()` a path to their own library file, likely to read data. In our compatibility mode, the library is loaded from anonymous mapping. The path looks like this:

```
[anon:16k:.../libname.so]
```

The `openat()` call fails with `ENOENT`.

Our attempts to fix this by exposing the real file path have not succeeded, suggesting the issue is more complex, possibly related to **file offsets being changed** by our compatibility layer.

Open Discussion & Call for Ideas

Kernel-side Solutions:

- Are there alternative kernel mechanisms we haven't considered for managing these misaligned ELFs?
- Is emulating the file path for anonymous mappings a viable strategy?

The Obfuscation Problem:

- Has anyone else encountered similar issues with obfuscated ELFs and file access?
- Are there better ways to provide a 'correct' file view to these applications?

Alternative Approaches:

- What have we missed?
- Are there other user-space or toolchain-based strategies that could complement or replace our current approach?
- Are there alternative approaches to achieve subpage protection:
  https://man7.org/linux/man-pages/man2/subpage_prot.2.html