Medovik: Enabling x86 Linux Compatibility in TrustZone

Lucian Mocan Michael Polinksi Friedrich Doku Peter Dinda

Northwestern | Plab

The Problem: Security vs. Compatibility is hard

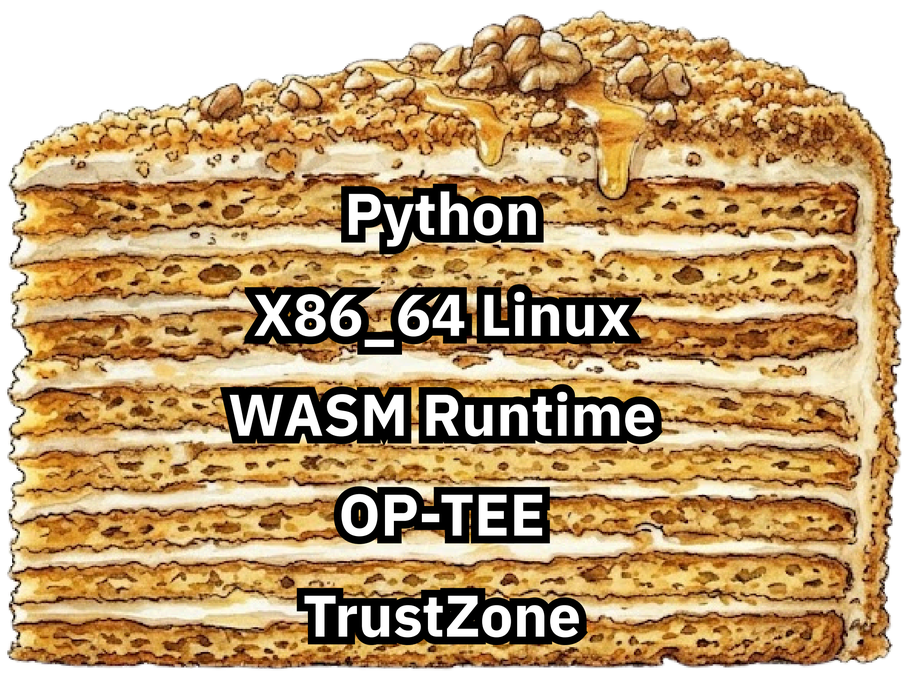
Trusted Execution Environments (TEEs) like ARM TrustZone offer strong hardware-backed isolation, but adopting them is difficult:

- **Unusual Programming Models:** Running programs in OP-TEE requires writing custom *Trusted Applications (TAs)*.
- TAs must navigate complex APIs and use Trusted Firmware-A (TF-A) simply to switch contexts to the Secure World.
- This completely breaks standard Linux toolchains, making porting or testing x86 codebases **prohibitively expensive**.

Can we run ***unmodified x86 binaries*** inside an ARM TEE?

The Medovik Sandbox: A Triple-Layered Approach

We introduce **Medovik**, a heavily nested sandbox that trades baseline performance for absolute compatibility.



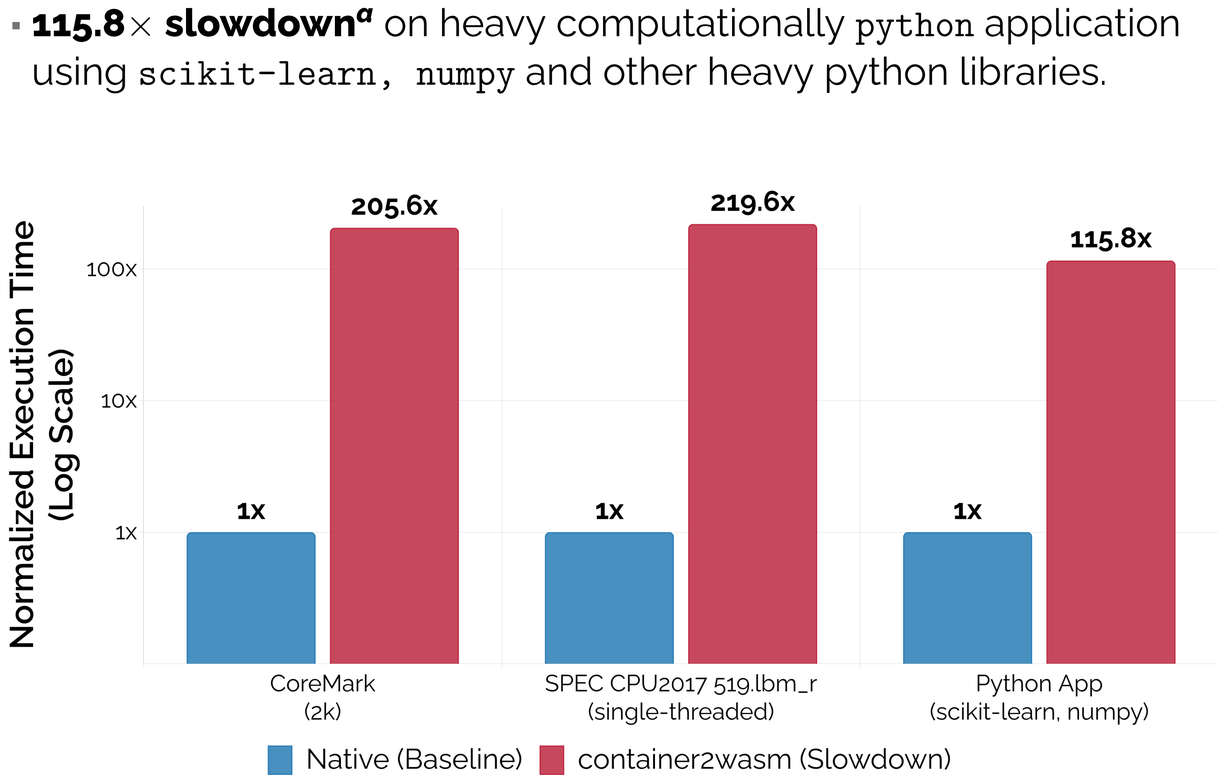
Layer 1: Host → Layer 2: WASM (WAMR) → Layer 3: x86 Emulator (Bochs)

This provides a complete Linux-like environment securely enclosed behind three boundaries.

The Performance Wall: Slowdown

Emulating x86 on top of WASM comes with severe overhead. On a Neoverse-N1 CPU, we observed:

- **205.6× slowdown** on CoreMark (2k).
- **219.6× slowdown** on SPEC CPU2017 `519.lbm_r` (single-threaded).
- **115.8× slowdown**[a] on heavy computationally `python` application using `scikit-learn, numpy` and other heavy python libraries.



Performance Overhead: Native vs. container2wasm. Execution times normalized to a native baseline (lower is better, logarithmic scale) demonstrating the severe overhead of emulating x86 on top of WASM on a Neoverse-N1 CPU.

[a] **ignoring python imports setup time**

The Solution: Punch-Through Hypercalls

To mitigate this massive overhead, we developed a punch-through hypercall mechanism. By intercepting specific instructions, we allow hot-paths to selectively bypass the emulation layers.

Case Study: LBM Benchmark

- Identified `LBM_performStreamCollideTRT` as the primary hot path.
- Offloaded just ∼**140 lines of code** (out of 1299) directly to AOT-compiled WebAssembly.

1. Guest Application (x86_64)

```
static inline void hypercall_lbm_collide(...) {
    uint32_t eax = 0xCAFEBABE; /* Magic SIG */

    /* Inject magic into EAX, invoke CPUID */
    __asm__ volatile (
        "cpuid"
        : "+a" (eax), "+b" (ebx), "+c" (ecx), "=d" (edx)
        : : "memory"
    );
}
```

↓

2. CPU Emulation (Bochs in WASM)

```
/* Emulation loop hits CPUID instruction */
if (EAX == 0xCAFEBABE) {
    /* 1. Sync guest memory */
    access_read_linear(src_laddr, ...);

    /* 2. Execute optimized function shortcut */
    LBM_performStreamCollideTRT(wasm_src, wasm_dst);

    /* 3. Update guest memory */
    access_write_linear(dst_laddr, ...);

    BX_NEXT_INSTR(i);
    return;
}
```

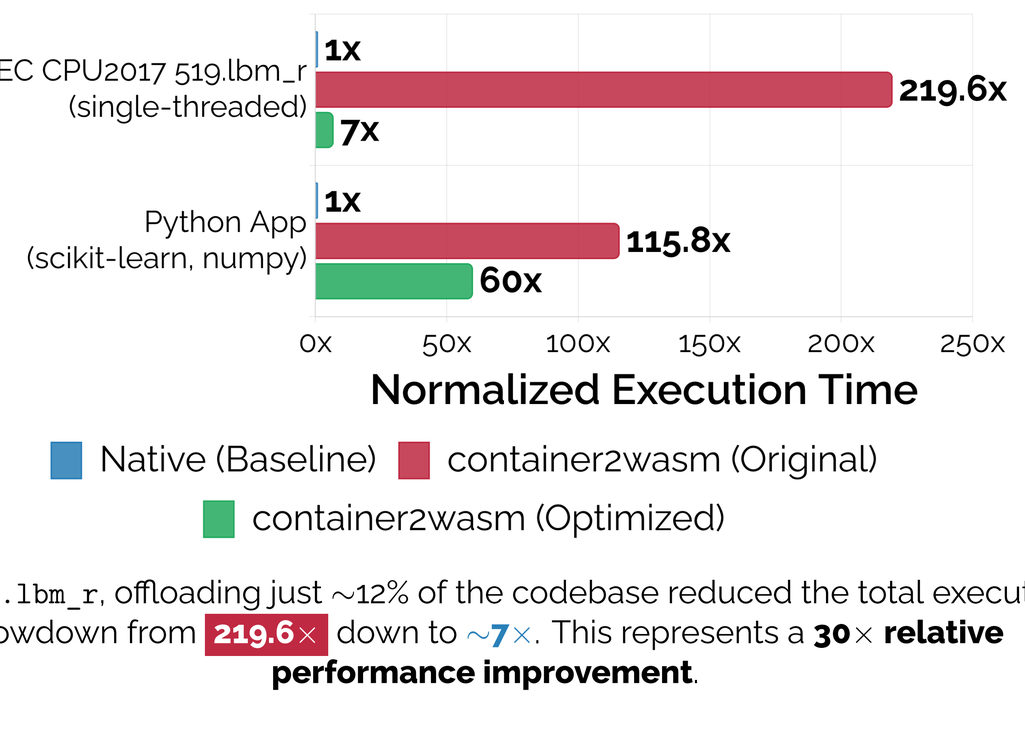
↓

3. Optimized Payload (WASM / Host)

```
/* Runs at near-native speed, avoiding
   instruction-by-instruction emulation */
void LBM_performStreamCollideTRT(...) {
    /* Heavy physics loop runs here... */
}
```

Instead of emulating complex physics calculations, the guest issues a hypercall. The emulator intercepts this and executes a near-native speed WebAssembly implementation of the routine.

Results: Massive Relative Gains



For `519.lbm_r`, offloading just ∼12% of the codebase reduced the total execution slowdown from **219.6×** down to ∼**7×**. This represents a **30× relative performance improvement**.

Takeaways

While Medovik remains slower than native execution, these results prove that selective hot-path offloading can make heavily nested sandboxes viable for compute-heavy tasks.