

NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-51

November 20, 2026

Medovik: Simplifying Performant Porting to TEEs by Adding Even More Layers to the Honey Cake

Lucian Mocan, Michael Polinski, Peter Dinda

Abstract

Applications are commonly developed on commodity x86_64 Linux systems using high-level languages, shared libraries, dynamic linking, device frameworks, and distribution-specific dependencies. These affordances are precisely what make porting such applications to trusted execution environments (TEEs), such as ARM TrustZone, difficult. TEE operating systems rarely provide the same language runtimes, library ecosystems, device interfaces, or process abstractions, and developers must also cross architecture boundaries and manually partition code and I/O across trusted and untrusted execution environments. The result is often an all-or-nothing porting effort before the application can run at all.

We propose the Medovik model, a compatibility-first approach to TEE porting. Medovik first preserves the original application environment by running the application and its dependency stack inside an emulated x86_64 Linux machine hosted by the TEE. This makes the original application stack immediately runnable on, e.g., ARM TrustZone, albeit slowly. The next step in the Medovik model is *profiling-guided punch-through:* the developer profiles the application in its ordinary environment and incrementally replaces selected performance-critical functions with optimized handlers in the TEE-hosted runtime. Our evaluation, which considers both benchmarks and a human motion-tracking and redaction application, shows that Medovik can run unmodified applications first and then recover substantial performance by porting only selected hot paths.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508

Keywords

software portability, trusted execution environments (TEEs), embedded systems, virtualization, emulation, WebAssembly (WASM), privacy, edge computing, Internet of Things (IoT)

Medovik: Simplifying Performant Porting to TEEs by Adding Even More Layers to the Honey Cake

Lucian Mocan
Northwestern University
Evanston, IL, USA

Michael Polinski
Northwestern University
Evanston, IL, USA

Peter Dinda
Northwestern University
Evanston, IL, USA

Abstract

Applications are commonly developed on commodity x86_64 Linux systems using high-level languages, shared libraries, dynamic linking, device frameworks, and distribution-specific dependencies. These affordances are precisely what make porting such applications to trusted execution environments (TEEs), such as ARM TrustZone, difficult. TEE operating systems rarely provide the same language runtimes, library ecosystems, device interfaces, or process abstractions, and developers must also cross architecture boundaries and manually partition code and I/O across trusted and untrusted execution environments. The result is often an all-or-nothing porting effort before the application can run at all.

We propose the Medovik model, a compatibility-first approach to TEE porting. Medovik first preserves the original application environment by running the application and its dependency stack inside an emulated x86_64 Linux machine hosted by the TEE. This makes the original application stack immediately runnable on, e.g., ARM TrustZone, albeit slowly. The next step in the Medovik model is *profiling-guided punch-through*[1]: the developer profiles the application in its ordinary environment and incrementally replaces selected performance-critical functions with optimized handlers in the TEE-hosted runtime. Our evaluation, which considers both benchmarks and a human motion-tracking and redaction application, shows that Medovik can run unmodified applications first and then recover substantial performance by porting only selected hot paths.

Keywords: software portability, trusted execution environments (TEEs), embedded systems, virtualization, emulation, WebAssembly (WASM), privacy, edge computing, Internet of Things (IoT)

1 Introduction

Edge and IoT devices are a natural next target for confidential computing: they increasingly process privacy-critical user data locally, but their TEEs usually expose restrictive execution environments rather than the full-VM abstractions available in cloud confidential computing. Hardware-based Trusted Execution Environments (TEEs), such as ARM TrustZone [3] and Intel SGX [18], are the standard architectural solutions for this, attempting to provide hardware isolation from the rest of the OS. However, using them to deploy modern, dependency-heavy applications to edge/IoT devices presents a severe engineering challenge.

Developers typically build and test code on standard Linux setups running on x86_64 hardware platforms, but TEE operating systems are fundamentally incompatible with these setups. A recent systematic review highlights that the vast diversity in hardware architectures, SDKs, and bespoke APIs makes application porting and cross-platform development overwhelmingly complex [17]. This fragmentation breaks standard toolchains, and a recent empirical study demonstrates that the steep learning curve often drives developers to bypass safe APIs or introduce insecure coding practices violating the trust boundary [21].

While newer cloud-focused extensions like Intel TDX [15] and AMD SEV [1] allow for lifting entire x86_64 virtual machines into secure enclaves, edge and IoT devices are typically ARM-based. Although the ARMv9 Realm Management Extension (RME) [2] was announced in 2021 to provide similar virtualization abstractions, it is not yet an option in commodity edge/IoT hardware [5]. Even with RME functionality, plenty of application code still starts x86_64-native. Thus, moving an existing application into an edge/IoT TEE requires a complex porting effort. Even ordinary I/O becomes a porting problem: TEE code cannot freely use host OS services or physical peripherals, so interactions with the outside world must be mediated across the TEE boundary [8, 29]. Manual partitioning is notoriously difficult to implement securely, and poor implementations frequently lead to massive performance overheads and system availability bottlenecks [14].

We propose the Medovik model, an intentionally layered approach to TEE porting. Instead of requiring a complete port before the application can run, Medovik first preserves the original x86_64 Linux application and dependency environment inside compatibility layers hosted by the target TEE. This compatibility-first step adds overhead, but changes the developer's first milestone from performance to execution. The developer can then use *profiling-guided punch-through*: profile the application in its ordinary environment, identify hot paths, and replace only those paths with optimized handlers that bypass the slow layers. Figure 1 gives a high-level view of traditional all-or-nothing porting and the Medovik model; §3 describes the model and prototype in detail.

Our overall goal is to ease the porting of applications to

[1] Medovik is an Eastern European honey cake made of many layers; in our model, those layers can be selectively skewered by punch-through.

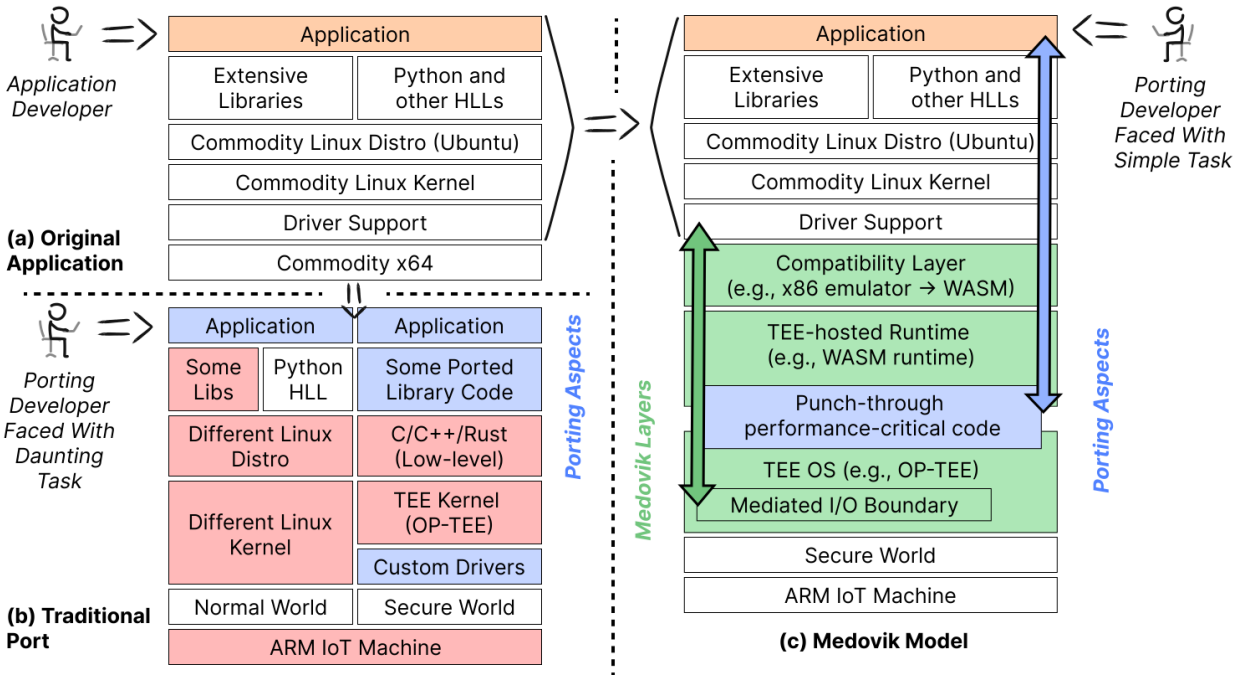


Figure 1. Medovik simplifies porting to TEEs. (a) Applications are built for commodity x86_64/Linux stacks. (b) Traditional TEE porting forces developers to split or replace application, runtime, OS, and I/O layers across Normal and Secure Worlds. (c) Medovik preserves the original stack inside TEE-hosted compatibility layers, then uses profiling-guided punch-through for selected hot paths.

TEE environments on Edge/IoT devices. Our specific contributions are as follows.

- We lay out the challenges involved in porting applications in the current all-or-nothing porting model (§2), using a human motion tracking/redaction system as an example.
- We describe our proposed alternative Medovik porting model in detail, illustrating its compatibility-first and profiling-guided punch-through elements (§3).
- We describe the design of our Medovik prototype, which implements compatibility-first execution through a nested sandbox architecture that enables the immediate deployment of unmodified x86_64 Linux applications in ARM TrustZone. The prototype implements punch-through via an emulator-bypass hypercall mechanism that allows hot functions to be ported incrementally to native WASM (§3).
- We evaluate our Medovik prototype on benchmark suites and on the human motion tracking/redaction system. The unmodified code runs via the compatibility-first path with ~100x slowdown, while profiling-guided punch-through for just the 1-2 hottest functions (just hundreds of lines) reduces this to ~1.15-35x (§4).

2 Challenges of Porting To TEEs

Porting existing applications into edge/IoT Trusted Execution Environments (TEEs), such as ARM TrustZone can be overwhelming even when they already target embedded hardware. The Medovik effort grew out of our attempt to port ERASE [27, 28], a human motion tracking and redaction application, into such an environment with significant cooperation from its developers. We have encountered similar issues when porting other such tools, and a distributed query framework, to TEEs.

The recurring problem is that application developers make reasonable assumptions about the execution environment: available languages, libraries, device interfaces, and foundational OS features. In a TEE, these assumptions often fail simultaneously. ERASE provides a concrete example of this.

2.1 ERASE Porting Effort

The goal of ERASE [27, 28] is to achieve privacy-preserving live video redaction on low-end edge systems. Humans are tracked through a venue with cameras and ultra-wideband localization. A novel and fast signal processing (non-AI) algorithm computes a bounding box around each human that can be seen in each frame. For each human who has not opted into image capture, the application redacts all the pixels in the bounding box. Tracking and redaction occurs only on the edge device, so that no data for anyone who has not opted in ever leaves the edge device. Over 60 frames/second for one person can be achieved on a ROCKPro64 SBC (ARM Cortex A72), dropping to 20 frames/second for five people, performing about 10× better than previous work. We use the same hardware in this work.[2]

ERASE was published as an x86_64 Ubuntu Docker image, while our target ROCKPro64 is ARM-based and initially executes code in the "Normal World". Our goal was to run

[2]The ROCKPro64 SBC is a low-cost edge/IoT-class development platform.

ERASE in TrustZone's "Secure World", where attestation could verify the redaction code and hardware partitioning could provide a direct camera-to-"Secure World" path. Anyone trusting the attestation process could then know their video would pass through ERASE before leaving the device.

Systems environment. We explored several Secure World environments for ERASE, but each still led to the traditional-porting problem in Figure 1(b). OP-TEE [25] exposes Secure World code mainly as RPC handlers from Normal World Linux, a poor fit for ERASE as a long-lived process. KaTZe [13], a Secure World port of the Kitten lightweight kernel [23], supports process-like workloads and limited Linux system calls; split-core KaTZe/Linux can also route selected requests across worlds. In all cases, the developer must still choose which application, library, OS, and I/O pieces are rewritten, replaced, or split across the trust boundary.

2.2 Language Assumptions

There is typically a fundamental disconnect between the language environment of application designers and that of TEE toolchain engineers.

In the case of ERASE, the authors used the best languages and libraries for signal processing. This was initially Matlab, and then Python combined with NumPy/SciPy. For video I/O from both devices and files, they simply used OpenCV with its Python bindings. This made the ERASE code superficially compact, but created deep dependencies. This conflicts with the traditional porting options, which expect embedded systems-style code: compact, written in a low-level language like C, C++, or Rust, and self-contained save an explicit hardware abstraction layer (HAL) and OS abstraction layer (OSAL). Of course, the authors could have re-written their code to fit this model, but why should they? The performance level is already sufficient, and is likely to improve with its dependencies. Likewise, it would be gratuitous to re-engineer a complex camera device driver that OpenCV already is already appropriately controlling.

When we ourselves attempted to port ERASE down to raw C with minimal dependencies and a strict HAL/OSAL, we were left with a range of issues that required frequent interaction with the ERASE authors. We also faced a range of teething pains with camera drivers within the TEE environment that largely stemmed from the fact that the complex driver interface is intended to have a friendly abstraction layer on top of it... like OpenCV.

2.3 Environment Assumptions

Application developers make assumptions about the environment of their platform, including their operating system and its packaged libraries and frameworks. For instance, the ERASE designers made initial implicit assumptions that the program would run on top of modern Ubuntu on x86_64. Essentially all needed tools were an `apt install` away.

TEEs complicate this further. Of the three potential TEE operating systems we considered to host the application, one did not have the notion of processes. They had at best partial implementations of the Linux system call interface. In the case of KaTZe, implementing the remaining needed system calls correctly proved difficult.

Additionally, regardless of language, shared libraries are often non-optional. Libraries like NumPy/SciPy and OpenCV are built as system ABI-based shared libraries and written in low-level languages for performance. For complex libraries, there may not even be an option to create a static version. And many higher-level language runtimes and frameworks assume the ability to do a `dlopen()`, obviating static libraries altogether. Shared libraries are non-trivial to implement, and are notably absent from TEE OS implementations (or insufficiently supported, as is the case for the KaTZe kernel).

Finally, device drivers remain lacking. As a result, for ERASE, video data would need to flow through the insecure Normal World, or a custom driver for KaTZe would be needed. Porting a Linux kernel driver – and keeping it compatible with OpenCV and its dependencies – would require significant effort and knowledge of the hardware.

3 Medovik Model and Prototype

The Medovik model is a compatibility-first approach to porting existing applications to constrained TEE environments. Rather than requiring developers to port an entire application, operating-system interface, runtime, and dependency stack before execution is possible, Medovik first preserves the original software/hardware environment and runs it inside a compatibility substrate hosted by the TEE. This changes the first porting milestone from performance to execution: the application should run before the developer is forced to decide which pieces must be rewritten.

Assumptions. The model makes three assumptions. First, the application can be packaged together with the operating system and library state it expects (e.g., a container-derived image). Second, the target TEE can host a small execution substrate capable of running a compatibility layer, such as a WebAssembly runtime, emulator, or similar abstraction layer. Third, the compatibility layer exposes a controlled mechanism for redirecting selected guest operations into optimized code without requiring the rest of the application to be modified (e.g., a host-function interface).

Core elements. Medovik has two core elements. It starts with *compatibility-first execution*: the original application binary and its dependency environment are preserved as much as possible, even if doing so requires additional layers of emulation or interpretation that introduces slowdown. The second element is *profiling-guided punch-through*: developers profile the application in its ordinary execution environment, identify hot paths that are expected to remain

performance-critical after deployment, and incrementally port only those paths through explicit punch-through mechanisms. This allows the developer to focus their attention on a (hopefully) tiny subset of the codebase to ameliorate much of the slowdown. It is only the porting of that tiny part of the codebase that experiences the challenges laid out in §2. Furthermore, some of the challenges may not apply to this profile-determined part of the codebase

Importantly, the Medovik model is not tied to a specific emulator, WebAssembly runtime, hypercalls, or TEE OS. A prototype might instantiate the model using an x86_64 emulator compiled to WebAssembly and hosted by an ARM TrustZone runtime, as ours does, but other implementations could use different emulators, different TEE substrates, or different mechanisms for intercepting and replacing hot paths. Similarly, I/O and device access can be treated as additional compatibility boundaries mediated across the TEE boundary according to the deployment threat model. The essential property is that the developer can run the unmodified application first and then recover performance incrementally, without porting the full application and dependency closure.

3.1 Prototype

Our prototype instantiates the model for a deliberately difficult target: running unmodified x86_64 Linux applications inside an ARM TrustZone TEE. The implementation has five nested layers and a hypercall mechanism for punch-through. We implement the prototype on a ROCKPro64 board using OP-TEE [22] 4.9.0 as the TEE OS, WAMR [7] as the WebAssembly runtime, Bochs [16] as the x86_64 emulator, and Debian-based Docker images as the source application environments.

Prototype layers. The prototype layers are organized as follows, from the original application downward to the TEE:

- *Application.* The input is an original Linux x86_64 application compiled for some target Linux distribution.
- *Linux kernel and OCI container.* The application is placed in a Docker [10] container with its dependencies. This container becomes the root filesystem of an OS image.
- *x86_64 emulator.* An emulator (e.g. Bochs or QEMU) boots the image on an emulated x86_64 platform. The emulator itself is compiled to (and may even codegen) WebAssembly.
- *WebAssembly runtime.* A TEE-hosted WebAssembly runtime executes the emulator and provides the host interface required by it. Our prototype's runtime is WAMR with support for ahead-of-time (AOT) compiled modules.
- *TEE OS.* The runtime is compiled against a TEE OS, which provides the platform interface of system calls needed to execute inside the Secure World.

The cost of these layers is substantial overhead: application instructions execute through an emulated x86_64 machine, the emulator runs as WebAssembly, and the WebAssembly runtime executes inside a constrained TEE.

The remainder of this section follows the resulting porting workflow: constructing the Docker-derived guest image, executing it through the nested layers, profiling the original application to select hot paths, and adding punch-through handlers for those paths.

Leveraging containers. The first two layers are produced by packaging the application with its expected Linux environment. In our prototype, this environment is a Debian Docker container image with the necessary libraries and dependencies added. We build a bootable image containing the Linux kernel and this Docker container as its root filesystem. Ideally, in the Medovik stack, this is the only task undertaken by the end developer.

x86_64 emulation in WebAssembly. The guest image is packaged with Bochs [16] into a WebAssembly executable. Bochs is compiled to WebAssembly, but the machine it emulates remains x86_64. Existing open-source tools such as `container2wasm` [9] perform this style of packaging by taking a Docker image as input and adding a Linux kernel and the Bochs emulator to produce a WebAssembly executable that effectively runs a full guest OS and application.

We extend Bochs with hypercalls for punch-through. At a high level, Bochs intercepts selected guest-side instruction patterns and dispatches to native C handlers added to the Bochs codebase. Because these handlers are compiled with Bochs, they become part of the same WebAssembly module as the emulator. The hypercall path is described later.

WebAssembly runtime in TrustZone. The WebAssembly runtime makes the emulator executable inside the TEE and provides the host interface expected by the emulator module. We use WAMR because WATZ [19] demonstrates a WAMR port to TrustZone on top of OP-TEE, giving us a concrete Secure World embedding. WAMR also targets embedded deployments and supports ahead-of-time compilation through `wamrc` [6].

AOT compilation is important for punch-through. The hypercall handlers are first optimized when Bochs is compiled to WebAssembly, and can then be optimized again when WAMR compiles WebAssembly bytecode to machine code. As reported by WATZ and confirmed by our measurements in Figure 2, AOT-compiled WebAssembly can run with low overhead relative to native execution. In some of our benchmarks, `wamrc`'s optimization passes and vectorization even make AOT-compiled WebAssembly faster than our native baseline; we discuss these cases in §4. Thus, WAMR hosts both the full emulator and the optimized hot-path handlers used to bypass it.

Profiling. After the application runs in the compatibility-first configuration, the developer identifies punch-through candidates by profiling the application in its ordinary execution environment. This keeps profiling outside the slow TEE-backed stack and lets the developer use any standard

profiler available for the original application. In our experiments, we use `gprof` [12] for C and C++ programs and `py-spy` [4] for Python programs. The profile need not predict every source of emulation overhead; it only needs to identify functions whose computation is important enough to make them plausible punch-through candidates.

Punch-through handlers. Our prototype implements punch-through by redirecting selected guest-side call sites. The developer replaces a call to a hot function with a small shim that invokes the x86_64 `cpuid` instruction with a custom Medovik leaf value placed into `EAX`. When the guest executes `cpuid`, Bochs inspects the emulated `EAX` register, and if the Medovik leaf is found, it dispatches the call to the native C handler that has been added to its codebase. Because these handlers are compiled with Bochs, they become part of the same WebAssembly module as the emulator.

The guest-side redirection depends on the source language. For C and C++ programs, `LD_PRELOAD` suffices to switch to the shim function. For Python applications, such as the ERASE pipeline described in §2, we use the Python C API [26] to expose replacement module functions that the application can import and call as hypercall-backed operations. This uses Python's ordinary function and object rebinding mechanisms which do not require rewriting the application.

The corresponding handler executes as WebAssembly code inside the TEE-hosted runtime instead of executing the original hot function through full x86_64 emulation. When the handler needs guest data, Bochs translates from the guest address space into the host-side representation used by the emulator. This translation is page-aware: guest memory may span multiple 4 KB pages, so the handler walks and reassembles the relevant pieces before running the optimized replacement and copying results back to the emulated program.

Scope and trusted computing base. Our prototype is intended to evaluate the Medovik porting model, not to minimize the trusted computing base of the application we want to secure. The TCB, relative to a manual port of the application, grows to include the WebAssembly runtime and emulator. However, a measure of the true security tradeoff would have to account for the chances of introducing security vulnerabilities during porting, as discussed in § 1.

The prototype does not solve all TEE porting dilemmas, and currently does not have full secure device passthrough.

4 Evaluation

The purpose of our evaluation is to use the Medovik prototype to assess the prospects for the Medovik model. Does compatibility-first execution work—can applications really be run out of the box with no effort? Can profiling-guided punch-through ameliorate the overhead?

Experimental Setup. We evaluate Medovik on a RockPro64 [24] that has a Rockchip RK3399 SoC and 4 GB of LPDDR4 memory. The RK3399 is a heterogeneous big.LITTLE processor; we pin benchmark processes to its Cortex-A72 performance cores. In our Linux configuration, these are cores 4 and 5. For experiments that enter OP-TEE, we pin the normal-world process that invokes the trusted application.

Applications include CoreMark [11], the NAS benchmark suite [20], a range of selections from the SPEC 2017 benchmark suite [31], and the ERASE system described in §2. Native and direct WebAssembly benchmarks are run 30 times. Selected SPEC CPU workloads are run three times, matching the SPEC CPU2017 default and the three-iteration reportable-run convention that uses the median time [30]. Full `container2wasm` runs are measured once because they are substantially slower. Slowdowns are reported relative to native execution on the same board and core set.

We compile native binaries with `clang-21` from Ubuntu 24.04 using `-march=native -O3`, direct WebAssembly binaries with `wasm32-wasip1-clang` from WASI SDK 28 [33] (which is based on LLVM 21.1.0) and `-O3`, and WAMR AOT modules for the RockPro64 performance cores by setting `wamrc`'s target to `aarch64` and CPU to `cortex-a72`.

4.1 Compatibility-First Capability and Overhead

All of our workloads run out-of-the-box without any manual effort. This comes at a cost of overhead/slowdown. Figure 2 compares native execution, direct WebAssembly execution, and the full `container2wasm` configuration. The full compatibility-first configuration has a geometric-mean slowdown of approximately 134x (about 100x for ERASE), which is the starting point before any punch-through performance enhancement is done. Note that direct WebAssembly[3] tracks native performance, suggesting that WebAssembly is a good choice for a Medovik approach.[4] Note that some of our benchmarks cannot directly compile out of the box to WebAssembly, but *can* run in the emulated x86_64 machine!

For ERASE, we also include separate measurements in which we run the x86_64 application on QEMU directly on ARM (caveat: without WebAssembly or `container2wasm`). In this case, the starting point slowdown is as little as 24x. While Bochs uses instruction emulation, QEMU uses dynamic binary translation with the Tiny Code Generator (TCG) generating native code for hot instruction blocks. We also explored a development version of QEMU whose TCG generates WebAssembly, but it did not appear ready for prime-time yet. Clearly, the Medovik starting point improves with advances

[3] We compile the workload source *directly* to wasm32/WASI (if possible), using WAMR for AOT compilation, and run the resulting binary.

[4] Interestingly, the AOT-compiled WebAssembly version is sometimes slightly faster than the native baseline. We attribute this to differences in the compiler pipelines: the WebAssembly version is first compiled by the WASI SDK toolchain and then compiled again by WAMR's LLVM-based AOT compiler for the Cortex-A72 target [7]. This second compiler stage can make different target-specific inlining, vectorization, and code-generation choices than our native `clang-21` baseline.

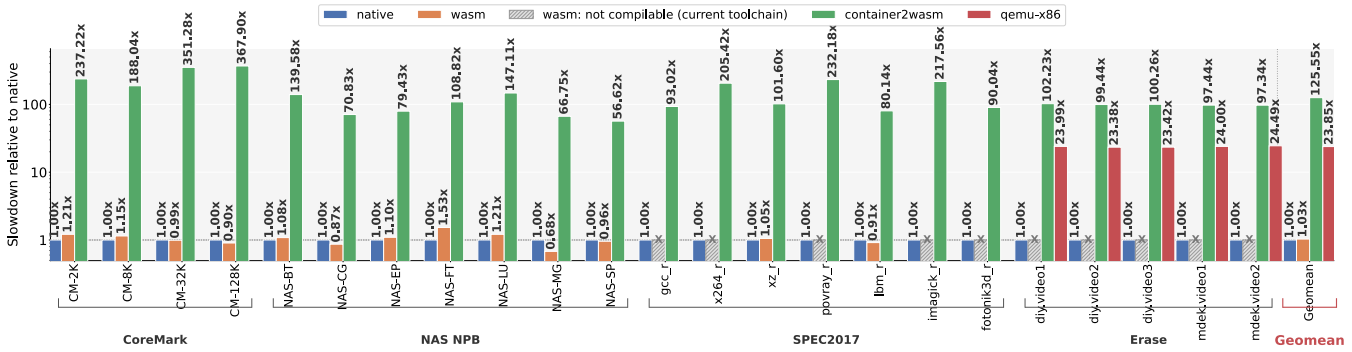


Figure 2. Baseline performance across native execution, direct WebAssembly execution (where possible), and the full compatibility-first Medovik configuration. Medovik allows trivial reuse of existing programs at the cost of significant performance overhead.

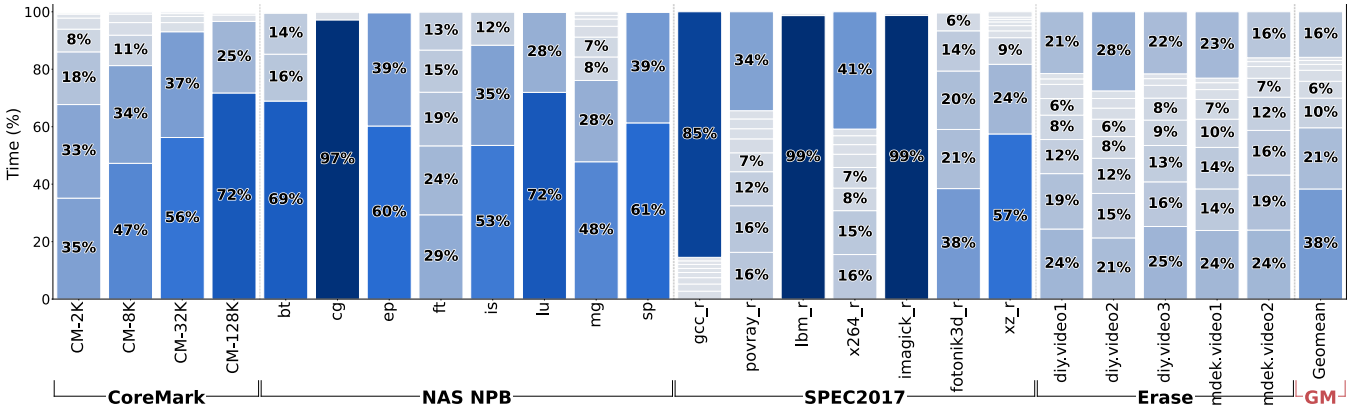


Figure 3. Profile data for benchmark and application workloads. Execution time is typically concentrated in a small number of hot functions, providing candidate targets for punch-through.

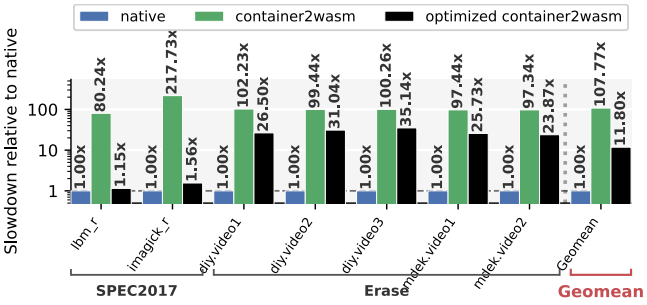


Figure 4. Porting only the top 1-2 functions in the profile significantly reduces slowdown. Medovik enables such incremental performance-focused porting (instead of all or nothing porting),

in emulator technology. Combining our QEMU-ARM and direct WebAssembly results suggests that the general starting slowdown of the Medovik model is likely to be somewhere between 1x and 24x. Our remaining ERASE results use Bochs and thus the entire Medovik stack.

4.2 Profiling Hot Paths

We now profile applications as described in §3.1. Figure 3 breaks down the proportion of total time spent per function in hotness order. Across all workloads, the single hottest function accounts for 38.3% on average; the top three functions, 69.9%; and the top eight, 84.2%. Porting only the hottest functions should have a very outsized effect on slowdown.

Workloads such as `lbm_r` and `imagick_r` spend roughly 99% of their time in a single function, making them ideal punch-through candidates. In contrast, `gcc_r` is a poor candidate because 85% of execution is spread across thousands of functions. ERASE falls between these extremes. Its hot code is less concentrated, but enough time is spent in the leading one or two functions to make targeted punch-through useful, as we'll see next.

4.3 Punch-Through Performance

We implement punch-through for three workloads from the profiles in Figure 4, which shows the resulting speedups. For `lbm_r`, punching through the ~140-line `LBM_performStreamCollideTRT` kernel, which accounts for 98.7% of runtime, reduces slowdown from ~80x to ~1.15x. For `imagick_r`, which does not directly compile to `wasm32`, punch-through of ~500 lines also nearly restores native performance.

ERASE is harder: its profile is less concentrated, and its hottest entry is the Python-level `main` coordinator. We punch through three functions: `randomized_svd` from SciPy (replaced with an Eigen-based [32] C++ handler); `gd_update_efficient_vec`, at ~19% of runtime; and `safe_sparse_dot`, at ~12%. This requires more effort than the C/C++ port, but

leaves the Python runtime and most application logic unchanged, reducing slowdown from ~99x to ~28x.

These examples should not be read only as a lines-of-code comparison. The main cost avoided by Medovik is the all-or-nothing port described in §1 and §2. This matters most for ERASE: a conventional port would have to confront the Python runtime and the NumPy, SciPy, and scikit-learn dependency stack before the application could run *at all*. In Medovik, those dependencies remain in the guest image; punch-through only replaces selected numerical kernels when performance requires it.

5 Conclusion

Counterintuitively, adding layers to the stack can simplify TEE porting rather than complicate it. By preserving the original application environment first and recovering performance through profiling-guided punch-through, Medovik changes porting from an all-or-nothing effort into an incremental process. Our prototype demonstrates this approach for the difficult case of running unmodified x86_64 Linux applications inside ARM TrustZone.

We continue to enhance our prototype. A natural next step is to replace the current Bochs emulator with a QEMU-based `container2wasm` stack. As QEMU is substantially faster than Bochs, this could reduce compatibility-first overhead across all functions before any punch-through optimization is applied to specific functions.

Acknowledgements

We thank Friedrich Doku and Robert Dick for their comments and thoughts on this work, and Haotian Qiao for help with ERASE.

This work was accomplished while Lucian Mocan was a visiting predoctoral fellow at Northwestern. His home university is the University of Strasbourg.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508.

References

[1] Advanced Micro Devices, Inc. 2021. *AMD Memory Encryption.* Advanced Micro Devices, Inc. https://docs.amd.com/v/u/en-US/memory-encryption-white-paper White paper, Document 70363.

[2] Arm Limited [n. d.]. *Learn the Architecture: Realm Management Extension Guide.* Arm Limited. https://developer.arm.com/documentation/den0126/0102/Overview DEN0126.

[3] Arm Limited 2009. *ARM Security Technology: Building a Secure System using TrustZone Technology.* Arm Limited. https://developer.arm.com/documentation/PRD29-GENC-009492/c/ White paper.

[4] Ben Frederickson. 2026. py-spy. https://github.com/benfred/py-spy

[5] Andrin Bertschi and Shweta Shinde. 2025. OpenCCA: An Open Framework to Enable Arm CCA Research. In *8th Workshop on System Software for Trusted Execution (SysTEX 2025).*

[6] Bytecode Alliance. 2026. wamr-compiler. https://github.com/bytecodealliance/wasm-micro-runtime/tree/main/wamr-compiler

[7] Bytecode Alliance. 2026. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/

[8] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).* USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[9] Cloud Native Computing Foundation. 2026. container2wasm. https://www.cncf.io/projects/container2wasm/

[10] Docker. 2026. Docker Documentation. https://docs.docker.com/get-started/get-docker/

[11] EEMBC. 2009. CoreMark. https://www.eembc.org/coremark/

[12] Free Software Foundation. 1998. GNU gprof 2.9.1 Manual. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

[13] Nick Gordon. 2024. *Secure I/O on Trusted Platforms with Lightweight Kernels.* Ph. D. Dissertation. Department of Computer Science, University of Pittsburgh.

[14] Richard Habeeb, Hao Chen, Man-Ki Yoon, and Zhong Shao. 2025. It's a Non-Stop PARTEE! Practical Multi-Enclave Availability Through Partitioning and Asynchrony. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC).* 228–244. doi:10.1109/ACSAC67867.2025.00032

[15] Intel Corporation 2022. *Intel Trust Domain Extensions.* Intel Corporation. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf White paper.

[16] Kevin Lawton. [n. d.]. Bochs: The Open Source IA-32 Emulation Project. http://bochs.sourceforge.net.

[17] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. 2024. SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (Singapore, Singapore) *(ASIA CCS '24).* Association for Computing Machinery, New York, NY, USA, 1600–1616. doi:10.1145/3634737.3644993

[18] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13).* Association for Computing Machinery, New York, NY, USA, Article 10, 1 pages. doi:10.1145/2487726.2488368

[19] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone. 1177-1189 pages. doi:10.1109/ICDCS54860.2022.00116

[20] NASA Advanced Supercomputing Division. 2026. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html

[21] Yuqing Niu, Jieke Shi, Ruidong Han, Ye Liu, Chengyan Ma, Yunbo Lyu, and David Lo. 2026. What You Trust Is Insecure: Demystifying How Developers (Mis)Use Trusted Execution Environments in Practice. arXiv:2512.17363 [cs.SE] https://arxiv.org/abs/2512.17363

[22] OP-TEE. 2026. OP-TEE Documentation. https://optee.readthedocs.io/en/latest/index.html

[23] Kevin Pedretti. 2013. *Kitten Lightweight Kernel Overview.* Technical Report SAND2013-10537P. Sandia National Laboratories. https://www.osti.gov/servlets/purl/1677583 Presentation.

[24] PINE64. 2026. ROCKPro64. https://pine64.org/devices/rockpro64/

[25] OP-TEE Project. [n. d.]. About OP-TEE. https://optee.readthedocs.io/en/latest/general/about.html

[26] Python Software Foundation. 2026. Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html

[27] Haotian Qiao, Vidya Srinivas, Peter Dinda, and Robert Dick. 2025. Efficient Video Redaction at the Edge: Human Motion Tracking for Privacy Protection. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS 2025.*

[28] Haotian Qiao, Vidya Srinivas, Peter Dinda, and Robert Dick. 2025. Efficient Video Redaction at the Edge: Human Motion Tracking for Privacy Protection. *ACM Transactions on Embeded Computer Systems* 24, 5s (Septermber 2025), 22 pages.

[29] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. doi:10.1145/3373376.3378469

[30] SPEC. 2021. runcpu - Using CPU 2017. https://www.spec.org/cpu2017/docs/runcpu.html

[31] Standard Performance Evaluation Corporation. 2026. SPEC CPU2017. https://www.spec.org/cpu2017/

[32] Eigen Team. 2026. Eigen: A C++ Template Library for Linear Algebra. https://libeigen.gitlab.io/

[33] WebAssembly. 2025. wasi-sdk-28. https://github.com/WebAssembly/wasi-sdk/releases/tag/wasi-sdk-28