UFR de mathématique et d'informatique
Université de Strasbourg

Réseau Figure
COMPRENDRE • EXPLORER • TRANSFORMER

B1G ACADEMIC ALLIANCE

MASTER'S DEGREE IN COMPUTER SCIENCE
SCIENCE AND ENGINEERING OF NETWORKS, INTERNET, AND SYSTEMS

Master's Thesis

Lucian MOCAN
lucian.mocan@etu.unistra.fr

Medovik: Simplifying Porting to Trusted Execution Environments

With an Exploration of LLM-Assisted Code Translation

August 1, 2026

Supervisor

Peter DINDA pdinda@northwestern.edu

Host Institution

Prescience Lab, Department of Computer Science
Northwestern University

Plab Northwestern | McCORMICK SCHOOL OF ENGINEERING

Internship Period

February 2 – July 31, 2026

Acknowledgments

I would first like to thank Professor Peter Dinda for taking me on for these six months, for the time he spent discussing ideas with me, answering my questions, and helping me move forward when I was stuck. His guidance shaped both the Medovik project and my understanding of research. I am also grateful to the entire Prescience Lab (special thanks to Michael Polinski, Friedrich Doku, David Krasowska, and Peizhi Liu) for making this experience feel real, welcoming, and alive.

I am grateful to Northwestern University and its Computer Science Department for hosting me during this time. I also thank the Big Ten Academic Alliance and the FIGURE network for making this internship possible.

I would also like to thank the Barac family for helping me find housing, settle into the city, and adapt to a new environment. Their help made the beginning of this experience much easier than it could have been.

I would like to thank my parents for their constant support and for always believing in me. Finally, I thank Jesus Christ for being by my side through the hills and valleys.

AI Use Disclosure: The research ideas and experiments in this thesis were developed by me in collaboration with the Prescience Lab. While writing this thesis, I used AI tools to revise English phrasing, discuss its structure, and identify repetition. I reviewed the final text and take responsibility for its technical accuracy.

Contents

Chapter 1

Introduction

As part of my Master's degree and Cursus Master en Ingénierie (CMI) in Internet, Systems, and Networks at the University of Strasbourg, I completed a six-month internship at the Prescience Lab, in the Department of Computer Science at Northwestern University. The internship took place from February 2 to July 31, 2026, under the supervision of Professor Peter Dinda.

This was an opportunity to work in an academic research environment and to take part in a systems research project from its early formulation to the submission of a workshop paper. The main result of this work was Medovik[1], a system model and prototype for simplifying the porting of applications to Trusted Execution Environments (TEEs). Together with Professor Peter Dinda and Michael Polinski, a PhD student in the Prescience Lab, I submitted my first paper as lead author, *Medovik: Simplifying Performant Porting to TEEs by Adding Even More Layers to the Honey Cake* [1], to the 14th Workshop on Programming Languages and Operating Systems (PLOS 2026), co-located with SOSP.

Trusted Execution Environments, described in more detail in Section 3.1.2, are difficult targets for existing applications because they lack the Linux runtimes, libraries, and device interfaces on which those applications depend. Medovik avoids an immediate full port by running the original x86_64/Linux environment through an emulator compiled to WebAssembly inside ARM TrustZone [2]. Because this compatibility path is slow, profiling identifies a small set of hot functions that can be replaced with punch-through handlers. I worked with Professor Peter Dinda and Michael Polinski to develop the Medovik model, then implemented and evaluated its prototype.

Medovik was the central project of my internship, but it was not the only thread of work. I also explored source-language porting with large language models (LLMs), for which I carried a literature review, formulated a research question, and built supporting experimental scripts. This secondary project focused on how to establish confidence in LLM-generated code translations.

Beyond the technical work itself, another question guided this internship: whether rigorous academic research is a path I want to pursue in the future. I believe every human being looks for something that can give them purpose. This internship taught me enough to begin realizing that purpose is not necessarily found in what one does, but in what one wants that work to become.

[1] Medovik is an Eastern European honey cake made of many layers.

Chapter 2

The Research Environment

2.1 Northwestern University

Founded in 1851, Northwestern University is a private research university located in Evanston, Illinois, in the United States. It spans three campuses in Evanston (Figure 2.1), Chicago, and Doha, Qatar, and serves more than 9,000 undergraduate students and 14,000 graduate students. Northwestern is also a major research institution, with more than 90 school-based research centers and 20 university-wide research centers [3].

Figure 2.1: Northwestern University's Evanston campus [4]

Figure 2.2: The Mudd Library, home of NU's Computer Science Department [5]

2.2 The McCormick School of Engineering

Northwestern Engineering, as it is commonly known, traces its origins back to 1909, when it was established as a College of Engineering. It took the name McCormick School of Engineering and Applied Science in 1989, following a $30 million grant from the Robert R. McCormick Foundation. Today, the school brings together over 200 faculty across disciplines ranging from biomedical and chemical engineering to computer science and industrial engineering [6].

2.3 The Department of Computer Science

The Department of Computer Science sits within McCormick and occupies Mudd Library (Figure 2.2) on the Evanston campus. Research in the department covers a broad range of areas, from systems and networking to artificial intelligence, theory, security, robotics, and others [7]. The department hosts over 25 research groups and labs, and is affiliated with more than a dozen university research centers and institutes, spanning fields as varied as quantum information, deep learning, sustainability, and transportation [8].

I think that the real strength of the department comes from its people and how they connect. Rather than focusing on a single area, dozens of research groups work on problems across the entire computing stack. Ideas are easily shared with other departments and schools across the university, fostering a culture of collaboration where different perspectives can meet and grow.

I had the opportunity to be part of this environment as a member of the Prescience Lab.

2.4 The Prescience Lab

The Prescience Lab has a strong history of mentorship. It has graduated over 25 master's students and 50 undergraduates while completing more than 15 research projects. During my internship, the group consisted of eight PhD students, three masters students, five undergraduates, and me.

Professor Peter Dinda directs the Prescience Lab. He holds appointments in both Computer Science and Electrical and Computer Engineering [9]. His research focuses on experimental computer systems, specifically virtualization and parallel and distributed systems. Over his career, he has published more than 145 papers and was named an IEEE Fellow in 2015.

The lab runs several projects at the same time [10]. The **Constellation Project** [11] makes different types of processors work together. The goal is to hide that complexity so the programmer doesn't have to manage it. The **Interweaving Project** [12] rethinks the whole hardware and software stack. It asks how we would build it today if we could start from scratch.

The **Buoyancy Project** [13] focuses on floating-point correctness. Finally, the **Privacy Backplane** [14] is the focus of my internship. I describe this work in the next section.

2.5 The Privacy Backplane Project

The Privacy Backplane project envisions a framework designed to give people individual privacy in Internet of Things (IoT) environments [14]. IoT devices are now everywhere in our daily lives. They constantly collect personal data, usually under broad rules like GDPR [15] or CCPA [16] that apply the same way to everyone. The Privacy Backplane is an effort to make an individual choice possible. Each person would be able to control how their own data is accessed and used. Realizing this vision would require a legal framework that obligates IoT environments to negotiate with users, together with a technical system that enforces the resulting choices [14].

Several research efforts in the lab approach this problem from different systems angles. Some study how to establish trust in software execution, others explore how hardware capabilities can enforce isolation while still allowing efficient access to devices, and another line of work investigates lightweight kernel support for secure I/O on trusted platforms [17, 18, 19]. Together, these projects show the broader systems context in which the Privacy Backplane is being developed.

The effort most directly connected to my internship is ERASE, a human motion-tracking and redaction application. ERASE processes video at the edge and redacts people who have not opted into image capture [20, 21]. In the Privacy Backplane context, Trusted Execution Environments (TEEs) are a natural enforcement mechanism for applications of this kind. My internship focused on the systems question that follows: how do we get an application like ERASE, which was not necessarily developed for a TEE operating system, to run inside a TEE? The next chapter returns to ERASE in more detail and presents this problem through Medovik.

Chapter 3

Medovik: Make It Run, Make It Fast

When I arrived, Medovik existed only as an initial research idea; there was no prototype or established implementation plan. I met with Professor Dinda weekly to discuss progress and research decisions, but I worked with substantial autonomy between those meetings. I led the hardware integration, system design, prototype implementation, evaluation, and paper writing. Professor Dinda helped shape the research direction, while Michael Polinski helped with the testbed and contributed to the paper.

3.1 Problem Statement and Objectives

3.1.1 Security Guarantees

Modern computing depends heavily on security guarantees. Over time, both software and hardware have tried to provide those guarantees, although they do so in different ways.

Software

Memory safety is the obvious place to start. Languages such as C and C++ give programmers direct control over memory, which is why they remain central in kernels, runtimes, and other low-level systems code. That control comes with a cost: memory errors are still a major source of vulnerabilities. This is one reason systems languages such as Rust have attracted so much attention. Through its ownership model, Rust rejects many unsafe memory patterns at compile time without requiring a garbage collector [22, 23].

Formal verification takes a different route. Although testing can find bugs, it cannot prove by itself that a system is correct. Formal verification uses mathematical specifications and machine-checked proofs to show that an implementation satisfies specific properties. A major example is the seL4 microkernel, which comes with machine-checked proofs of implementation correctness, along with proofs about security properties such as confidentiality, integrity, and availability for correctly configured systems [24]. Similar work exists lower in the software stack: CompCert is a formally verified C compiler [25], while HACL* and EverCrypt provide verified cryptographic components [26, 27].

Sandboxing and isolation solve a narrower problem: containment. A sandbox restricts what code can access, so that a compromised component has limited reach outside its environment. Browsers use this idea heavily. Chromium, for example, uses sandboxing to constrain less-trusted processes [28]. WebAssembly applies this concept by running compiled code inside a strictly controlled, memory-safe virtual machine, preventing the application from accessing the host's memory or resources without explicit permission [29]. Containers provide another common form of isolation by relying on operating-system mechanisms such as namespaces, control groups, and restricted capabilities [30].

These techniques are powerful, but each still depends on some trusted software layer: a compiler, verifier, runtime, browser, container engine, operating system, or kernel. If that layer falls outside the trust boundary, software guarantees alone are not enough [31]. Hardware-backed guarantees become relevant at exactly this point, which is why Trusted Execution Environments matter for this project.

Hardware and the Trusted Computing Base

In any secure system, the set of hardware, firmware, and software components critical to its security is known as the Trusted Computing Base (TCB). If a component within the TCB is compromised, the security of the entire system can be defeated. Therefore, a core goal of secure systems design is to shrink

the TCB as much as possible. The foundation of this base is the *root of trust*. It is a component that later security decisions depend on. Because there is no lower layer available to verify it, the system must trust it by design [32]. A Trusted Platform Module (TPM) is a common example. It can store keys, certificates, and platform measurements, and it is often used for boot integrity and attestation [33].

Password checking provides a simple example of TCB reduction, as shown in Figure 3.1. If a normal application sends a password to the operating system (OS) and the OS performs the check, then the OS is part of the TCB for that operation. A malicious or compromised OS could read the password, change the result, or leak the secret state used for verification. When the check runs inside an isolated hardware-backed environment instead, the ordinary OS no longer needs to be trusted to preserve the password's confidentiality or the check's integrity. It still routes the request and can delay, block, or refuse it, so availability continues to depend on components outside the TEE.

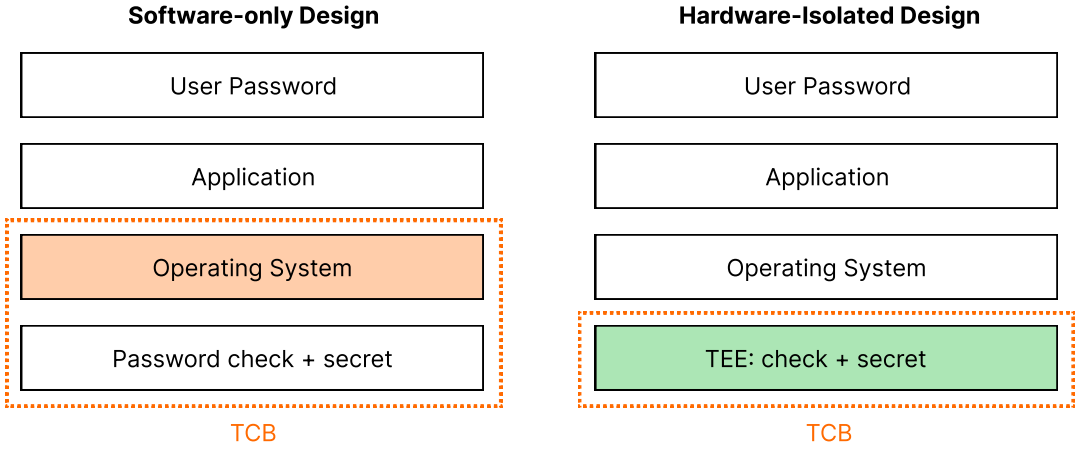

Figure 3.1: Property-specific TCB reduction for password checking. The TEE protects the password's confidentiality and the check's integrity, while the operating system can still deny service.

Hardware isolation can take several forms. At the scale of virtual machines, AMD SEV encrypts guest memory using keys managed by the AMD Secure Processor [34]. SEV-SNP adds integrity protections against a malicious hypervisor [35]. At the instruction-set level, CHERI takes a different path by adding hardware capabilities for fine-grained memory protection and software compartmentalization [36].

The form of hardware isolation used in this project is the Trusted Execution Environment, introduced in the next section.

3.1.2 Trusted Execution Environments (TEEs)

A Trusted Execution Environment (TEE) is not just another name for a sandbox. Sabt et al. define it as a protected processing environment that provides authenticity of executed code, integrity of run-time state, confidentiality of code and data, and mechanisms for attesting its trustworthiness to another party [37]. In practical terms, code inside a TEE should be protected from the rest of the system, including the ordinary operating system.

A sandbox mainly limits what a program can do to the outside world. A TEE also protects the program from that outside world. For applications that manipulate sensitive data on devices with large software stacks, this changes the trust boundary: the whole operating system no longer has to be trusted for the protected computation.

However, isolation alone is insufficient if the user cannot verify what is isolated. While a TEE provides a secure environment, the caller still needs proof that the correct code was loaded into it. This is done using a cryptographic mechanism called attestation. Its purpose is to allow a TEE to prove its identity and integrity to an external party. When a function is loaded into the TEE, the hardware computes a cryptographic representation of its initial state, which is then signed with a hardware-bound key. If a

client application wants to invoke that secure function, it can request an attestation report. By verifying the signature and checking the measurement, the caller can be confident that the expected function was loaded inside a real TEE, rather than a malicious OS pretending to be one. This does not prove that the function is free from bugs or runtime attacks, but it does authenticate its initial state.

Different hardware platforms implement this idea differently. Intel SGX protects selected code and data inside enclaves [38]. ARM TrustZone instead splits the machine into two execution worlds: the Normal World and the Secure World [2]. This internship focuses on ARM TrustZone, because the target devices are ARM-based edge/IoT systems.

In TrustZone, the Normal World typically runs the ordinary operating system, such as Linux. The Secure World hosts trusted code with stronger isolation from the rest of the system. Figure 3.2 shows how this split maps onto ARM exception levels.

Since this work targets ARM-based edge devices, the relevant case is TrustZone on ARMv8-A, the architecture used by the boards in my experimental setup. ARMv8-A organizes execution into four exception levels (ELs), from EL0, the least privileged, to EL3, the most privileged. Applications typically run at EL0, operating systems at EL1, hypervisors at EL2, and the secure monitor at EL3. Figure 3.2 shows which software runs at each exception level in the Normal and Secure Worlds.

Transitions between the Normal World and the Secure World go through the secure monitor at EL3. Firmware such as Trusted Firmware-A provides part of this low-level secure infrastructure and initializes the platform before the operating systems run [39]. Normal World code cannot call Secure World code like an ordinary library. It issues a secure monitor call, and the monitor controls the transition. The same boundary that supports the security model also makes TEE programming very different from ordinary application programming.

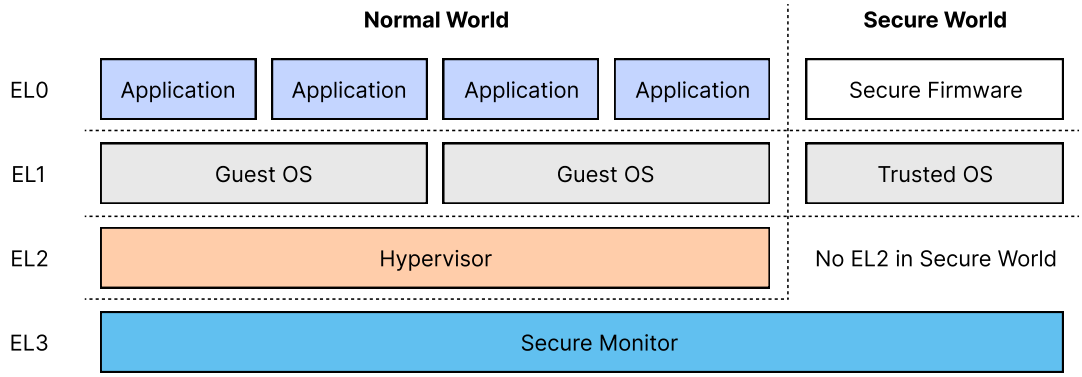

Figure 3.2: ARMv8-A TrustZone exception-level layout for the RK3399 target used in this work [2].

3.1.3 Challenges of Porting to TEEs

Because of this strict boundary, porting an existing application into a TEE is rarely straightforward. 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 [40]. This fragmentation breaks standard toolchains, and a recent empirical study demonstrates that the steep learning curve drives developers to bypass safe APIs or introduce insecure practices that violate the trust boundary [41].

While newer cloud-focused technologies such as Intel TDX [42] and AMD SEV [34] allow entire x86_64 virtual machines to run as hardware-protected confidential VMs, edge and IoT devices are typically ARM-based. Although the ARMv9 Realm Management Extension (RME) [43] was announced in 2021 to provide similar virtualization abstractions, it is not yet an option in commodity edge/IoT

hardware [44]. 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.

Frameworks like Enarx [45] attempt to simplify this porting process by deploying applications as WebAssembly modules inside SGX or SEV, theoretically allowing developers to execute high-level languages like Python. However, because these solutions require the application itself to be compiled directly to WebAssembly, they are incapable of executing pre-compiled native C extensions or system ABI-based shared libraries. 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 [46, 38]. Manual partitioning is notoriously difficult to implement securely, and poor implementations frequently lead to huge performance overheads and system availability bottlenecks [47].

Porting existing applications into edge/IoT TEEs, such as ARM TrustZone, can be overwhelming even when they already target embedded hardware. The Medovik effort grew out of the research team's attempt to port ERASE [21, 20], a human motion tracking and redaction application, into such an environment with significant cooperation from its developers. They have encountered similar issues when porting other such tools and a distributed query framework to TEEs [1].

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.

ERASE Porting Effort

The goal of ERASE [21, 20] 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 occur only on the edge device, so that no data for anyone who has not opted in ever leaves the edge device. ERASE achieves over 60 frames per second for one person on a ROCKPro64 SBC (ARM Cortex-A72), dropping to 20 frames per second for five people, approximately $10\times$ faster than previous work. I used the same hardware for my work during this internship.[1]

Originally, ERASE was published as an x86_64 Ubuntu Docker image. Porting it to the ROCKPro64 presented two distinct challenges. The first was architectural: translating x86_64 code to ARM64. While modern ARM64 support has improved and mitigates most of that effort, this is a secondary concern.

The primary challenge was moving the application into TrustZone's Secure World. Running ERASE in the Secure World is essential because it allows attestation to verify the redaction code, and it enables hardware partitioning to provide a direct path from the camera to the secure environment. This lets anyone who trusts the attestation process verify that the video is redacted before it leaves the device.

Systems Environment

To host ERASE in the Secure World, I explored several TEE operating environments. However, each option forced the developer into the traditional porting trap illustrated in Figure 3.3(b).

For example, OP-TEE [48] is a widely used Secure World OS, but it is designed around a Remote Procedure Call (RPC) model. It expects Secure World code to act as short-lived handlers responding to requests from Normal World Linux. This is a poor fit for ERASE, which operates as a continuous, long-lived video processing pipeline. Alternatively, KaTZe [19], a Secure World port of the Kitten lightweight kernel [49], is designed for process-like workloads and supports a limited subset of Linux system calls. While its split-core design can route selected requests between the Normal and Secure Worlds, it still does not provide a complete Linux environment.

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

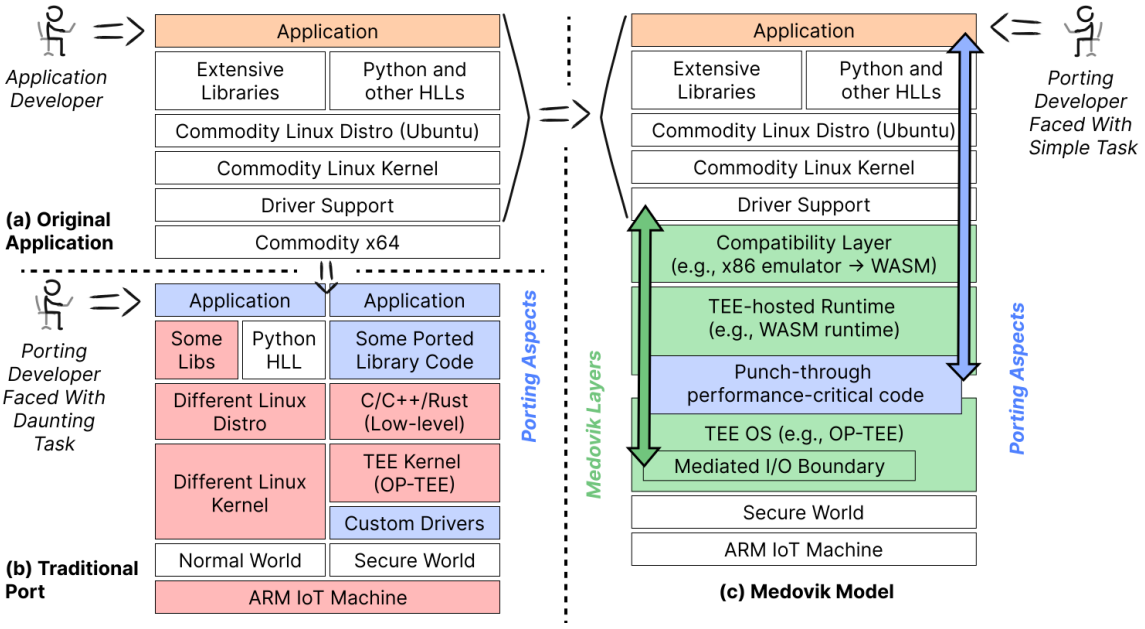

Figure 3.3: Medovik simplifies porting to TEEs. (a) Applications are built for commodity x86_64/Linux stacks. (b) Traditional TEE porting forces developers to split or replace application, runtime, OS, and I/O layers across Normal and Secure Worlds. (c) Medovik preserves the original stack inside TEE-hosted compatibility layers, then uses profiling-guided punch-through for selected hot paths.

In all cases, the fundamental problem remains: the developer must manually dissect the application. They are forced to choose exactly which application logic, libraries, OS dependencies, and I/O components must be rewritten, replaced, or split across the trust boundary [1].

Language Assumptions

There is typically a fundamental disconnect between the language environment of application developers and that of TEE toolchain engineers. In the case of ERASE, the authors chose the most effective tools for signal processing. Their implementation began in MATLAB and eventually transitioned to Python using NumPy and SciPy. For video I/O from both devices and files, they relied on OpenCV and its Python bindings. While this approach made the ERASE codebase superficially compact, it introduced enormous runtime dependencies.

This reality directly conflicts with traditional TEE porting options. Secure environments expect embedded-style code that is written in a low-level language like C, C++, or Rust, and is completely self-contained except for an explicit hardware abstraction layer (HAL) and OS abstraction layer (OSAL).

While the authors could theoretically rewrite the entire application to fit this restrictive model, such a massive engineering effort is difficult to justify. The current implementation already meets the necessary performance targets, and its execution speed is likely to improve naturally as its dependencies are optimized over time [1].

Environment Assumptions

Application developers rely heavily on the underlying operating system and its ecosystem of packaged libraries and frameworks. For instance, the ERASE designers implicitly assumed the application would run on a modern Ubuntu distribution, where virtually all necessary dependencies are readily available through standard package managers.

TEEs fundamentally disrupt this assumption. As noted in the previous section, the Secure World operating environments I evaluated lacked basic OS features. Some lacked the concept of isolated processes entirely, and those that did support them offered only partial implementations of the Linux system

call interface. In the case of KaTZe, correctly implementing the missing system calls required by ERASE proved to be a major engineering hurdle.

Furthermore, regardless of the primary programming language, shared libraries are often strictly required. Dependencies like NumPy, SciPy, and OpenCV are typically compiled as system ABI-based shared libraries to maximize performance. For many complex packages, compiling a purely static version is not even a supported option. Additionally, many higher-level language runtimes rely on dynamic loading functions like `dlopen()`, making static linking impossible. Despite their necessity, shared libraries are notoriously difficult to implement securely and are either entirely absent or insufficiently supported in TEE operating systems, including KaTZe.

Finally, device driver support within TEEs is severely limited. For ERASE to access the camera, the video data would either have to be routed insecurely through the Normal World, or a custom secure driver would have to be written for KaTZe. Porting an existing Linux kernel driver into the Secure World and maintaining its compatibility with OpenCV and its dependencies would demand considerable hardware expertise and development effort.

3.1.4 Reducing Application Porting Effort

The constraints described above lead to two research questions:

1. Can an existing x86_64/Linux application execute inside an ARM TrustZone environment without a manual TEE-specific port?
2. If compatibility layers make that execution too slow, can profiling-guided replacement of selected functions recover useful performance without rewriting the complete application?

Section 3.2 defines the Medovik model used to investigate these questions, and Section 3.4 evaluates the prototype.

3.2 Medovik Model and Prototype

The Medovik model changes the first porting milestone from efficient execution to correct execution. It initially preserves the applications expected x86_64/Linux environment inside a compatibility substrate hosted by the TEE, while optimization begins only after the application runs.

3.2.1 Assumptions and Core Elements

For this model to work, three conditions must hold.

1. The application can be packaged together with the operating system and library state it expects (e.g., a container-derived image).
2. 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.
3. 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).

These assumptions still impose constraints. They are much weaker, however, than requiring a full manual port before the first execution.

Furthermore, Medovik has two core elements. The first is *compatibility-first execution*. The original application binary and its dependency environment are kept as close as possible to their normal form, even when doing so requires extra layers of emulation or interpretation. This path is slow by design, but it gives the developer a working starting point. The second element is *profiling-guided punch-through*. After the application runs, the developer profiles it in its ordinary environment, identifies the functions that

dominate execution time, and ports only those performance-critical paths. The replacement functions then bypass the slowest layers through explicit punch-through mechanisms.

Why is this useful? Because many programs spend most of their time in a small part of their code. When that is true, the difficult TEE-porting work described in §3.1.3 applies only to a small set of functions, rather than to the entire application and dependency stack.

The model is deliberately not tied to one emulator, one WebAssembly runtime, or one TEE operating system. My prototype uses an x86_64 emulator compiled to WebAssembly and hosted in an ARM TrustZone environment. Another implementation could use different layers or a different punch-through mechanism.

3.2.2 Prototype

My prototype instantiates the model for a deliberately difficult target: running unmodified x86_64 Linux applications inside an ARM TrustZone TEE. I implemented it on a ROCKPro64 board using OP-TEE [50] 4.9.0 as the TEE OS, WAMR [51] as the WebAssembly runtime, Bochs [52] as the x86_64 emulator, and Debian-based Docker images as source application environments. The prototype also includes a hypercall mechanism for punch-through, described later in this section.

Prototype Layers

From the original application down to the TEE, the prototype has five layers:

- *Application.* The input is an existing Linux x86_64 application.
- *Linux kernel and container image.* The application and its dependencies are placed in a Docker [53] container, which becomes the root filesystem of a bootable OS image.
- *x86_64 emulator.* Bochs boots the image on an emulated x86_64 machine. In this prototype, Bochs itself is compiled to WebAssembly.
- *WebAssembly runtime.* WAMR executes the emulator inside the TEE and provides the host interface required by the WebAssembly module. I use ahead-of-time compilation to reduce its overhead.
- *TEE OS.* OP-TEE provides the Secure World execution environment in which the WebAssembly runtime runs.

This stack is expensive. Application instructions first pass through an emulated x86_64 machine; the emulator runs as WebAssembly; and the WebAssembly runtime itself executes inside a constrained TEE. That overhead is the reason punch-through becomes necessary.

3.3 Implementation Work

After defining the prototype layers, the next task was to deploy those layers on the target hardware and connect them into a functional system. This was not a straightforward process and required several specific engineering choices.

3.3.1 Selecting the Secure World Environment

The target hardware for the prototype was the ROCKPro64. Building a TrustZone stack for this board requires Trusted Firmware-A (TF-A), U-Boot, a Normal World OS, and a Secure World OS.

The initial design choice was the Secure World OS. I first attempted to use Kitten [54], a lightweight research kernel designed for high-performance computing. However, debugging errors in this undocumented research codebase proved prohibitive. I therefore pivoted to OP-TEE [48], an open-source Trusted OS for ARM TrustZone. This choice was also informed by WATZ [55], a project that successfully runs the WebAssembly Micro Runtime (WAMR) inside OP-TEE. While WATZ targeted different hardware, it validated the architectural choice of embedding WebAssembly in this specific Secure World environment.

3.3.2 Hardware Porting and Toolchain Setup

While OP-TEE is well-supported, there was no ready-to-use build for the ROCKPro64. To resolve this, I used the RockPi 4 configuration because it has the same RK3399 SoC and already had OP-TEE support. Starting from the RockPi 4 configuration, I compiled a custom Linux kernel and modified the `.dts` device-tree file so Linux would recognize OP-TEE and load its drivers.

To ensure the research was reproducible, I wrote scripts to automate this build process and integrated a GitHub Actions workflow to generate consistent release artifacts. Finally, to create a more functional development environment, I replaced the default minimal Buildroot filesystem with Alpine Linux, providing access to a standard package manager.

3.3.3 Emulation and WebAssembly Integration

With the Secure World environment established, the next challenge was executing the x86_64 guest system. The application and its dependencies are packaged into a Debian-based Docker image, which serves as the root filesystem for the bootable Linux guest.

To execute this guest, I selected `container2wasm` [56]. This tool packages the Docker-derived image alongside Bochs [52], an x86_64 emulator compiled to WebAssembly. By running Bochs as a WebAssembly module, the emulated machine remains purely x86_64, allowing the Linux application to run unmodified on the ARM-based TEE.

A higher-performance alternative would have been to use an emulator like QEMU to leverage its Tiny Code Generator (TCG) and dynamic binary translation. However, a full-featured WebAssembly port of QEMU does not currently exist. The partially working ports that are available rely heavily on browser-specific APIs, making them impossible to deploy out of the box in a standalone TEE runtime. Committing to a QEMU porting effort before validating the core Medovik model would have introduced unnecessary engineering risk.

Once the baseline architecture was validated, I initiated a secondary effort to port QEMU to WebAssembly to further optimize performance. This ongoing port faces significant systems challenges. First, standalone WebAssembly runtimes currently lack robust threading support, unlike browser engines that can easily rely on Web Workers. Second, achieving optimal performance requires a mature Just-In-Time (JIT) compiler for the ARM64 target. While runtimes like Wasmtime provide excellent JIT support, they introduce a large dependency footprint that is poorly suited for a constrained Secure World environment.

Given these constraints, the prototype relies on the Bochs implementation. Integrating this generated WebAssembly module into the Secure World, however, still required extending the runtime. The version of WAMR used by WATZ lacked several WebAssembly System Interface (WASI) functions required by Bochs. Specifically, Bochs depends on host clock and time support to accurately emulate hardware timers for the guest OS. I implemented this missing WASI support directly into the runtime to allow the emulator to boot correctly.

3.3.4 Compilation Strategy for Performance

Bochs already dominates the cost of the compatibility stack. Because every emulated instruction passes through the WebAssembly runtime, additional overhead in that layer is amplified across the emulators instruction stream. WAMRs JIT mode made end-to-end Bochs execution impractically slow, whereas ahead-of-time (AOT)-compiled WebAssembly approaches native performance in the direct WebAssembly experiments reported in Section 3.4.2. AOT cannot remove the cost of x86 emulation, but it prevents the WebAssembly layer from becoming another dominant bottleneck. It is also important for punch-through performance. Because the handlers are compiled into the same WebAssembly module as Bochs, WAMR AOT-compiles them to ARM64 as well. Once dispatched, a handler bypasses x86 emulation and runs near native speed, aside from hypercall and guest-memory transfer costs.

3.3.5 Profiling

Choosing a Profiler

Punch-through only makes sense when a small number of functions account for a large part of the execution time. I profiled each workload in its ordinary environment to identify candidate functions for punch-through.

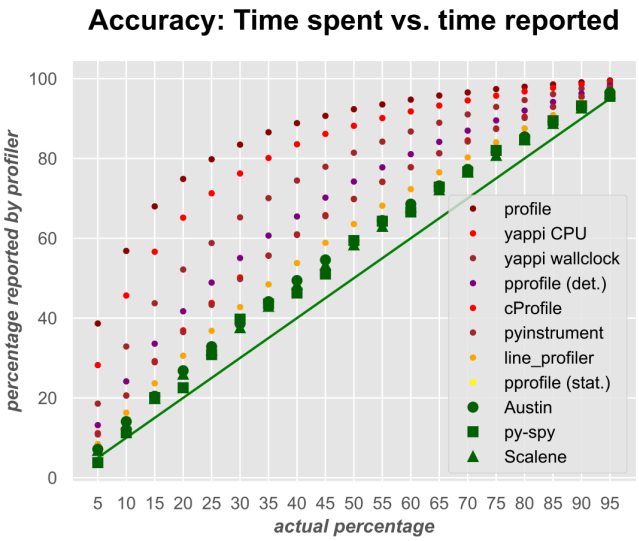

Figure 3.4: Reported accuracy varies considerably across Python profilers. Points near the diagonal report a CPU percentage close to the actual value [57].

Profiling itself was new to me. The closest I had come to performance engineering was through a few parallel programming projects. Even in those, I focused more on correctness and reasonable complexity than on squeezing performance out of a particular implementation. At first, I thought a profiler gave a precise measurement of where a program spent its time. I learned that the reported times depend on how the profiler observes the program. There are two main approaches. A *deterministic* profiler records function calls and returns, while a *sampling* profiler periodically inspects the active call stack. Both add overhead and may account for compiled-library time differently.

For C and C++ workloads, I used `gprof` [58]. It uses compiler instrumentation to record call relationships together with periodic sampling to estimate execution time. For Python, I started with `cProfile` [59] and used its output to select punch-through candidates for ERASE. My first punch-through attempts did not produce the performance improvement I expected. This made me question whether `cProfile` had identified the right bottlenecks, so I started looking at other Python profilers.

That search led me to the accuracy comparison by Scalene [57]'s authors shown in Figure 3.4. In their experiment, some commonly used Python profilers, including `cProfile`, reported CPU-time shares far from the actual values. I then evaluated Scalene and `py-spy` [60]. Scalene's line-level output separates time spent in Python, compiled libraries, and the system. This helped me see what kind of work made a line expensive, but I needed to know which functions dominated total execution time. `py-spy` gave me that view more directly. If I had based the punch-through work only on my first results, I might have spent time replacing the wrong functions.

Exploring Executionless Profiling

I use *executionless profiling* to describe predicting likely hot functions from source code and a workload description without executing the program. In Medovik, I could profile each program in its ordinary environment, but that is not always possible. A developer may have the source code and a description of the workload before the target platform or its profiling tools are ready. Dynamic profiling can also be expensive when a workload is long-running or when its behavior must be tested across many inputs.

In the remainder of this thesis, I use *coding agent* to mean an LLM-based system that can inspect a repository and use available tools to carry out a task over multiple steps. Could such an agent make a useful prediction without running the application at all?

I do not expect an agent to replace measurement, but it could identify plausible hot functions, give the developer somewhere to start, and narrow the amount of code that needs targeted profiling. I gave a coding agent the source tree and a description of the workload in natural language. It could list files, read source code, and search the repository, but it could not run the program, tests, compiler, or profiler. I asked it to rank the functions it expected to dominate runtime and to give a time-share range, confidence level, and short explanation for each prediction. To establish a baseline, I repeated the same task without these restrictions, allowing the agent to run the program and use any available profiling tools.

I tested both versions on `pyflate`, a pure-Python bzip2 decompression workload from the Python Performance Benchmark Suite. Because this was a preliminary experiment, I used `cProfile` as a deliberately modest dynamic baseline. As Figure 3.4 shows, `cProfile` is among the less accurate profilers for attributing CPU time. The experiment tested whether source-only predictions could identify a set of hot functions comparable to this basic dynamic baseline. Across 30 prediction-only runs, the agent found an average of 78.0% of the dynamically measured top five algorithmic functions and 81.3% of the top ten.

Those numbers are encouraging, but the agent did not give the same answer every time. The mean pairwise Jaccard similarity[2] between runs was 0.541 for the top five and 0.693 for the top ten. The experiment also covered only one workload, and some early runs produced different output formats that I had to normalize before comparison.

Only after starting this experiment did I find Phaedrus, a compiler-assisted framework that uses generative models and LLMs to predict dynamic application behavior across inputs [61]. Phaedrus studies the idea at a much larger scale. My idea was a little narrower. Could a general coding agent inspect a repository and recover useful information about one workload's hot functions? Based on my results, I am inclined to say yes, but I am not sure I would use those predictions if better profiling tools were available.

3.3.6 Punch-Through Handlers

I implemented punch-through by redirecting selected guest-side call sites. Figure 3.5 illustrates the complete execution sequence and memory access pattern across the emulation boundary. The guest program calls a small shim instead of the original hot function. This shim executes the x86_64 `cpuid` instruction with a custom Medovik value in the emulated `EAX` register.

When Bochs sees this value, it treats the instruction as a hypercall and dispatches to a handler added to the Bochs codebase. Since the handler is compiled together with Bochs, it becomes part of the same WebAssembly module as the emulator. The handler then runs inside WAMR instead of being interpreted through full x86_64 emulation.

The redirection mechanism depends on the source language. For C and C++ programs, `LD_PRELOAD` is enough to replace the original function with the shim. For Python applications such as ERASE, I used the Python C API [62] to expose replacement module functions that the application can import normally.

As shown in Figure 3.5, managing memory across the boundary requires operating directly within Bochs's allocated WebAssembly memory space. When a handler needs guest data, Bochs has to translate from the emulated guest address space to the host-side representation used by the emulator. Guest memory may cross several 4 KB pages, so the handler cannot simply copy one flat block. It must gather data from those pages, run the optimized replacement, and copy the result back to the emulated program.

I also investigated a zero-copy design in which punch-through handlers would operate directly on the existing guest-memory pages instead of reassembling the data in a temporary buffer and copying the result back. Section 3.4.4 shows that the copy and write-back overhead is negligible for the coarse-grained handlers evaluated in this prototype. Direct memory access could nevertheless benefit finer-grained handlers and remove one source of overhead. I did not complete this design during the internship; implementing it would likely require extensions to WAMR's memory interface and possibly to OP-TEE's memory-management mechanisms.

3.3.7 Scope and Trusted Computing Base

This prototype evaluates the Medovik porting model. It does not try to minimize the trusted computing base. Compared with a manual port, the TCB grows to include the WebAssembly runtime and the emulator. That is a real cost. The security tradeoff is more subtle than the size of the TCB alone,

[2]The Jaccard similarity of two sets is the size of their intersection divided by the size of their union. Here, I computed it for the predicted top-k function sets from every pair of runs and then took the mean. A value of 1 means that the sets are identical, while 0 means that they have no function in common.

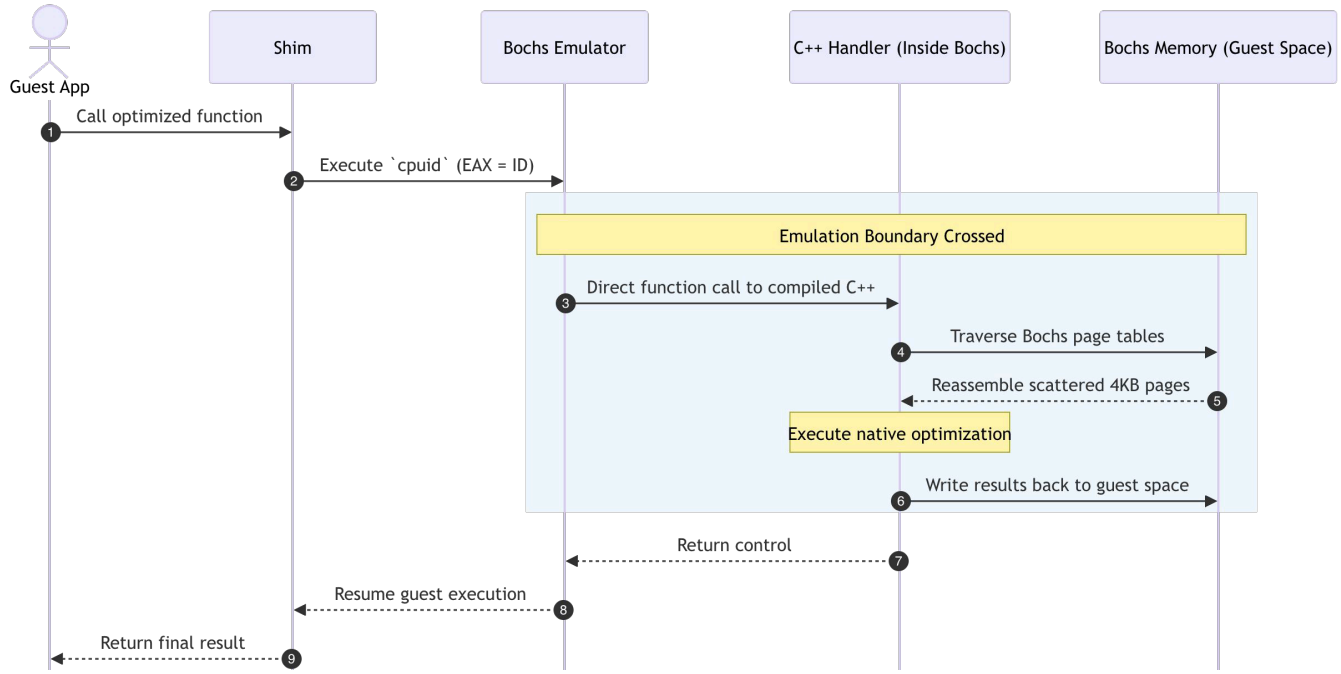

Figure 3.5: Execution sequence of the Medovik punch-through mechanism. A guest shim issues a `cpuid` hypercall. Bochs intercepts it in the WebAssembly environment and dispatches a handler that directly accesses guest memory within Bochs.

though. A manual port can also introduce vulnerabilities, especially when developers have to rewrite large parts of an application and its dependencies for a constrained TEE environment. The prototype therefore explores an engineering tradeoff: accept extra layers so that less code must be manually ported.

The prototype does not yet provide full secure device passthrough. During my internship, a team of undergraduate students ported Wasm3 [63] and WAMR to Nautilus, a TrustZone-enabled research kernel. Nautilus could provide Medovik with a less restrictive alternative to OP-TEEs execution model.

3.4 Results & Evaluation

The evaluation follows the two research questions defined in Section 3.1.4.

3.4.1 Experimental Setup

I ran the main experiments on a RockPro64 [64] with a Rockchip RK3399 SoC and 4 GB of LPDDR4 memory. Since the RK3399 is a heterogeneous big.LITTLE processor, I pinned benchmark processes to the Cortex-A72 performance cores. In my Linux configuration, these are cores 4 and 5. For experiments that enter OP-TEE, I also pinned the Normal World process that invokes the trusted application.

The workloads include CoreMark [65], the NAS benchmark suite [66], selected SPEC CPU2017 workloads [67], and ERASE, the redaction application discussed in §3.1.3. Native and direct WebAssembly runs were measured 30 times. Selected SPEC CPU workloads were run three times, matching the SPEC CPU2017 reportable-run convention, which uses the median time [68]. Full `container2wasm` runs were measured once because they are much slower. I report slowdowns relative to native execution on the same board and core set.

For compilation, I used `clang-21` from Ubuntu 24.04 with `-march=native -O3` for native binaries. Direct WebAssembly binaries were compiled with `wasm32-wasip1-clang` from WASI SDK 28 [69], also with `-O3`. WAMR AOT modules were compiled for the RockPro64 performance cores by setting `wamrc`'s target to `aarch64` and its CPU to `cortex-a72`.

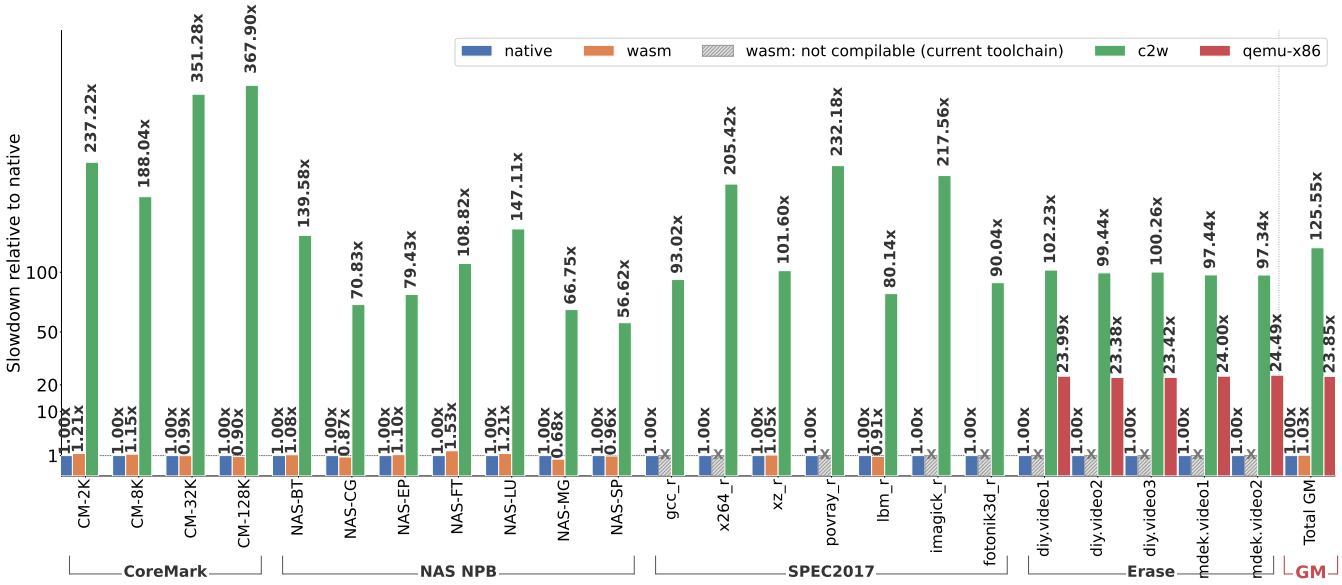

Figure 3.6: Compatibility-first execution requires no porting effort but incurs substantial slowdown. Direct WebAssembly closely matches native performance where supported.

3.4.2 Compatibility-First Execution

Every workload in the final set completed through the compatibility-first path (without any porting work by the application developer) and passed its workload-specific output validation. This included applications that the current direct-WebAssembly toolchain could not compile. Figure 3.6 compares native execution, direct WebAssembly execution, and the full `container2wasm` configuration. That generality came at a geometric-mean slowdown of about 125×, including about 100× for ERASE.

Direct WebAssembly is much closer to native execution.[3] This supports the choice of WebAssembly as a layer in the Medovik prototype. In a few cases, the AOT-compiled WebAssembly version was even slightly faster than the native baseline. The small speedups observed for some WebAssembly workloads likely come from the additional optimization stages in the WebAssembly pipeline. After the WASI SDK compiles the source to WebAssembly, `wamrc` applies another set of LLVM optimization passes while generating AOT code for the Cortex-A72.

For ERASE, I also ran the original x86_64 application through QEMU directly on ARM, without WebAssembly or `container2wasm`. The slowdown fell to approximately 24×. I could not integrate QEMU into the WebAssembly stack during the internship, but this result suggests that the compatibility-first baseline strongly depends on the choice of emulator.

3.4.3 Profiling Hot Paths

After measuring the compatibility-first baseline, I looked at whether the slowdown was concentrated enough for punch-through to help. As described in §3.3.5, I profiled applications in their ordinary execution environment and ranked functions by their share of total runtime. Figure 3.7 shows the result. Across all workloads, the single hottest function accounts for 38.3% of runtime on average. The top three functions account for 69.9%, and the top eight for 84.2%. That concentration is what makes punch-through plausible. The pattern is not uniform. Workloads such as `lbm_r` and `imagick_r` spend roughly 99% of their time in a single function, making them ideal punch-through candidates. `gcc_r` is the opposite case: 85% of execution is spread across thousands of functions, so a small number of handlers cannot recover much performance. ERASE sits between those extremes. Its hot code is less concentrated, but the leading functions still account for enough runtime to make targeted punch-through worth testing.

[3] Here, the workload source is compiled directly to wasm32/WASI when possible, AOT-compiled with WAMR, and run as a WebAssembly binary.

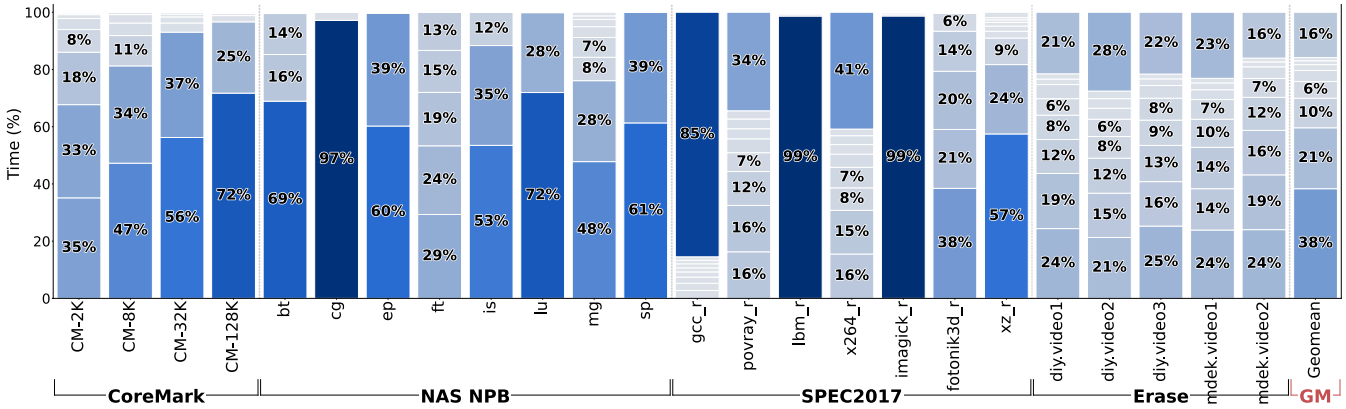

Figure 3.7: Profiled execution time by function across workloads. Blue segments show individual functions or groups, with darker shades indicating larger shares; the light segment combines all remaining functions. Execution time is often concentrated in a small number of hot functions.

3.4.4 Punch-Through Performance

I implemented punch-through for three workloads: `lbm_r`, `imagick_r`, and ERASE. Figure 3.8 shows the performance improvement, while Table 3.1 summarizes the implementation effort and validation. `lbm_r` represents Medovik's best case. One kernel accounts for 98.7% of its execution time, so replacing it removes nearly all the emulation overhead. `imagick_r` shows something different: codebase size matters less than where execution time is concentrated.

Although `imagick_r` contains 173,581 lines of C and C++, the tested workload spends nearly all its time in `Morphology Apply`. I implemented the guest-memory copy and write-back mechanism for `lbm_r`, then reused it for `imagick_r`. After punch-through, their slowdowns fell to 1.15× and 1.56×. Both handlers were LLM-assisted, manually reviewed, and checked against the original outputs.

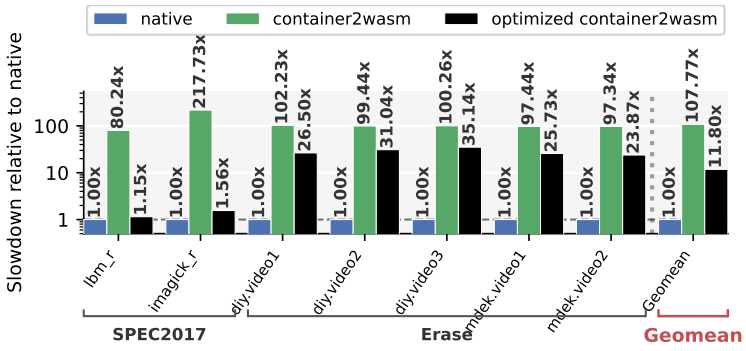

Figure 3.8: Punch-through of selected hot functions substantially reduces slowdown.

ERASE is harder: its profile is less concentrated, and its hottest entry is the Python-level `main` coordinator. I first tried punching through NumPy's `maximum` and `minimum` operations inside `gd_update_efficient_vec`. These operations account for a significant share of its execution time, but punching through them barely helps. Because each operation does little work on its own, the repeated cost of issuing the hypercall, translating and copying guest memory, and writing the result back outweighs the emulation time saved.

So I tried punching through the complete `gd_update_efficient_vec` function instead. It accounts for approximately 19% of runtime and contains 357 lines of Python for updating the bounding boxes. This worked much better. The program crosses the emulation boundary once for the whole function instead of once for every small NumPy operation. The other two replacements were `randomized_svd` and `safe_sparse_dot`, the latter accounting for approximately 12% of runtime. These calls occupy only three lines in ERASE, but they depend on scikit-learn, SciPy, NumPy, CPython, and their compiled native extensions. Those three lines cannot run inside OP-TEE. For `randomized_svd`, I followed the scikit-learn source implementation [70] so that the C++ version preserved the same algorithmic steps and parameter choices. With LLM assistance, I translated the implementation to C++ and used Eigen, a C++ template library for linear algebra [71]. The original Python dependency stack remained inside the emulated Linux guest.

Table 3.1: Application size, punch-through effort, validation, and performance. External runtime and library dependencies retained inside the guest are not included in the application source counts.

Workload	Application source	Punch-through implementation	Preserved code and environment	Validation	Slowdown
`lbm_r`	1,037 C/C++ lines	`LBM_performStreamCollideTRT`; approximately 150 lines	benchmark code, libraries, and Linux guest	Exact output comparison	80.24× → 1.15×
`imagick_r`	173,581 C/C++ lines	`MorphologyApply`; approximately 335 lines; reused the memory-transfer mechanism	`imagick_r` code, libraries, and Linux guest	Exact output comparison	217.73× → 1.56×
ERASE	2,177 Python lines, excluding dependencies	Three handlers; approximately 880 handler lines and 100 lines of Bochs changes; estimated 1–2 developer-days	Most ERASE code, Python and numerical dependencies, and Linux guest	Ground-truth output evaluation	∼99× → ∼28×

The reported ERASE result uses all three punch-throughs. They required more effort than the C/C++ cases, but left the Python runtime and most application logic unchanged, reducing slowdown from approximately 99× to 28×. The improvement is larger than the profiled shares of the functions alone would suggest because the optimized path no longer loads SciPy inside the emulated guest. As shown by the top segments of the ERASE stacked bars in Figure 3.7, Python module initialization accounts for between 16% and 28% of the original execution time.

3.4.5 Discussion

The results support the main idea behind Medovik. Adding layers makes the first execution slower, but it also makes the first execution possible. Once the application runs, profiling can identify a much smaller part of the codebase to port by hand.

The prototype is still limited. Its compatibility-first overhead is high, mostly because of the current Bochs-based stack. It also does not solve secure device passthrough, which remains necessary for a complete ERASE deployment. Still, the experiments show that the model is not only conceptual: unmodified x86_64 Linux applications can run inside an ARM TrustZone setting, and targeted punch-through can recover substantial performance without rewriting the full application.

A remaining limitation of this approach, highlighted by the ERASE evaluation, is the manual engineering effort required to rewrite the target functions for punch-through. Translating complex numerical routines, such as `randomized_svd` and `gd_update_efficient_vec`, from Python to C++ traditionally demands significant developer time and domain expertise. To mitigate this bottleneck, I used LLMs to automate the translation process, first generating a naive C++ implementation and subsequently refining it into an optimized version. Ensuring the semantic correctness of these AI-generated ports required a rigorous validation approach, which directly inspired the secondary research thread explored in the following chapter.

At this point, one could ask whether the selective porting in the Medovik model could itself be automated. An LLM-based workflow could use profiling results to select a hot function, translate it into a punch-through handler, compile it, compare its output with the original, and refine it until the checks pass. My work on the ERASE handlers suggests that this is possible.

This raises another question: if an LLM can automate Medovik's selective porting, why not use it for a traditional TEE port? Recent work has used LLMs to port sensitive leaf functions into TEEs, reporting success rates of 91.8% for Java and 84.3% for Python [72]. However, LLM assistance does not change what a traditional port requires: the application and its dependencies must still conform to the TEE SDK and restricted environment. For ERASE, that remains a whole-stack port. Medovik avoids that application-wide rewrite by preserving the existing stack and limiting the porting work to selected hot functions.

Chapter 4

Side Thread: LLM-driven Codebase Porting

This side project started as a result of one of our weekly Prescience Lab meetings. Professor Dinda was discussing future research ideas related to the Genesis Mission, a U.S. Department of Energy initiative launched to accelerate scientific discovery through artificial intelligence [73].

4.1 From AI for Science to Software Modernization

Several serious examples already exist. AlphaFold changed protein-structure prediction by reaching near-experimental accuracy on many proteins [74]. In materials science, GNoME used deep learning to predict large numbers of candidate crystal structures [75]. In mathematics, FunSearch combined language models with program search to find new constructions and heuristics [76], while AlphaGeometry combined a language model with a symbolic engine to solve Olympiad-level geometry problems [77].

Three of these systems follow the same pattern: the model generates a candidate, and a separate system evaluates it. Density functional theory calculations evaluate GNoME's candidate structures, an evaluator executes and scores FunSearch's programs, and AlphaGeometry delegates deduction steps to symbolic engines.

Porting codebases is a recurring challenge. C and C++ remain common because they give programmers control over layout, allocation, calling conventions, and system interfaces. That control also leaves room for bugs that newer languages try to rule out by construction. A mature codebase is full of implicit knowledge: edge cases, performance assumptions, API contracts, build constraints, undocumented behavior, and tests that only cover part of the real behavior. Static analysis tools can recover some facts about a program, but program meaning is not always written down in a form that can be translated directly, and it is not obvious whether an LLM can truly understand it. What evidence would let us claim that a translation preserves the behavior of the original program? Does it respect the target language's memory model and idioms, and does it preserve assumptions that were never written down explicitly? The equivalence and correctness of the generated program still have to be justified after the fact.

Proving that a program does what it is supposed to do remains one of software engineering's hardest problems [78]. I was not starting from scratch. I had already worked with formal verification in two previous projects on Althread, a language for modeling and verifying concurrent and distributed systems [79]. During this internship, I also took COMP_ENG 356: Introduction to Formal Specification & Verification, where I worked directly with TLA+. TLA+ describes a system as a sequence of states and checks safety and liveness properties of that specification [80]. Between that earlier work and the course, formal methods were one of the first things I considered when I started thinking about how to validate translated code.

TLA+ was not a natural fit for proving that two implementations behave the same way. Lean seemed closer to the problem because programs and functions can be defined in the same language used to state and mechanically prove theorems about their behavior [81]. Still, a proof is useful only if the formal definitions accurately represent the source and translated implementations. For production software involving memory, concurrency, and I/O, building that connection requires substantial proof engineering. Scalability and the need for specialist expertise remain major obstacles [78, 82].

One possible answer is extensive testing. In practice, software is often evaluated by whether it passes its test suite, behaves correctly in deployment, and avoids regressions that users can observe. Good test coverage is part of normal engineering practice, and automated test generation can enlarge the space

```c
int64_t sum_to(int64_t n, int64_t acc) {
    if (n == 0) return acc;
    return sum_to(n - 1, acc + n);
}
```

(a) C: GCC or clang (LLVM) eliminates the recursion at `-O2` in this case.

```julia
function sum_to(n, acc)
    n == 0 && return acc
    return sum_to(n - 1, acc + n)
end
```

(b) Julia: the direct translation remains recursive in the tested version.

Figure 4.1: Program behavior can depend on the compiler pipeline. This tail-recursive C function and its direct Julia translation behave differently on large inputs.

that is tested [83]. When the old and new implementations can be run on the same inputs, differential testing can reveal cases in which their behavior differs [84]. Such evidence might show that the new implementation matches the old one over the tested cases. It would not guarantee that the translation preserved every edge case, every numerical behavior, or every implicit contract in the original code [84].

Figure 4.1 shows a concrete case. In this test, `sum_to(20000000, 0)` completes when the C version is compiled with `-O2`, because GCC turns the recursion into a loop. At `-O0`, the same call crashes with a stack overflow. A direct Julia translation remains recursive and can fail in the same way. As such, the C and Julia versions may both pass ordinary tests, but when the compiler does not eliminate the recursion, a large enough input can exhaust the stack at a limit that depends on the machine and runtime.

4.2 Evaluating LLM-Driven Code Translation

The validation problems described above lead to three questions:

1. Does the translation fix bugs missed by the tests, introduce new ones, or both?
2. How much evidence is enough to trust the translated program?
3. How much human correction is needed, and how much time does the translation ultimately save?

The industry examples, recent research, and my experiments offer different views of these questions.

4.2.1 Industry Signals: Bun and React Compiler

Two recent open-source ports offer an early view of what this approach looks like in practice. In July 2026, Bun maintainer Jarred Sumner published an account of an automated rewrite of Bun's 535,496 lines of Zig into Rust. After 11 days of LLM-driven work, the port passed the existing test suite in continuous integration across six platforms [85, 86]. The project was motivated by recurring memory-management bugs caused by interactions between manually managed memory and JavaScript's garbage collector. According to the maintainer, the Rust version also performed 2-5% better in its benchmarks.

The rewrite was merged into the main branch before its versioned release, and 19 known regressions were later identified and fixed. At the time of Sumner's account, around 4% of the Rust code remained inside `unsafe` blocks, partly to interface with C++ libraries. The result was therefore not a pure safe-Rust rewrite and still depended on unsafe code at some library boundaries.

The initial workflow consumed 5.9 billion uncached input tokens, 690 million output tokens, and 72 billion cached input-token reads. Sumner estimated the total at roughly $165,000 at API pricing. He also estimated that a manual rewrite would have required three engineers with full knowledge of the codebase working for about a year [85]. A rewrite that the project would not have attempted manually became possible at a great but finite inference cost.

As Figure 4.2 illustrates, the rewrite depended on a multi-agent adversarial review process. One agent implemented each translation task, two or more agents searched for errors, and another agent applied the feedback. This is evidence that LLM agents can help port a large codebase when they operate

inside a strict engineering framework. However, there is no formal evidence of correctness. The Rust compiler, a TypeScript test suite with more than 1 million assertions, continuous integration, benchmarks, adversarial LLM review, and manual inspection by the maintainer [85] provide confidence in the port.

Bun is not the only public example. In June 2026, an experimental Rust port of the React Compiler was merged into the React repository [87]. Unlike a standard web application, a compiler must preserve accepted inputs, generated code, error behavior, and diagnostics. The React port was heavily guided by humans but mostly written with AI assistance. To compare it with the TypeScript implementation, the team required all 1,725 existing test fixtures to pass. The validation process compared generated code and errors, as well as the compiler's intermediate representation after each pass.

The React maintainer nevertheless described the port as experimental and warned that undiscovered bugs could remain [87]. Together, the Bun and React cases suggest that LLM-assisted translation can work at production scale when generation is surrounded by deterministic checks and close human supervision. I would argue that passing the complete test suite for a codebase with strong test coverage provides a good degree of confidence.

4.2.2 Research Directions in Code Translation

The industry examples gave me a fairly clear picture of how practitioners approach this problem, so I next looked at what academia was doing. AlphaTrans was my starting point. It treats translation as a repository-level problem, using program analysis to decompose a project and translate its fragments in reverse call order [88]. Although most generated fragments were syntactically correct, AlphaTrans could automatically validate the functional correctness of only about a quarter of them.

PtrTrans focuses more narrowly on project-level C-to-Rust translation. Its argument is that translating one function at a time works poorly for pointers because ownership and lifetime decisions depend on how a pointer is used across the project. To recover this context, PtrTrans builds a pointer knowledge graph containing points-to relationships, ownership, mutability, nullability, and lifetime information [89]. The graph then gives the model a global view of the pointer semantics it must preserve. AlphaTrans and PtrTrans therefore reach a similar conclusion that code translation is not a local problem.

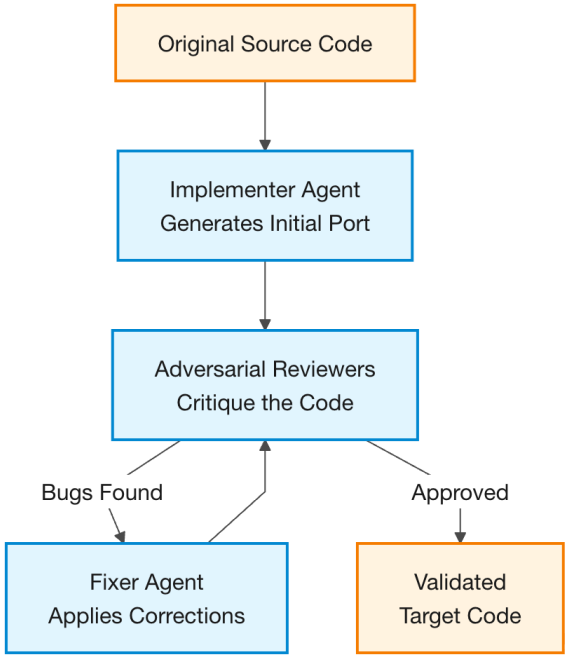

Figure 4.2: Multi-agent validation loop used in the Bun rewrite: an Implementer generates code, Adversarial Reviewers critique it, and a Fixer revises it until all checks pass.

Other work focuses on the information given to the model and on how the result should be judged. SpecTra generates static specifications, test cases, and natural-language descriptions from the source program, filters them for consistency, and supplies them to the model during translation [90]. Across C-to-Rust, C-to-Go, and JavaScript-to-TypeScript tasks, the authors report improvements of up to ten percentage points over translation from source code alone.

CodeAlignBench examines a different part of the problem. A generated program can be functionally correct and still ignore constraints that matter to its developer. The benchmark therefore tests whether models follow the contract stated in the original request and whether they can refine a solution in response to follow-up instructions [91]. Although CodeAlignBench is not a code-translation system, it is making the case that behavioral equivalence, though necessary, is not the only property that makes a translation acceptable.

Table 4.1: Preliminary single-run LULESH throughput for the graph-colored Julia port and the OpenMP C++ implementation. Higher throughput is better.

Threads	Size	Julia (z/s)	C++ OpenMP (z/s)	Julia/C++
44	60	6314.95	5958.47	1.06×
44	80	9045.59	5872.88	1.54×
44	100	9006.75	4869.48	1.85×
44	120	9166.29	5250.65	1.75×
88	60	5163.75	4718.59	1.09×
88	80	6975.31	6421.66	1.09×
88	100	8613.22	4365.09	1.97×
88	120	8905.11	5126.98	1.74×

4.2.3 My Porting Experiments on LULESH

Inspired by these industry signals and by the research direction Professor Dinda had outlined, I built several experimental pipelines around LULESH, the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application [92]. Initially, I wanted to understand whether an LLM could produce Julia code and whether it would be possible to produce external evidence that the translation was trustworthy. Unless I state otherwise, references to "the model" in this section mean Claude Sonnet 4.6 [93].

My first pipeline was naive. I used a classifier to separate the LULESH functions according to whether they perform pure arithmetic, access application state, use MPI, or perform I/O. For the arithmetic kernels, the model generated natural-language specifications, Lean definitions and proofs, Julia implementations, and Z3 checks. Z3 is a satisfiability modulo theories solver that checks logical constraints and searches for counterexamples [94].

The problem was that every formal artifact had been generated from another unverified artifact. A Lean proof established a property of the generated Lean definition, while a Z3 check established a property of the generated logical encoding. Neither shows that the Lean definition matches the original C++ or that the Z3 encoding matches the Julia implementation. The result was a working single-node Julia port that I could probably have generated without any of those extra steps.

Performance was not the goal of this experiment, although I hoped the Julia port would be roughly as fast as the C++ version. It was not. Out of curiosity, I asked the model to examine the Julia code for performance problems and propose optimizations. Its main proposal was an eight-color graph-coloring scheme. Elements assigned the same color do not share a node, so their force contributions can be processed in parallel without conflicting updates. I applied this change before running the measurements reported in Table 4.1.

I compared the optimized Julia version with the OpenMP implementation from the LULESH repository maintained by Lawrence Livermore National Laboratory [95]. I ran the experiments on Dubliner, an Ubuntu server with four Intel Xeon Gold 6238L processors, for a total of 88 physical cores and 176 hardware threads, and 376 GB of RAM. The C++ implementation was compiled with GCC 14.3.0 using `-O3`, `-fopenmp`, and `-DUSE_MPI=0`. I evaluated configurations using 44 and 88 threads with problem sizes 60, 80, 100, and 120. I warmed up the Julia implementation before measurement and excluded problem initialization from the timed region. Since this was only a preliminary test of whether the optimization was promising, I ran each C++ and Julia configuration once. The runs were executed sequentially, and throughput is reported using LULESH's figure of merit in zones per second.

To check the translation, I exposed the original C++ functions through a C-compatible wrapper and compared their outputs with the Julia functions on the same inputs. The complete Julia implementation also retained LULESH's final energy-symmetry check.

At 44 threads, the optimized Julia implementation achieved between 1.06× and 1.85× the C++ throughput. At 88 threads, the range was 1.09× to 1.97×. These results gave me preliminary evidence that an LLM-proposed optimization could make a translation competitive with the original implementation.

I then used DSPy to structure the workflow as a sequence of model-driven stages with explicit inputs and outputs [96, 97]. This made the translation and repair pipeline easier to modify as the experiment evolved. Each attempt received the C++ source, a description of the expected inputs and outputs, and any failure from the previous attempt. The instruction-tuned `google/gemma-4-31b-it` model generated the candidate translations. I accessed it through OpenRouter, a service that provides access to hosted language models through a common API [98, 99]. Claude Sonnet 4.6 helped me write and refine the DSPy signatures and prompts [93].

For the executable specifications, I used `deal`, a design-by-contract library that let me attach preconditions and postconditions directly to Python functions [100]. I checked those contracts with CrossHair, which executes the functions with symbolic inputs and searches for values that violate their preconditions or postconditions [101]. After generating each Julia implementation, I compiled the original C++ function behind a C-compatible interface and compared both versions on randomized inputs. When the outputs differed, I gave the failing input back to the model so it could revise the Julia translation.

This pipeline is more structured than my first attempt. The original trust problem remains. Both the Python specification and the Julia implementation are generated. Contracts and differential testing can do a lot of the heavy lifting. However, these checks do not show that the contracts capture the full behavior of the original C++ code or that the two implementations agree for every input. While this is arguably a better repair loop, it does not amount to a proof of equivalence.

I also explored lowering the C++ and Julia implementations to LLVM Intermediate Representation. Could their equivalence be proved at the IR level? A common representation does not remove the differences between their source languages, compilers, and runtimes [102]. Two frontends may encode the same computation using different control flow, runtime calls, exception mechanisms, garbage-collection interfaces, and floating-point assumptions. An LLVM-level comparison still requires a precise definition of observable behavior and a way to account for those differences.

4.3 Takeaway: Correction Traces as Expert Knowledge

This process gave me more insight into the problem. In a realistic porting effort, a team of software engineers would probably not accept the first generated version. They would review it, run tests, fix obvious bugs, adapt APIs, improve its use of target-language idioms, and make other necessary changes. What if we could extract value from this manual process by recording what was wrong, why it was wrong, how it was detected, and what made the corrected version acceptable? If this process were repeated across several codebases, those traces could become a dataset of expert porting behavior.

Instead of asking an LLM to translate code in one step, we could document how human experts validate and repair translations, then encode that workflow for future agents. A coding agent could be given a porting skill[1], a checking skill, and a repair skill. It could be required to generate tests, compare its output against the source implementation, classify failures, explain its corrections, and only then produce a final version. The model would still generate the code, but more of the expert knowledge would live in the process around it. Existing work [88, 103] provides partial evidence that this direction is plausible.

My next question is whether these correction traces would actually help on codebases the agent had never seen before. I would test this by comparing an agent given only source code, tests, and language documentation with one that also received traces from earlier ports. I would measure how often each agent produced a correct translation, how many repair attempts it needed, and which failures escaped the available checks.

[1] Here, I use a skill to mean a reusable workflow centered on a Markdown instruction file and optionally supported by scripts, references, or other resources.

Chapter 5

Reflections: A Researcher's Life in the Lab

This internship took place in a research lab, and I learned more than the technical content of my projects. I learned new tools, wrote research code, read papers, prepared experiments, helped with an event, wrote a paper, and spent enough time around PhD students to get a glimpse of what their lives look like. Six months is not long enough to fully understand that life, but it is long enough to feel its rhythm.

5.1 Related Work

Research does not happen in isolation. An idea can be measured only compared to what already exists. That usually means reading related work: finding papers, understanding what problem they solve, seeing what they do not solve, and deciding where your own question fits.

This part of research felt very different from my first literature survey four years ago, when I worked on Quality of Service in networks. Back then, I mostly relied on books, search engines, and going through papers one by one. It was slow but taught me to judge a paper's relevance. During this internship, LLMs and LLM-assisted search made the first part of the process much faster. I was able to find leads, related terms, and papers I might not have discovered as quickly alone. I still had to open the papers, check what they really claimed, and decide whether they mattered to my research question. I saw the idea phrased online as: you can outsource your thinking, but not your understanding [104]. That is how I feel about literature review. Tools can speed up search, suggest keywords, and surface adjacent work, but they cannot fully decide for me what a paper means for my own question.

Staying up to date also happens outside papers. LinkedIn, X, blogs, GitHub, and project announcements can all provide useful signals about what people in systems, security, and programming languages are working on.

5.2 Writing, Submission, and Rejection

Writing the Medovik paper made me step back and place the project in context again. I had to be clear about the problem I was trying to solve, what I claimed Medovik solved, and the limitations of the approach. I also had to work within a page limit, follow the venue's formatting and anonymity rules, and present the work in a form that reviewers could evaluate.

The paper was rejected, and I am preparing a revised version for submission to another venue. The reviews were still helpful, and some of their suggestions are already reflected in this thesis. At the same time, I felt that one reviewer had missed the main point of the paper. Professor Dinda asked me to condense each review into bullet points in my own words and then write a rebuttal, even though the workshop did not have a rebuttal period. The exercise taught me to separate my reaction to a review from the concerns it raised, then answer those concerns with facts and concrete changes. Advice on rebuttal writing also helped me see that when a reviewer misunderstands a paper, the framing of the paper may share part of the blame [105].

I came away thinking that peer review is somehow both broken and functional. Reviewers can make useful suggestions and still misunderstand part of a paper. Authors can disagree with a decision and still improve their work because of it. It is not a clean system, but through criticism, revision, and resubmission, it still helps research move forward.

5.3 Talks, Seminars, and Events

The Computer Science department regularly hosts talks at noon, where faculty, PhD students, and external guests present their research. These talks are a useful reminder that research is also a social activity. People explain ideas, ask questions and disagree.

Inside the Prescience Lab, Friday afternoon talks play a similar role. They could be conference reports, rehearsals for future talks, presentations of recent work, or preparation for exams.

On May 11, Northwestern hosted the Greater Chicago Area Systems Research Workshop [106]. More than 300 faculty members, students, and industry guests attended. I helped organize the event, and I presented my first poster [107] on the work I had been doing. Looking back, I realize that I did not use the networking opportunity as well as I could have. Next time, I would try harder to talk to people, ask questions, and use the event as a way to understand the community around the work.

5.4 Can Research Help You Find Purpose?

This question may sound too broad for a masters thesis, but it was part of what I was trying to understand. Could research be a life I would want?

If someone never asks whether their work is useful beyond publications, deadlines, career steps, and reputation, something feels missing. Of course, those things matter. Researchers need jobs, funding, papers, and recognition. But if research becomes only a race toward the next paper, it loses something. Hamming writes that important research is work that matters in the long run [108]. His argument raises an uncomfortable question: am I working on something because it matters, or because it is publishable?

Dijkstra once advised a young researcher, "Do only what only you can do." [109] I think few things can be done by only one person. To me, the sentence means finding work that fits my particular combination of skills, experience, and interests, especially when that combination lets me contribute something that would be harder for someone else. That kind of fit is rare. Perhaps finding purpose means recognizing where I have that advantage and then working to become as good at it as I can.

During this internship, I learned that research can be exciting, lonely, frustrating, and deeply satisfying, sometimes in the same week. Purpose does not simply come from being in a lab, writing a paper, or working on a hard technical problem. For me, it probably comes from deciding that the question is worth my attention, and then giving it that attention honestly.

Chapter 6

Conclusion

This internship began with an open research question rather than an existing implementation: could compatibility layers make unmodified Linux applications run inside an ARM TrustZone environment, and could selective porting recover useful performance? The resulting Medovik prototype answered the first question positively and showed that profiling-guided punch-through can substantially reduce the cost in favorable workloads, although ERASE remains limited by emulation overhead and the absence of secure device passthrough. The work led to a prototype codebase, an evaluation, and a workshop submission. Following the paper's rejection from the 14th Workshop on Programming Languages and Operating Systems (PLOS 2026), held in conjunction with SOSP 2026, we are reconsidering the design, including a possible move to faster QEMU-based emulation and support for device passthrough. I'm also aiming for a revised submission to the 8th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), held in conjunction with SC26. Its deadline is August 14, 2026.

6.1 Hindsight

Beyond the results, this internship gave me a realistic view of academic research and confirmed that I want to continue doing research, whether in academia or elsewhere.

Looking back, there are things I would do differently. Next time I work on a research idea, I want to pause more often and look at the whole picture. When the work is exciting, it is easy to sprint from one technical problem to the next. I should have stopped more often to ask whether the current experiment was answering the right question, whether the implementation was good enough for the claim, and whether I was spending effort in the right place.

I also learned something about autonomy. This internship gave me a lot of freedom, which I appreciated. But freedom also means managing yourself. I had to organize my time, keep progress moving, and decide when to ask for help. I often worked more than the required hours, sometimes by more than ten hours a week. Some of that came from motivation. Some of it came from pressure I put on myself. In the future, I want to stay ambitious without pushing myself too far. Is that actually possible?

The internship also had a personal side. I grew up in Moldova speaking Romanian, spent five years in France, and then moved again, this time to work every day in English. Since the internship was unpaid, I worked part-time on weekends to support myself. Some weeks, the technical demands of the project, financial pressure, and adapting to another country all came at once. I do not mention this as an excuse. I have often had to make something useful out of circumstances I did not choose, and this internship asked the same of me. I am proud of how I handled it. I did the technical work, wrote, presented, learned, adapted, and kept going.

6.2 Personal Direction

A PhD was already a serious option for me before this internship, and it remains one. What changed is that I now have a clearer idea of the kind of work and environment I want. I want to be somewhere where difficult and sometimes slightly crazy technical ideas are taken seriously, where I have the freedom to explore them, and where people care enough to challenge them. I do not yet know whether I will find that in academia, industrial research, or somewhere else.

Chapter 7

Bibliography

[1] Lucian Mocan, Michael Polinski, and Peter Dinda. "Medovik: Simplifying Performant Porting to TEEs by Adding Even More Layers to the Honey Cake". Submitted to the 14th Workshop on Programming Languages and Operating Systems (PLOS 2026), rejected. 2026.

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

[3] Northwestern University. *Facts*. URL: https://www.northwestern.edu/about/facts.html.

[4] Northwestern University. *Faculty Awarded $20 Million Grant from National Science Foundation*. URL: https://www.mccormick.northwestern.edu/images/news/2024/08/faculty-awarded-20-million-grant-from-national-science-foundation-header.jpg.

[5] Northwestern University. *Computer Science at McCormick (Fall 2018)*. URL: https://www.mccormick.northwestern.edu/magazine/images/fall-2018/computer-science-1.jpg.

[6] Northwestern University. *History*. URL: https://www.mccormick.northwestern.edu/about/history.html.

[7] Northwestern University. *Research Areas Computer Science, McCormick School of Engineering*. URL: https://www.mccormick.northwestern.edu/computer-science/research/areas/.

[8] Northwestern University. *Groups & Labs Research, Computer Science, McCormick School of Engineering*. URL: https://www.mccormick.northwestern.edu/computer-science/research/groups-labs.html.

[9] Peter A. Dinda. *Personal website*. URL: http://pdinda.org/.

[10] Northwestern University Prescience Lab. *Prescience Lab Computer Science Research Group*. URL: https://plab.cs.northwestern.edu/.

[11] Constellation Project. *The Constellation Project: Unifying Software and Hardware for Heterogeneous Parallelism*. URL: https://constellation-project.net/.

[12] The Interweaving Project. *The Interweaving Project*. URL: https://interweaving.org/.

[13] Buoyancy Project. *The Buoyancy Project: Exploring Systems and User Approaches to Floating Point Correctness and Resilience*. URL: https://buoyancy-project.org/.

[14] Privacy Backplane Project. *The Privacy Backplane Project: A Legal-Technical Framework for Individualized Privacy Policies in IoT*. URL: https://privacy-backplane.org/.

[15] Ben Wolford. *What is GDPR, the EU's new data protection law?* GDPR.eu. URL: https://gdpr.eu/what-is-gdpr/. Editor in Chief, GDPR.eu; joined Proton in 2018 to help lead the fight for data privacy.

[16] Office of the Attorney General. *California Consumer Privacy Act (CCPA)*. Office of the Attorney General, State of California. URL: https://oag.ca.gov/privacy/ccpa. Office of the Attorney General, State of California; updated March 13, 2024.

[17] Friedrich Doku and Peter Dinda. *TRUSTCHECKPOINTS: Time Betrays Malware for Unconditional Software Root of Trust*. Submitted to IEEE Symposium on Security and Privacy 2027, under review. 2025. arXiv: 2509.22762 [cs.CR]. URL: https://arxiv.org/abs/2509.22762.

[18] Friedrich Doku et al. *CAPIO: Safe Kernel-Bypass of Commodity Devices using Capabilities*. Submitted to EuroSys 2027 (22nd European Conference on Computer Systems), under review. 2025. arXiv: 2512.16957 [cs.CR]. URL: https://arxiv.org/abs/2512.16957.

[19] Nick Gordon. “Secure I/O on Trusted Platforms with Lightweight Kernels”. PhD thesis. Department of Computer Science, University of Pittsburgh, 2024.

[20] Haotian Qiao et al. “Efficient Video Redaction at the Edge: Human Motion Tracking for Privacy Protection”. In: *ACM Transactions on Embedded Computer Systems* 24.5s (Sept. 2025).

[21] Haotian Qiao et al. “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)*. Oct. 2025.

[22] Office of the National Cyber Director. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Tech. rep. 19 pages. The White House, Feb. 2024. URL: https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[23] The Rust Project Developers. *What Is Ownership?* Chapter 4: Understanding Ownership, The Rust Programming Language Book. 2026. URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[24] Gernot Heiser. *The seL4 Microkernel – An Introduction*. seL4 Foundation Whitepaper. Revision 1.4 of 2025-01-08. 2020. URL: https://sel4.systems/About/seL4-whitepaper.pdf.

[25] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[26] Jean Karim Zinzindohoué et al. *HACL*: A Verified Modern Cryptographic Library*. Cryptology ePrint Archive, Paper 2017/536. 2017. URL: https://eprint.iacr.org/2017/536.

[27] Jonathan Protzenko et al. *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*. Cryptology ePrint Archive, Paper 2019/757. 2019. DOI: 10.1109/SP40000.2020.00114. URL: https://eprint.iacr.org/2019/757.

[28] Chromium Project. *Sandbox*. URL: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.

[29] WebAssembly Community Group. *Introduction*. WebAssembly 3.0 Core Specification, version 3.0 (2026-06-25). URL: https://webassembly.github.io/spec/core/intro/introduction.html.

[30] Inc. Docker. *Docker Engine security*. Official documentation page (Docker Engine security overview). URL: https://docs.docker.com/engine/security/.

[31] G. Edward Suh et al. “AEGIS: architecture for tamper-evident and tamper-resistant processing”. In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ICS ’03. San Francisco, CA, USA: Association for Computing Machinery, 2003, pp. 160–171. ISBN: 1581137338. DOI: 10.1145/782814.782838. URL: https://doi.org/10.1145/782814.782838.

[32] GlobalPlatform. *Root of Trust Definitions and Requirements v1.1*. GlobalPlatform. Public Release. June 2018. URL: https://globalplatform.org/wp-content/uploads/2018/07/GP_RoT_Definitions_and_Requirements_v1.1_PublicRelease-2018-06-28.pdf. Document Reference: GP_REQ_025; alignment with TCG noted in revision history; 2018 public release.

[33] Trusted Computing Group. *Trusted Platform Module (TPM) Summary*. Trusted Computing Group. Whitepaper. Apr. 2008. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf. Original publication date from file name (April 29, 2008).

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

[35] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. AMD. 2020. URL: https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.

[36] Robert N.M. Watson et al. " CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization ". In: *2015 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2015, pp. 20–37. DOI: 10.1109/SP.2015.9. URL: https://doi.ieeecomputersociety.org/10.1109/SP.2015.9.

[37] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[38] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 645–658. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[39] Trusted Firmware-A Project. *Feature Overview*. Trusted Firmware-A Documentation. 1.1 Feature Overview (v2.15.0). URL: https://trustedfirmware-a.readthedocs.io/en/latest/about/features.html.

[40] Mengyuan Li et al. "SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 1600–1616. ISBN: 9798400704826. DOI: 10.1145/3634737.3644993. URL: https://doi.org/10.1145/3634737.3644993.

[41] Yuqing Niu et al. *What You Trust Is Insecure: Demystifying How Developers (Mis)Use Trusted Execution Environments in Practice*. 2026. arXiv: 2512.17363 [cs.SE]. URL: https://arxiv.org/abs/2512.17363.

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

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

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

[45] Enarx. *Enarx: Confidential Computing with WebAssembly*. 2025. URL: https://github.com/enarx/enarx.

[46] Youren Shen et al. "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*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 955–970. ISBN: 9781450371025. DOI: 10.1145/3373376.3378469. URL: https://doi.org/10.1145/3373376.3378469.

[47] Richard Habeeb et al. "It's a Non-Stop PARTEE! Practical Multi-Enclave Availability Through Partitioning and Asynchrony". In: *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*. 2025, pp. 228–244. DOI: 10.1109/ACSAC67867.2025.00032.

[48] OP-TEE Documentation. *About OP-TEE*. URL: https://optee.readthedocs.io/en/latest/general/about.html.

[49] Kevin Pedretti. *Kitten Lightweight Kernel Overview*. Tech. rep. SAND2013-10537P. Presentation. Sandia National Laboratories, Dec. 2013. URL: https://www.osti.gov/servlets/purl/1677583.

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

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

[52] The Bochs Project. *Bochs IA-32 Emulator Project*. URL: https://bochs.sourceforge.io/.

[53] Docker. *Docker Documentation*. 2026. URL: https://docs.docker.com/get-started/get-docker/.

[54] Hobbes OS/R Project. *Kitten Lightweight Kernel*. URL: https://github.com/HobbesOSR/kitten.

[55] Jämes Ménétrey et al. "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 1177–1189. DOI: 10.1109/ICDCS54860.2022.00116.

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

[57] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. "Triangulating Python Performance Issues with Scalene". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 51–64. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/berger.

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

[59] Python Software Foundation. *The Python Profilers*. Python 3.14.6 documentation. URL: https://docs.python.org/3/library/profile.html.

[60] Ben Frederickson. *py-spy*. 2026. URL: https://github.com/benfred/py-spy.

[61] Bodhisatwa Chatterjee, Neeraj Jadhav, and Santosh Pande. "Phaedrus: Predicting Dynamic Application Behavior with Lightweight Generative Models and LLMs". In: *Proc. ACM Program. Lang.* 10.OOPSLA1 (Apr. 2026). DOI: 10.1145/3798225. URL: https://doi.org/10.1145/3798225.

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

[63] wasm3 team. *wasm3: A fast WebAssembly interpreter and the most universal WASM runtime*. GitHub repository. URL: https://github.com/wasm3/wasm3.

[64] Pine64. *ROCKPro64 Single Board Computer*. URL: https://pine64.org/devices/rockpro64/.

[65] EEMBC. *CoreMark*. 2009. URL: https://www.eembc.org/coremark/.

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

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

[68] SPEC. *runcpu - Using CPU 2017*. May 2021. URL: https://www.spec.org/cpu2017/docs/runcpu.html.

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

[70] scikit-learn developers. *randomized_svd: Compute a Truncated Randomized SVD*. scikit-learn API Reference. URL: https://scikit-learn.org/stable/modules/generated/sklearn.utils.extmath.randomized_svd.html.

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

[72] Ruidong Han et al. "Automated TEE Adaptation with LLMs: Identifying, Transforming, and Porting Sensitive Functions in Programs". In: *IEEE Transactions on Software Engineering* 52.03 (Mar. 2026), pp. 923–938. ISSN: 1939-3520. DOI: 10.1109/TSE.2026.3655766. URL: https://doi.ieeecomputersociety.org/10.1109/TSE.2026.3655766.

[73] U.S. Department of Energy. *Genesis Mission*. URL: https://genesis.energy.gov/ (visited on 07/03/2026).

[74] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[75] Amil Merchant et al. "Scaling deep learning for materials discovery". In: *Nature* 624 (2023), pp. 80–85. DOI: 10.1038/s41586-023-06735-9. URL: https://doi.org/10.1038/s41586-023-06735-9.

[76] Bernardino Romera-Paredes et al. "Mathematical discoveries from program search with large language models". In: *Nature* 625 (2024), pp. 468–475. DOI: 10.1038/s41586-023-06924-6. URL: https://doi.org/10.1038/s41586-023-06924-6.

[77] Trieu H. Trinh et al. "Solving olympiad geometry without human demonstrations". In: *Nature* 625 (2024), pp. 476–482. DOI: 10.1038/s41586-023-06747-5. URL: https://doi.org/10.1038/s41586-023-06747-5.

[78] Talia Ringer et al. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Found. Trends Program. Lang.* 5.23 (Sept. 2019), pp. 102–281. ISSN: 2325-1107. DOI: 10.1561/2500000045. URL: https://doi.org/10.1561/2500000045.

[79] Althread. *Introduction to Althread*. URL: https://althread.github.io/en/docs/guide/intro/.

[80] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: https://lamport.azurewebsites.net/tla/book.html.

[81] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 1215, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635. ISBN: 978-3-030-79875-8. DOI: 10.1007/978-3-030-79876-5_37. URL: https://doi.org/10.1007/978-3-030-79876-5_37.

[82] Mario Gleirscher and Diego Marmsoler. "Formal methods in dependable systems engineering: a survey of professionals from Europe and North America". In: *Empirical Softw. Engg.* 25.6 (Nov. 2020), pp. 4473–4546. ISSN: 1382-3256. DOI: 10.1007/s10664-020-09836-5. URL: https://doi.org/10.1007/s10664-020-09836-5.

[83] Saswat Anand et al. "An orchestrated survey of methodologies for automated software test case generation". In: *J. Syst. Softw.* 86.8 (Aug. 2013), pp. 1978–2001. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.02.061. URL: https://doi.org/10.1016/j.jss.2013.02.061.

[84] William M. McKeeman. "Differential Testing for Software". In: *Digital Technical Journal* 10.1 (1998), pp. 100–107. URL: https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[85] Jarred Sumner. *Rewriting Bun in Rust*. July 2026. URL: https://bun.com/blog/bun-in-rust.

[86] Jarred Sumner. *Rewrite Bun in Rust*. Merged pull request #30412 in oven-sh/bun. May 2026. URL: https://github.com/oven-sh/bun/pull/30412 (visited on 07/03/2026).

[87] Joseph Savona. *[compiler] Port React Compiler to Rust*. Merged pull request #36173 in react/react on June 9, 2026. Mar. 2026. URL: https://github.com/react/react/pull/36173 (visited on 07/03/2026).

[88] Ali Reza Ibrahimzada et al. "AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation". In: *Proceedings of the ACM on Software Engineering* 2.FSE (2025). DOI: 10.1145/3729379. URL: https://doi.org/10.1145/3729379.

[89] Zhiqiang Yuan et al. "Project-Level C-to-Rust Translation via Pointer Knowledge Graphs". In: *Proc. ACM Softw. Eng.* 3.FSE (June 2026). DOI: 10.1145/3808169. URL: https://doi.org/10.1145/3808169.

[90] Vikram Nitin, Rahul Krishna, and Baishakhi Ray. *SpecTra: Enhancing the Code Translation Ability of Language Models by Generating Multi-Modal Specifications*. Preprint. 2024. DOI: 10.48550/arXiv.2405.18574. arXiv: 2405.18574 [cs.SE]. URL: https://arxiv.org/abs/2405.18574.

[91] Forough Mehralian et al. *CodeAlignBench: Assessing Code Generation Models on Developer-Preferred Code Adjustments*. Preprint. 2025. DOI: 10.48550/arXiv.2510.27565. arXiv: 2510.27565 [cs.SE]. URL: https://arxiv.org/abs/2510.27565.

[92] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Lawrence Livermore National Laboratory, Aug. 2013. URL: https://asc.llnl.gov/codes/proxy-apps/lulesh.

[93] Anthropic. *Introducing Claude Sonnet 4.6*. Feb. 17, 2026. URL: https://www.anthropic.com/news/claude-sonnet-4-6.

[94] Microsoft Research. *Online Z3 Guide: Introduction*. URL: https://microsoft.github.io/z3guide/docs/logic/intro/.

[95] Lawrence Livermore National Laboratory. *LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics*. URL: https://github.com/LLNL/LULESH.

[96] Omar Khattab et al. "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines". In: 2024.

[97] Omar Khattab et al. "Demonstrate-Search-Predict: Composing Retrieval and Language Models for Knowledge-Intensive NLP". In: *arXiv preprint arXiv:2212.14024* (2022).

[98] Google. *Gemma 4 Model Overview*. 2026. URL: https://ai.google.dev/gemma/docs/core.

[99] OpenRouter. *OpenRouter FAQ*. URL: https://openrouter.ai/docs/faq.

[100] Deal Project. *Deal Documentation*. URL: https://deal.readthedocs.io/.

[101] CrossHair Project. *CrossHair: Introduction*. URL: https://crosshair.readthedocs.io/en/latest/introduction.html.

[102] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *International Symposium on Code Generation and Optimization*. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[103] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Proceedings of the 38th International Conference on Neural Information Processing Systems*. NeurIPS '24. Vancouver, BC, Canada: Curran Associates Inc., 2024. ISBN: 9798331314385.

[104] @yacineMTB. *Post on outsourcing thinking and understanding*. X post. 2026. URL: https://x.com/yacineMTB/status/2035510954968887296 (visited on 07/04/2026).

[105] Andreas Zeller. *Patterns for Writing Good Rebuttals*. Oct. 2012. URL: https://andreas-zeller.info/2012/10/01/patterns-for-writing-good-rebuttals.html.

[106] Greater Chicago Area Systems Research Workshop. *GCASR 2026: 13th Greater Chicago Area Systems Research Workshop*. Held on May 11, 2026, at Northwestern University, Evanston, Illinois. May 2026. URL: https://gcasr.org/2026/.

[107] Lucian Mocan et al. "Medovik: A Triple-Layered Approach to Enabling x86 Linux Compatibility in TrustZone". Poster presented at the 13th Greater Chicago Area Systems Research Workshop (GCASR 2026), Northwestern University, Evanston, Illinois. May 2026. URL: https://gcasr.org/2026/pdfs/posters/Mocan_Lucian_MedovikEnablingx86Linux%20-%20Lucian%20Mocan.pdf.

[108] Richard W. Hamming. *You and Your Research*. Bell Communications Research Colloquium Seminar; transcription by J. F. Kaiser. Mar. 1986. URL: https://www.cs.virginia.edu/~robins/YouAndYourResearch.html.

[109] Staci R. Norman. *Edsger Wybe Dijkstra: 1930–2002*. The University of Texas at Austin, Department of Computer Science. Aug. 2002. URL: https://www.cs.utexas.edu/news/2002/edsger-wybe-dijkstra-1930-2002.