Safe Kernel Extensibility and Instrumentation With Webassembly

Faisal Abdelmonem

CMU-CS-25-123
August 2025

Carnegie Mellon University
Computer Science Department
School of Computer Science
Pittsburgh, PA, 15213

Thesis Committee

Anthony Rowe (Chair)
Benjamin Titzer

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Keywords: WebAssembly; kernel extensibility; Linux kernel; kprobes; syscall hooks; in-kernel runtime; sandboxing; eBPF; dynamic instrumentation; observability and debugging; JIT compilation

قَالَ رَسُولُ اللهِ ﷺ: "مَن سَلَكَ طَرِيقًا يَلْتَمِسُ فِيهِ عِلْمًا، سَهَّلَ اللهُ لَهُ بِهِ طَرِيقًا إِلَى الْجَنَّةِ."

رواه الترمذي

The Messenger of Allah ﷺ said: *"Whoever travels a path in search of knowledge, Allah will make easy for him a path to Paradise."*

Reported by Al-Tirmidhi

Abstract

Extending kernel functionality dynamically is essential for modern workloads in observability, profiling, and security, and is becoming increasingly popular for implementing low-latency, kernel-bypass logic in high-performant systems. However, existing mechanisms like kernel modules or eBPF come with steep learning curves, limited expressiveness, or tightly constrained environments. WebAssembly (Wasm), with its strong isolation guarantees, portable semantics, formally defined specification with machine-checked proofs, and low memory footprint, presents a compelling alternative for safe, runtime-extensible logic inside the kernel.

This work explores Wasm as a foundation for safe and flexible kernel extensibility. We present an early prototype that allows users to load and unload Wasm binaries into the kernel and hook them into system calls for interception and instrumentation. This prototype serves as an initial step toward rethinking kernel extensibility using Wasm as a secure and language-agnostic execution layer, enabling safer and more accessible in-kernel customization.

Acknowledgements

First and foremost, all praise and thanks are due to Allah ﷻ, the Most Gracious, the Most Merciful, for His blessings given to me during my studies and in completing this master's degree, for without Him I would not have been able to do any of this. I ask Allah ﷻ that this knowledge benefits me and that I may benefit others with it.

I owe my deepest gratitude to my parents. Their love, support, and encouragement have been the foundation of my life and studies. I can never repay all they have done, and I am blessed to have such loving parents. I pray that Allah ﷻ rewards their sacrifices and grants them the highest level of Jannah.

My heartfelt thanks go to my advisor, Anthony Rowe, for trusting me with this opportunity, and to my co-advisor, Benjamin Titzer, for his invaluable insights. Their combined wisdom, patience, and mentorship have been a constant source of guidance. I would also like to thank Arjun Ramesh for his continuous help and support, as well as the entire WiSE Lab group for making this past year both productive and enjoyable.

I would also like to extend my sincere gratitude to Khaled Harras, Ryan Riley, and Dave Eckhardt. Their guidance from the very beginning and support throughout the application process to the fifth year master's program at Carnegie Mellon University has been pivotal in shaping my academic journey.

Finally, I thank my siblings, grandparents, family, and friends for their steady support. Their encouragement and presence—even from afar—have been a constant source of motivation throughout this work.

Contents

List of Figures

Listings

To my beloved parents…

Chapter 1

Introduction

The Linux kernel is one of the most important and complex pieces of software in the world, designed to be highly generic and adaptable. While this generality makes it suitable for a wide range of use cases, it also presents a challenge for users who wish to modify or extend kernel functionality. For instance, contributing a modification to a syscall or altering the kernel scheduler often requires approval from the kernel community, and even after the code is written and submitted, it can take years for the change to be incorporated into the mainline kernel due to rigorous security and stability checks.

This tension between the need for customization and the rigidity of upstream contributions has led to a growing interest in kernel extensibility—the ability to safely and dynamically inject or modify logic within the kernel without needing to recompile or fork it. Kernel extensibility offers a powerful mechanism for tailoring operating system behavior to the specific needs of userspace applications or system workloads. It enables developers, researchers, and platform engineers to build tooling that operates within the kernel context, allowing for deeper observability, tighter control, and more efficient execution than is possible from userspace alone.

Such extensibility is increasingly important in today's computing landscape. Modern workloads, especially in cloud computing, edge environments, and high-performance systems, often require:

- **Observability and profiling** tools that can track system behavior with minimal overhead.
- **Security mechanisms** that perform in-kernel introspection, access control, or anomaly detection.
- **Protocol acceleration and kernel bypassing** strategies that reduce latency by shortcutting traditional OS paths.
- **Custom scheduling, I/O, or syscall** behavior that may be highly tailored to a specific workload.

Traditionally, this extensibility has been achieved through mechanisms like Loadable Kernel Modules (LKMs), which offer great flexibility but can be dangerous if improperly written. More recently, eBPF (extended Berkeley Packet Filter) has emerged as a safer alternative, allowing sandboxed programs to run within the kernel with strict verification and resource control [1]. However, eBPF has its own limitations: it uses a custom instruction set, has a steep learning

curve, and remains tightly constrained in functionality and expressiveness due to the need for safety and verifiability.

In this thesis, we investigate WebAssembly (Wasm) as a foundation for kernel extensibility; and leverage its portable bytecode format with strong isolation guarantees, formal semantics, and a growing ecosystem of compiler targets and tooling. We propose that Wasm can provide a middle ground between the raw power of kernel modules and the safety of eBPF, enabling secure, efficient, and language-agnostic kernel extensions. Our work explores this hypothesis through the design and implementation of a prototype system that allows users to dynamically load and hook Wasm binaries into Linux kernel syscalls for the purpose of interception and instrumentation [1].

This thesis aims to demonstrate that Wasm is a viable and promising approach to safe kernel extensibility, offering new capabilities for system customization while preserving the security and stability guarantees required in production kernels.

1.1 Paper Outline

The remainder of this paper is organized as follows: Chapter 2 presents background on WebAssembly (Wasm) and the context for our design; Chapter 3 surveys state-of-the-art approaches and identifies their limitations; Chapter 4 details our solution architecture; Chapter 5 describes the prototype evaluation methodology and findings; Chapter 6 outlines future potential and extensions; and Chapter 7 concludes the paper.

[1] Source code available at: https://github.com/Faisal-Saleh/kernel-wasm-runtime

Chapter 2

Background

2.1 Safe Kernel Extensibility

Modern operating systems are expected to support a diverse set of workloads, devices, and environments. Kernel extensibility: the ability to augment or modify kernel behavior without modifying the kernel source code or recompiling the entire kernel. This capability is crucial for enabling specialized behaviors in domains such as security, networking, tracing, and virtualization.

2.1.1 Observability and Instrumentation

Observability refers to the ability to inspect the internal state and behavior of a system from the outside. In modern systems, deep kernel introspection is essential for profiling, debugging, performance tuning, and tracing. Extensible frameworks like eBPF have significantly expanded what is possible in terms of dynamic instrumentation without rebooting or recompiling the kernel [2]. However, they remain constrained by strict safety checks and limited language support.

2.1.2 Security Enforcement

Security policies often require fine-grained enforcement logic [3], such as syscall filtering, access control, and attack detection. Hardcoding these into the kernel is inflexible and often unscalable across diverse deployments. Extensible mechanisms allow tailored logic, sandboxed inspection, and even anomaly detection hooks without touching the kernel core.

2.1.3 Networking and Protocol Offloading

Modern data centers demand high-performance packet processing pipelines, where milliseconds of latency can translate into significant cost. Kernel extensibility enables custom protocol handling, traffic shaping, or flow control logic directly in kernelspace. Technologies like XDP (eXpress Data Path) [4], [5] built on top of eBPF are popular for such use cases.

2.1.4 Kernel Bypass and Low-latency Logic

One major trend in high-performance computing is kernel bypass — avoiding the kernel for data-path operations to minimize latency and maximize throughput (e.g., DPDK, RDMA) [6]–[8]. Yet, some logic still benefits from proximity to the kernel, like metadata tracking or policy checks. Kernel extensibility provides a tradeoff: letting certain lightweight logic run "close to the metal" without hardcoding it.

2.1.5 Research and Experimental Prototyping

Academic and industrial researchers often need to prototype new kernel algorithms — like custom schedulers or memory policies — without waiting on upstream acceptance. Extensible approaches reduce friction and lower the barrier to innovation. They also reduce risks, as experiments can be sandboxed and safely unloaded.

2.2 WebAssembly: A Portable, Safe, and Efficient Runtime

WebAssembly (Wasm) [9] is a low-level virtual machine, renowned for its portability, security, and efficiency. Initially designed as a portable low-level bytecode format for executing code in web browsers, Wasm's applicability has expanded significantly, extending into diverse and performance-sensitive domains. Its rigorous formal specification [10] and robust type system, validated through machine-checked proofs [11], provide a strong foundation for safe execution. Coupled with a rapidly evolving ecosystem of high-performance runtimes [12] and increased emphasis on verification [13], WebAssembly is quickly becoming the preferred technology for secure, flexible, and high-performance applications, making it espicially promising for kernel-level extensibility.

2.2.1 Extensibility

A fundamental advantage of Wasm is its language-agnostic architecture. Serving as a universal compilation target, Wasm supports an extensive array of programming languages, including C, Rust, Go, JavaScript, Ruby, Python, and many more. This broad support enables developers to choose languages best suited to their specific use cases while leveraging Wasm's portability and security features. For example, Rust, a language known for its memory safety, can be compiled into Wasm modules optimized for secure and efficient execution in sensitive environments, such as kernel modules. This flexibility not only enriches the developer ecosystem but also simplifies integration with existing systems.

2.2.2 Security

Security is a foundational aspect of WebAssembly. Wasm modules operate within sandboxed environments, isolating their execution from direct interactions with host system memory. This isolation inherently mitigates common low-level vulnerabilities such as buffer overflows and unauthorized memory accesses, which are major concerns in kernel-level programming. Additionally, WebAssembly's dedication to formal verification and machine-checked proofs

enforces strict safety standards prior to execution, significantly reducing potential vulnerabilities. These robust security guarantees make Wasm particularly compelling for kernel-level implementations, where safety and reliability are critical.

2.2.3 Performance

WebAssembly is typically JIT-compiled [14], enabling near-native execution speeds with minimal overhead. Its compact binary format is designed for fast loading and efficient execution across diverse platforms, making it a strong candidate for performance-critical workloads, including kernel-level applications.

The Wasm specification continues to evolve, incorporating features like SIMD vectorization [15], [16], extended memory addressing through memory64 [17], and more precise memory control via multi-memory [18] and custom page sizes [19]. These continuous improvements make Wasm increasingly capable of leveraging modern hardware, reinforcing its suitability for high-performance, low-level environments such as kernel extensions, where responsiveness and efficiency are essential.

2.2.4 Expanding Ecosystem

Just like performance the entire wasm ecosystem continues to rapidly grow, supported by extensive community and industry involvement. This thriving ecosystem encompasses a broad array of development tools, libraries, and runtime environments [12]. As it continues to evolve, Wasm is becoming more versatile, enabling use cases across various domains. In the context of kernel development, this growing ecosystem offers the potential for more advanced features and improved integration with system components. For example, the ability to use record and replay [20] could enable users to debug the Wasm code running within the kernel. This would represent a significant improvement over the current challenges in debugging kernel modules, where traditional debugging methods are often difficult and inefficient.

Chapter 3

Related Work

3.1 Kernel Modules

The Linux kernel has historically provided Loadable Kernel Modules (LKMs) as a mechanism to dynamically insert custom code into the kernel at runtime without requiring a kernel recompilation or reboot [21]. This functionality has existed since the early 1990s, allowing developers considerable flexibility in extending kernel functionality to support new hardware, filesystem implementations, network protocols, and various other functionalities [22].

Kernel modules are commonly used in scenarios such as:

- Device drivers for supporting hardware not included in the mainline kernel.
- Filesystem implementations, enabling the kernel to interact with various storage solutions.
- Security mechanisms, such as intrusion detection and access control frameworks.
- Networking modules, including custom protocol handling and traffic management.

A simple example excerpt demonstrating the typical structure of a kernel module is as follows:

```c
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void) {
printk(KERN_INFO "Hello, kernel module!\n");
return 0;
}

static void __exit hello_exit(void) {
printk(KERN_INFO "Goodbye, kernel module!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Example Author");
MODULE_DESCRIPTION("A simple Linux kernel module example");
```

Listing 3.1: Kernel Module Example

This example illustrates the straightforward structure and lifecycle management of kernel modules. Typically, the function registered with `module_init` is invoked by the kernel upon insertion of the module, while the corresponding `module_exit` function is called upon its removal. For an in-depth introduction and detailed guidance on developing Linux kernel modules, readers may refer to *The Linux Kernel Module Programming Guide* [23].

3.2 Limitations of Kernel Modules

Although kernel modules offer extensive flexibility, enabling dynamic and powerful customization, they present significant drawbacks and challenges:

3.2.1 Complexity

Kernel module programming is development in the kernel space that requires specialized knowledge of kernel APIs, concurrency management, synchronization primitives, memory management techniques, and kernel internal data structures. Furthermore, a solid understanding of operating systems is essential. Errors in resource allocation or concurrency handling (e.g., deadlocks or race conditions) are prevalent and notoriously difficult to detect due to the absence of comprehensive debugging tools in kernel space [23].

3.2.2 Safety & Security

Kernel modules operate with full kernel privileges, having unrestricted access to kernel internals. Consequently, a minor bug or vulnerability within a module can lead to critical system failures such as kernel panics, data corruption, or severe security breaches. Because modules execute directly in kernel mode, they bypass many safeguards provided to user-space applications, making debugging challenging and potentially dangerous.

Given their unrestricted privileges, kernel modules require rigorous scrutiny before acceptance into kernel repositories. Distributions and organizations often restrict or discourage module loading due to security risks. Module authors must carefully establish and maintain trust, as even well-intentioned but poorly implemented modules pose substantial security threats. Meeting this standard demands deep familiarity with kernel internals, rigorous testing practices, and formal code review—requirements that are unrealistic for the average developer who simply wants to insert a small, task-specific extension into the kernel.

3.2.3 Limited Isolation

Unlike newer extensibility methods such as eBPF and WebAssembly, traditional kernel modules lack inherent isolation. They directly interact with kernel memory and resources, offering no built-in containment. This absence of isolation greatly increases the risk of cascading failures, security exploits, and unintended interactions between independently developed modules.

3.3 Extended Berkeley Packet Filter (eBPF)

The Berkeley Packet Filter (BPF) was originally introduced in the early 1990s as a lightweight, in-kernel virtual machine for efficient packet filtering [24]. The classic BPF design was very minimal: it used a single accumulator and an index register to load packet data, perform comparisons, and return a decision such as "accept" or "drop". This simplicity enabled high-performance filtering but restricted BPF's applicability to networking tasks.

Extended BPF (eBPF) generalizes this model into a 64-bit, register-based virtual machine capable of attaching to a wide range of kernel hooks beyond networking [25]. eBPF programs are typically written in restricted C, compiled to bytecode, and loaded into the kernel via the `bpf()` system call. Before execution, each program must pass a static verifier [26]–[28] that ensures safety properties such as bounded loops, valid memory access, and guaranteed termination. One of the main additions to eBPF was the Just-In-Time (JIT) compiler [29]–[31], which significantly improved execution speed and allowed programs to run at near-native performance.

To support more complex workloads, eBPF provides **maps**—persistent key–value data structures shared between kernel and user space—and a set of **helper functions** that offer controlled access to kernel functionality. Over time, it has expanded to support diverse attach points, ranging from kprobes (discussed in the next section) and tracepoints to sockets and cgroups, enabling use cases in observability, profiling, networking, and security. These capabilities, combined with strong safety guarantees, have established eBPF as a widely adopted mechanism for safe, dynamic kernel extensibility in production systems.

3.4 Limitations of eBPF

Despite strong safety guarantees and broad adoption, eBPF imposes several constraints that limit its applicability in some contexts:

3.4.1 Programming Model and Verification

eBPF adopts a deliberately restricted programming model to enable static verification: programs are written in a constrained C dialect and compiled to a limited instruction set. Features such as unbounded loops, recursion, and dynamic allocation are prohibited or tightly controlled to guarantee memory safety and termination [25]. The in-kernel verifier enforces these properties via conservative static analysis; as a result, programs that are semantically safe may still be rejected if their correctness cannot be proven within the verifier's model, often necessitating non-intuitive code restructurings or additional annotations [32], [33].

3.4.2 Kernel Interaction and Portability

eBPF programs cannot call arbitrary kernel functions. All interaction occurs through predefined helper functions and through persistent key–value data structures (maps). When required functionality is not exposed via helpers, kernel changes are needed before new use cases are

possible [25]. Although BTF improves type introspection and tooling, practical portability remains sensitive to kernel version, configuration, and helper availability across deployments [25].

3.4.3 Performance Considerations

While Just-In-Time compilation enables near-native execution for many workloads, certain patterns—such as frequent map operations, heavy helper usage, or large data copies—can introduce measurable overhead. Recent compiler work targets these costs with code-generation and code-size optimizations, but performance remains workload-dependent [34]

Chapter 4

Solution Architecture

4.1 Goals and Assumptions

The primary goal of this work is to design and implement a proof-of-concept framework for safe, language-agnostic kernel extensibility using WebAssembly as the execution environment. The prototype aims to demonstrate that small, task-specific extensions can be dynamically loaded, executed, and removed from the kernel without compromising stability, while retaining sufficient performance for practical use. The focus is on feasibility and correctness rather than exhaustive optimization.

The system is developed under the following assumptions:

- The kernel module and supporting user-space tooling are deployed on a Linux `PREEMPT_RT 6.6-rt`.
- WebAssembly modules are assumed to be authored by trusted developers within the deployment context; malicious inputs are out of scope for this prototype. Nevertheless, the embedded Wasm runtime still enforces memory-safety and isolation (e.g., bounds-checked linear memory and no direct access to kernel pointers).
- Extensions are short-running and side-effect-free beyond their intended functionality, avoiding blocking operations or unbounded computation.
- The prototype relies on existing kernel instrumentation mechanisms (e.g., kprobes) and does not require modifications to the core kernel.
- Success is defined by correct hook invocation, safe load/unload cycles, and bounded execution without kernel faults.

4.2 Kernel Hooks and Wasm Integration

4.2.1 Kprobes

Kprobes are a Linux dynamic instrumentation feature, introduced in 2005 [35]–[37] and commonly used from loadable kernel modules. They allow developers to hook into kernel functions—including system-call entry points—without modifying kernel source or rebuilding

the kernel, providing a practical way to observe or interpose on execution with minimal intrusion. Later frameworks such as eBPF capitalized on the same idea by offering attachment to similar hook points; our prototype follows this approach by placing lightweight hooks at selected symbols to trigger custom logic.

Conceptually, a kprobe associates a handler with a specific instruction address, typically identified by a symbol name. When execution reaches that address, the kprobe framework temporarily traps control to a registered handler and then resumes the original flow. In practice, a probe is described by a `struct kprobe` (e.g., fields `.symbol_name` or `.addr`, and handler pointers `.pre_handler` / `.post_handler`) and installed with `register_kprobe()`, then removed with `unregister_kprobe()`. A related mechanism, *kretprobe*, attaches at function return. Because handlers may run in atomic context, they must not sleep and should keep work minimal. A minimal usage pattern is shown in Listing 4.1, and the control flow—breakpoint trap, pre-handler, single-step of the original instruction, post-handler, and resume—is illustrated in Figure 4.1.

Minimal API example.

```c
#include <linux/kprobes.h>

static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    /* Inspect arguments via regs; do minimal, non-sleeping work. */
    return 0; /* Return 0 to continue normal processing. */
}

/* Note: post_handler signature includes flags. */
static void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    /* Optional: observe state after the probed instruction executed. */
}

static struct kprobe kp = {
    .symbol_name = "__x64_sys_mkdir", /* Any resolvable kernel symbol. */
    .pre_handler = handler_pre,
    .post_handler = handler_post,
};

static int __init mod_init(void)
{
    int ret = register_kprobe(&kp);
    return ret;
}

static void __exit mod_exit(void)
{
    unregister_kprobe(&kp);
}
```

Listing 4.1: Registering a basic kprobe with pre/post handlers

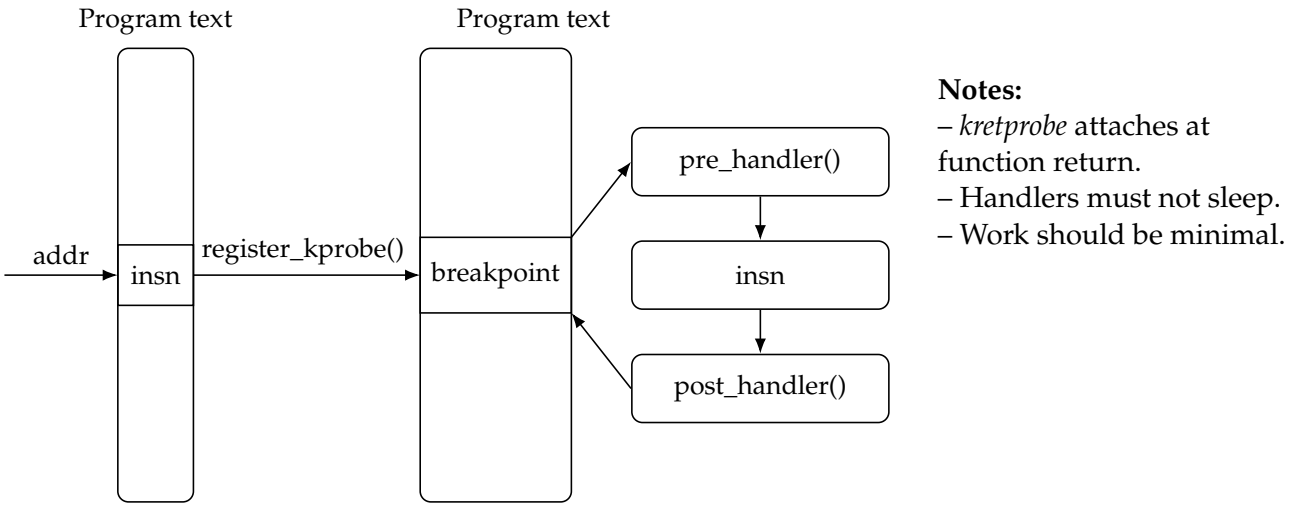

Figure 4.1: Kprobe control flow at a probed kernel symbol (address).

4.2.2 Symbol Selection

Selecting appropriate probe points is central to correctness and stability. A Linux system call typically enters through an architecture-specific wrapper before invoking a chain of internal helpers for security checks, argument translation, and subsystem handoffs.

In this prototype, we deliberately target only the *outermost* syscall entry point. On x86_64, these functions generally follow the naming convention:

```
__x86_sys_(sysname)
```

(e.g., `__x86_sys_mkdir`, `__x86_sys_openat`). Attaching at this boundary captures the raw userspace arguments before any internal transformation, avoiding reconstruction later and reducing sensitivity to changes in downstream helpers. These syscall entry points tend to remain relatively stable across kernel versions compared to deeper internal helpers, making them a more robust choice for long-term maintenance. This approach yields semantically stable hook points while keeping the instrumentation surface small and well-defined.

Current Prototype Behavior

In the current implementation, symbol resolution and suitability checks are not enforced by the module. It is assumed that users supply a valid, probeable symbol name (e.g., a visible, non-inlined function that is not on the kprobe blacklist). The activation path records the requested symbol and installs the pre-handler accordingly. Hardening this step—e.g., rejecting blacklisted or inlined targets, validating availability, and screening unsafe contexts—is left as future work.

4.2.3 Wasm Runtime Embedding

We embed the `wasm3` [38] engine directly in the kernel module. For each WebAssembly (Wasm) binary loaded, the module creates a dedicated *environment* and *runtime* instance, each with its

own linear memory. This per-module instantiation isolates state across extensions and avoids interference between concurrently loaded modules. During activation, the module loads the Wasm bytecode, resolves the target function(s) once (e.g., via `m3_FindFunction`), and records the resulting function handles in an internal registry alongside the selected probe/symbol. No host functions (imports) are exposed to Wasm in this prototype; Wasm code only has access to values passed as arguments.

Invocation path.

When a kprobe pre-handler fires, the handler performs a lookup in the registry to find the bound Wasm function for the current symbol and, if active, invokes it through a thin adapter (`wasm_call`). The adapter prepares the call arguments from `struct pt_regs` (e.g., syscall arguments available at the probe point), places them into the instance's linear memory or call frame as required by the resolved function's signature, and executes the Wasm function on the corresponding runtime. Because function resolution is done at load time, the fast path avoids repeated symbol or function lookups. Figure 4.2 demonstrates the invocation path when the probe is active and found.

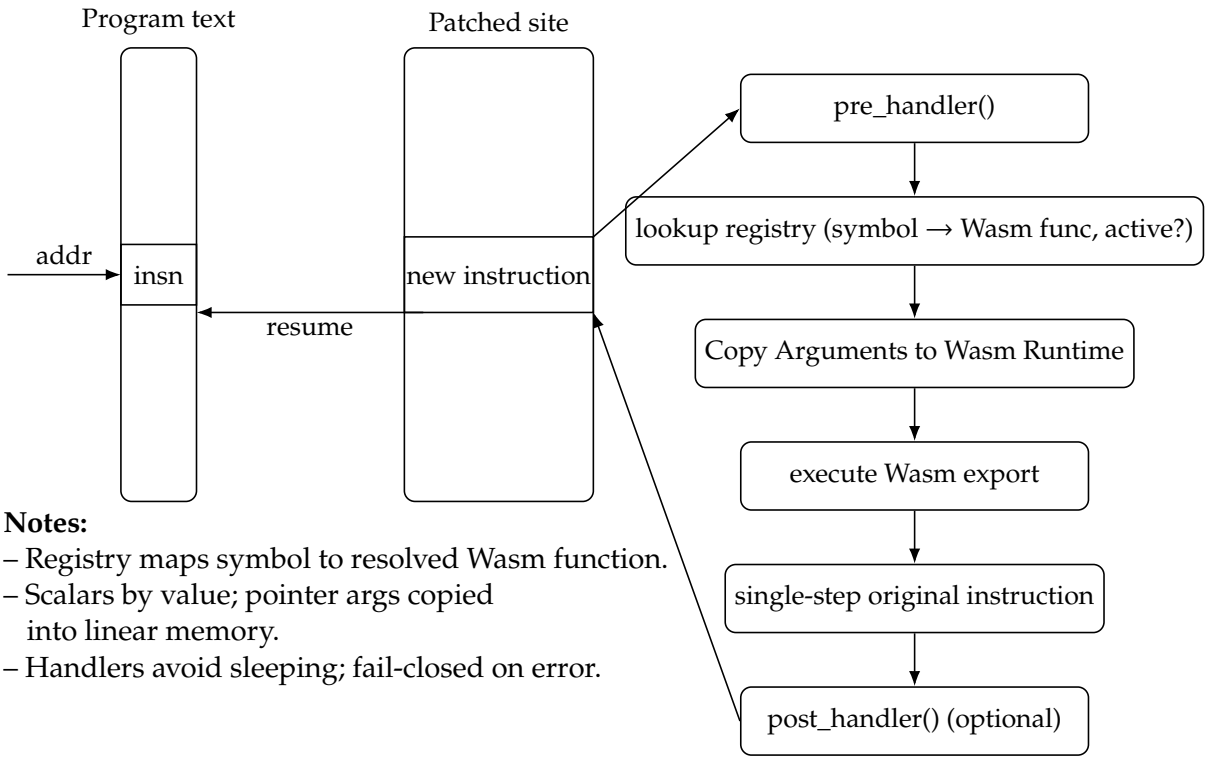

Figure 4.2: Wasm module invocation path

Safety and Error Handling

All execution occurs inside the `wasm3` sandbox; the prototype does not export kernel pointers to Wasm and does not provide host imports. If a call fails (e.g., trap, internal error), the handler

follows a fail-closed policy: it logs the error and resumes normal kernel execution without altering the control path. The pre-handler guards access to the probe registry with a short critical section (e.g., `spin_lock_irqsave`) and releases the lock before returning. In the event of unrecoverable conditions (e.g., allocation failure), the handler returns a negative error code after unlocking, allowing the kernel to proceed while signaling the failure to the caller.

Limitations (current prototype)

Resource limits (e.g., instruction budgets, timeouts, per-instance memory caps) are not enforced beyond `wasm3`'s intrinsic checks, and the host ABI is intentionally minimal (no imports). Extending the embedding with bounded copies for complex arguments, explicit execution limits, and a small set of audited host helpers is left to future work.

4.3 Wasm Module Structure

This section outlines the shape of a WebAssembly (Wasm) module used by the prototype. Modules are compiled to Wasm (from C in our case) and expose a small, explicit interface: (i) named exports corresponding to kprobe hooks ("pre"/"post" at the outermost syscall entry), and (ii) a utility export `report` used to return a short, human-readable status string. No host imports are required or exposed; all execution happens inside the embedded interpreter with the module's own linear memory.

Minimal Wasm module.

```c

__attribute__((used)) char buffer[256];
__attribute__((used)) char report_buffer[256];
__attribute__((used)) unsigned int buffer_size = sizeof(buffer);

static const char *g_pathname = 0;

__attribute__((export_name("kprobe:__x86_sys_mkdir:pre")))
void func1(const char *pathname, unsigned short mode) {
    /* Opaque pointer value: stored for correlation, not dereferenced. */
    g_pathname = pathname;
}

__attribute__((export_name("kprobe:__x86_sys_mkdir:post")))
void func2(const char *pathname, unsigned short mode) {
    /* Symmetric post-hook (optional). */
    g_pathname = pathname;
}

__attribute__((export_name("report")))
const char *report(void) {
    /* Returns a pointer into this module's linear memory. */
    return "the report string";
}
```

Listing 4.2: wasm module format

4.3.1 Exports and Naming

The export naming scheme is inspired by DTrace-style probe identifiers. Each hook is exported using the pattern

`kprobe:hook:pre` or `kprobe:hook:post`,

where `hook` denotes the outermost syscall name (i.e., the architecture-specific syscall entry symbol). This convention lets the kernel module resolve the correct function once at load time and bind it to the selected probe point. currently, the prototype only targets syscalls, so the `kprobe` provider prefix is fixed; however, the scheme is intentionally extensible to additional providers (e.g., tracepoints or fentry/fexit) in future work.

4.3.2 Data Objects

The static arrays (e.g., `buffer`, `report_buffer`) are not general-purpose data structures for the module author; they serve as preallocated regions in the Wasm linear memory that the kernel module uses to marshal arguments and collect results. Preallocation avoids dynamic allocation on the probe fast path and ensures that copies into the module's memory are bounded. The symbol `buffer_size` advertises the maximum capacity available for inbound copies (e.g., pathnames).

4.3.3 Arguments

Scalar syscall arguments (e.g., flags, modes) are passed directly to the exported functions. For pointer-typed arguments (e.g., strings such as pathnames), the kernel module *does not* pass raw kernel or userspace pointers. Instead, on each hook it copies the pointed-to data into the

module's preallocated `buffer` (up to `buffer_size`, with NUL termination as appropriate) and then passes a pointer (i.e., an offset within the module's linear memory) to the Wasm function. From the module's perspective, this pointer is fully dereferenceable and can be treated as a normal C pointer into its own memory. The copy direction is host→module (for arguments); modifications performed by the Wasm code do not propagate back to kernel memory unless an explicit copy-back is implemented.

4.3.4 Call/Return Contract

At load time, the kernel module resolves the exported hook names and the `report` function (one per Wasm module). On a kprobe fire, the corresponding pre/post (if exported) is invoked with the syscall arguments—scalars passed by value, and pointer-typed arguments passed as pointers *to copies* placed in the module's linear memory. For `report`, the kernel calls `report()` via the interpreter, obtains a pointer (offset) into the module's linear memory, and copies a bounded, NUL-terminated string back to userspace. In practice, modules write into `report_buffer` and return its address; the kernel side enforces a maximum length (e.g., via a known buffer size and bounded `strnlen`) to ensure copies are clamped.

4.4 Architecture Overview

At a high level, the prototype comprises a userspace control tool, a kernel module that embeds a *wasm3* runtime, and kprobe-based hooks at selected syscall entry points. Wasm modules expose fixed, *optional* exports—`kprobe:hook:pre`, `kprobe:hook:post` (where `hook` is the outermost syscall symbol, e.g., `__x86_sys_openat`), and `report`. The kernel module resolves whatever of these exports are present at load time and stores the resulting function handles in a small registry. The design separates a *control path* for lifecycle operations from a *data path* that executes on hook invocations.

4.4.1 Control Path

Userspace issues `ioctl` commands on a character device to *load*, *activate*, *deactivate*, and *unload* extensions. On *load*, the module copies the Wasm binary from userspace into kernel memory, creates a per-module *wasm3* environment/runtime with private linear memory, resolves any available exports (`kprobe:...:pre/:post`, `report`), and records the bindings. *Activate/deactivate* attach or detach the kprobe at the chosen outermost syscall symbol; *unload* tears down the instance and frees state. The same interface provides a *report* operation (when the `report` export exists): the module calls `report()`, copies the returned string from the instance's linear memory into a bounded kernel buffer, and returns it to userspace via `copy_to_user`. If an export is not provided, the corresponding operation is simply unavailable while the others continue to function.

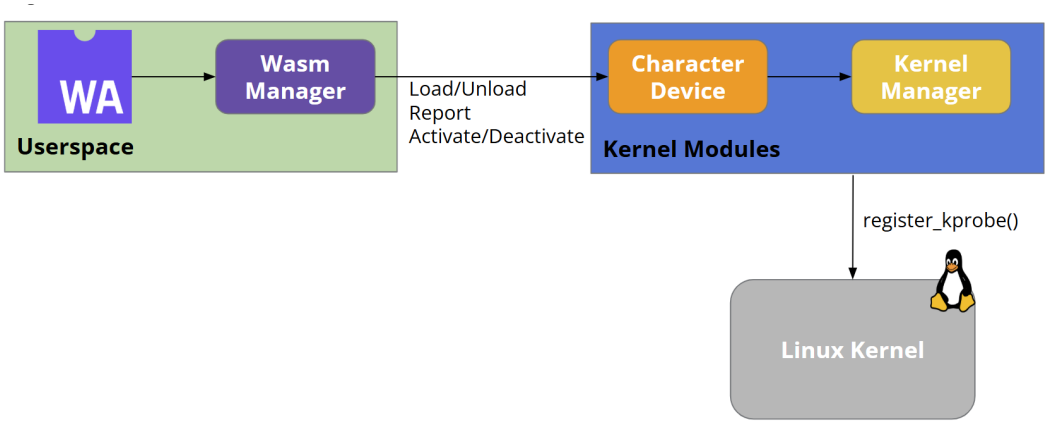

Figure 4.3: A depiction of how control reaches from the user to the kernel

4.4.2 Data Path

When a probed syscall entry is reached, the pre/post handler performs a lookup in the registry and, if the corresponding export was resolved, invokes it. Scalars are passed by value. For pointer-typed syscall arguments (e.g., pathnames), the handler first copies the pointed-to data into a preallocated buffer in the module's linear memory (up to a known bound) and then passes the address of that copy to the Wasm function. No host imports are exposed; execution remains within the interpreter, handlers avoid sleeping and heavy work, and errors (e.g., a trap) are handled fail-closed: the kernel resumes the unmodified path after logging.

This arrangement keeps configuration and symbol resolution on the control path, leaves the fast path minimal and predictable, and leverages a simple, optional export convention to bind syscall hooks cleanly without modifying kernel sources or rebooting.

Chapter 5

Prototype Evaluation and Validation

This chapter validates the prototype's core claims:

(i) hooks at outermost syscall entry points fire reliably,

(ii) scalar and pointer arguments are marshalled correctly into Wasm linear memory, and

(iii) the control/data path separation preserves stability under typical use.

The emphasis is on functional correctness rather than exhaustive performance benchmarking.

5.1 Methodology and Environment

All experiments were run on a controlled Linux environment (matching the assumptions in Chapter 4). Wasm modules were compiled from C to Wasm and loaded via the userspace control tool; hooks were attached at the outermost syscall entries (e.g., `__x86_sys_mkdir`, `__x86_sys_openat`, and the process-creation entry appropriate to the kernel). Results were retrieved using the module's `report()` export and were validated against ebpf ground-truths run using `bpftrace`. Kernel logs (`demsg`) were monitored for errors.

5.2 Test Cases and Protocol

T1: Counter

Goal: verify that a hook is invoked correctly once the probe triggers.

Protocol: load the hook, spawn X child processes (using `fork`), deactivate the hook, spawn Y additional children, reactivate the hook, and spawn Z more children. call `report()` and compare to $X + Z$ and the `bpftrace` output.

This was also done to test activate/deactivate work correctly.

it is also important to note that this test setup was also run with `mkdir` and `openat` syscalls.

Success: reported count equals the number of successful children; clean deactivate/unload; no kernel faults.

T2: Mkdir Mode Capture

Goal: validate argument marshalling for a pointer (`pathname`) and a scalar (`mode`).

Protocol: invoke `mkdir` in the Wasm pre-hook, read the copied pathname from linear memory and the scalar mode; summarize via `report()`, and compare them with the `eBPF` ground-truths.

Success: paths are intact and NUL-terminated; reported string matches eBPF results; no out-of-bound copies.

T3: Accept Under a Reverse Proxy (Cache On/Off)

Goal: demonstrate debugging utility: with the proxy cache warmed, `accept` should trigger once for a request sequence; with cache disabled, it triggers per connection.

Protocol: enable cache, warm with M requests, record `accept` count; disable cache, repeat; compare.

Success: markedly lower `accept` count with cache enabled than disabled; hook stable across both runs.

5.3 Discussion and Reliability Checks

In T1 and T2, counters and captured arguments matched ground truth, indicating that pre-handlers fire once per syscall entry, scalar arguments are passed by value correctly, and pointer-typed arguments are safely copied into the module's linear memory with bounded lengths. In T4, the `accept` counts reflected expected proxy behavior, illustrating the system's usefulness for diagnosing configuration assumptions. Throughout all tests, lifecycle operations (*load/activate/deactivate/unload*) executed without kernel faults; handler errors (if any) were logged and handled fail-closed.

5.4 Summary

The evaluation supports the prototype's central claim: small, task-specific extensions can be dynamically loaded, invoked at syscall entry, and removed while preserving kernel stability. The observed behavior across the four scenarios demonstrates reliable hook firing, correct argument marshalling, and robust lifecycle handling, providing a solid foundation for the future extensions outlined in Chapter 6.

The full set of test scripts, Wasm probes, and equivalent eBPF probes used in this evaluation are available in our public repository [39].

Chapter 6

Future Potential

This prototype demonstrates the feasibility of safe, language-agnostic kernel extensibility with WebAssembly. Looking ahead, its value grows with stronger tooling, faster runtimes, and smoother integration paths.

6.1 Expanding Wasm Ecosystem

Embedding a WebAssembly runtime in the kernel opens the door to a wide language ecosystem (C/C++, Rust, Go, etc.) and mature tooling. In particular, *record–reduce–replay* for Wasm can make kernel-resident extensions far easier to debug and validate: Wasm-R3 instruments modules to record executions, aggressively reduces traces, and emits replayable benchmarks that run on any engine *without* the original host environment [40]. Integrating such a workflow would let developers capture real workloads in production, reproduce them offline, and iteratively refine kernel hooks and Wasm logic with high fidelity. This complements the prototype's fail-closed design by adding a practical path to deterministic debugging and regression testing.

6.2 Integrating With WHAMM

Beyond ad-hoc hooks, WHAMM proposes a DSL for programmable Wasm instrumentation with declarative match rules, static/dynamic predication, and automatic state reporting, delivering monitors either by bytecode rewriting or by interfacing directly with the engine [41]. Because WHAMM produces instrumentation *as Wasm*, your kernel runtime could load these monitors like any other module, reusing engine optimizations while keeping the ABI narrow. This promises richer, policy-driven probes (e.g., conditional sampling, selective argument capture) with compiler-assisted overhead reduction.

6.3 Adopting a JIT Runtime

Moving from `wasm3` (interpreter) to Wasmtime would unlock JIT compilation via Cranelift and a steady stream of backend optimizations (e.g., compile-time optimization modes, IR improvements) while retaining strong sandboxing [42]. In principle, JITed hot paths should

cut per-hook latency for compute-heavy modules and broaden the set of viable in-kernel analyses. Engine-level features (e.g., better codegen on x86_64/aarch64, evolving IR support) also position the system for longer-term maintainability and performance portability.

6.4 Compatibility Path: eBPF → Wasm

To lower adoption friction, a translation path from eBPF bytecode to Wasm would let existing programs run under the proposed Wasm-in-kernel runtime with minimal rewrite. Early ecosystem efforts already bridge these worlds: toolchains that package eBPF logic as Wasm modules for cross-platform execution, and runtimes that execute eBPF compiled to Wasm [43], [44]. Building a robust eBPF→Wasm compiler for your system would preserve familiar semantics (maps, helpers via a thin compatibility layer), ease migration of legacy code, and consolidate extensibility on a single, portable sandbox.

Chapter 7

Conclusion

This work presented a proof-of-concept framework for safe, language-agnostic kernel extensibility by embedding a WebAssembly runtime in a Linux kernel module and triggering Wasm functions from kprobe hooks at outermost syscall entry points. The architecture cleanly separates a control path (load/activate/deactivate/unload/report via a character device and `ioctl`) from a data path (pre/post handlers that marshal syscall arguments into Wasm linear memory and invoke resolved exports). The prototype keeps the fast path minimal (no sleeping, bounded copies, fail-closed on errors), avoids exposing kernel pointers to Wasm, and binds hooks using a simple, optional export convention (`kprobe:hook:pre/:post` and `report`). Functional validation across four scenarios (fork counter, `mkdir` counter, `mkdir` mode capture, and `accept` under proxy cache on/off) demonstrated that hooks fire reliably, scalars and strings are marshalled correctly, and lifecycle operations execute without kernel faults.

A core contribution is showing that WebAssembly can serve as a practical, uniform sandbox for in-kernel extensions while remaining developer-friendly. Embedding a WebAssembly runtime in the kernel enables *polyglot* extensibility: developers can author extensions in any language that compiles to Wasm (e.g., C/C++, Rust, Go, Python), reusing mature build and testing ecosystems. This stands in contrast to traditional approaches—kernel modules and eBPF—which are effectively limited to C (for eBPF, a restricted C dialect), narrowing both the contributor base and available tooling [9], [22], [25]. Polyglot support not only improves productivity; it also permits adopting language-level safety properties (e.g., Rust's ownership and borrowing) while preserving a consistent sandboxed execution boundary in the kernel.

The current prototype has clear limitations: there is no verifier beyond the interpreter's intrinsic checks; resource limits (instruction budgets/timeouts) are not yet enforced; the host ABI is intentionally minimal (no imports); and we rely on an interpreter (`wasm3`) rather than a JIT. These choices were deliberate to keep the design small and auditable but leave performance and expressiveness on the table. The evaluation was likewise scoped to functional correctness, not exhaustive benchmarking.

Looking forward, the path to a production-ready system is compelling. First, migrating to a JIT-enabled engine such as Wasmtime would leverage ongoing compiler optimizations while preserving sandboxing. Second, integrating record–reduce–replay for Wasm would add a practical route to deterministic debugging and regression testing for kernel-resident extensions. Third, compiler-driven instrumentation (e.g., WHAMM) could generate policy-rich monitors

as Wasm modules without expanding the kernel ABI. Finally, an eBPF→Wasm translation path would provide continuity for existing programs and ease adoption. Together, these directions move the design from a feasibility prototype toward a robust platform for safe, maintainable, and widely accessible kernel extensibility.

Bibliography

[1] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström, "The ebpf runtime in the linux kernel", *arXiv preprint arXiv:2410.00026*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.00026.

[2] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 2019, ISBN: 978-0136554820.

[3] S. Smalley, P. Loscocco, W. A. Morrison, G. Kroah-Hartman, J. Morris, and S. Wright, "Linux security modules: General security support for the linux kernel", in *USENIX Security Symposium*, 2003.

[4] J. D. Brouer and T. Høiland-Jørgensen, "XDP – express data path: High performance packet processing in the linux kernel", in *Netdev 2.1, The Technical Conference on Linux Networking*, 2017. [Online]. Available: https://netdevconf.info/2.1/papers/XDP-presentation-netdev-2.1.pdf.

[5] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel", in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "{Ix}: A protected dataplane operating system for high throughput and low latency", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 49–65.

[7] *Data plane development kit (dpdk)*, https://www.dpdk.org/, Linux Foundation project page, accessed 2025-08-16.

[8] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification", Tech. Rep., 2007.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly", in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 185–200.

[10] WebAssembly Community Group, *Webassembly specifications*, https://webassembly.github.io/spec/, Accessed: 2025-08-04, 2021.

[11] C. Watt, "Mechanising and verifying the webassembly specification", in *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, 2018, pp. 53–65.

[12] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, "Research on webassembly runtimes: A survey", *ACM Transactions on Software Engineering and Methodology*, 2024.

[13] E. Bosman and H. Bos, "Framing signals-a return to portable shellcode", in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 243–258.

[14] B. L. Titzer, "Whose baseline compiler is it anyway?", in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2024, pp. 207–220.
[15] WebAssembly Community Group, *Webassembly 128-bit packed simd extension*, https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md, Accessed: 2025-08-05, 2021.
[16] WebAssembly Community Group, *Webassembly relaxed simd proposal*, https://github.com/WebAssembly/relaxed-simd, Accessed: 2025-08-05, 2024.
[17] WebAssembly Community Group, *Webassembly memory-64 proposal*, https://github.com/WebAssembly/memory64, Accessed: 2025-08-05, 2024.
[18] WebAssembly Community Group, *Webassembly multi memory proposal*, https://github.com/WebAssembly/multi-memory, Accessed: 2025-08-05, 2024.
[19] WebAssembly Community Group, *Webassembly custom page sizes proposal*, https://github.com/WebAssembly/custom-page-sizes, Accessed: 2025-08-05, 2024.
[20] D. Baek, J. Getz, Y. Sim, D. Lehmann, B. L. Titzer, S. Ryu, and M. Pradel, "Wasm-r3: Record-reduce-replay for realistic and standalone webassembly benchmarks", *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2156–2182, 2024.
[21] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd. O'Reilly Media, Inc., 2005, ISBN: 9780596005900.
[22] R. Love, *Linux Kernel Development*, 3rd. Addison-Wesley Professional, 2010, ISBN: 9780672329463.
[23] P. J. Salzman, M. Burian, and O. Pomerantz, *The linux kernel module programming guide*, https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf, Accessed: 2025-08-06.
[24] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture.", in *USENIX winter*, vol. 46, 1993, pp. 259–270.
[25] L. Rice, *Learning eBPF*. O'Reilly Media, 2022.
[26] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking", in *OSDI*, vol. 96, 1996, pp. 229–243.
[27] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the verifier: Ebpf range analysis verification", in *International Conference on Computer Aided Verification*, Springer, 2023, pp. 226–251.
[28] H. Sun and Z. Su, "Validating the {ebpf} verifier via state embedding", in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 615–628.
[29] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: A trustworthy {in-kernel} interpreter infrastructure", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 33–47.
[30] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing jit compilers for in-kernel dsls", in *International Conference on Computer Aided Verification*, Springer, 2020, pp. 564–586.
[31] L. Nelson, J. Van Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to {bpf} just-in-time compilers in the linux kernel", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 41–61.
[32] J. Corbet, *A thorough introduction to ebpf*, https://lwn.net/Articles/740157/, Accessed: 2025-08-08, 2018.

[33] X. Wu, Y. Feng, T. Huang, X. Lu, S. Lin, L. Xie, S. Zhao, and Q. Cao, "Vep: A two-stage verification toolchain for full ebpf programmability", in *NSDI*, 2025.

[34] J. Mao, H. Ding, J. Zhai, and S. Ma, "Merlin: Multi-tier optimization of ebpf code for performance and compactness", in *ASPLOS*, 2024.

[35] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes", in *Linux Symposium*, vol. 6, 2006, p. 5.

[36] *Kprobes*, https://sourceware.org/systemtap/kprobes/, Notes that Kprobes entered mainline in Nov. 2004 (Linux 2.6.9-rc2). Accessed 2025-08-16.

[37] J. Corbet, *An introduction to kprobes*, https://lwn.net/Articles/132196/, Accessed 2025-08-16, Apr. 2005.

[38] V. Shymanskyy and the Wasm3 contributors, *Wasm3: A fast webassembly interpreter and the most universal wasm runtime*, https://github.com/wasm3/wasm3, MIT License. Accessed: 2025-08-11, 2025.

[39] F. Abdelmonem, *Kernel-wasm-runtime: A prototype framework for kernel extensibility using webassembly*, https://github.com/Faisal-Saleh/kernel-wasm-runtime, Accessed: 2025-08-15, 2025.

[40] D. Baek, J. Getz, Y. Sim, D. Lehmann, B. L. Titzer, S. Ryu, and M. Pradel, "Wasm-r3: Record-reduce-replay for realistic and standalone webassembly benchmarks", in *OOPSLA*, 2024. DOI: 10.1145/3689787. [Online]. Available: https://dl.acm.org/doi/10.1145/3689787.

[41] E. Gilbert, M. Schneider, Z. An, S. Thalanki, W. Bowman, A. Bai, B. L. Titzer, and H. Miller, "Debugging webassembly? put some whamm on it!", *arXiv preprint arXiv:2504.20192*, 2025.

[42] B. Alliance, *Wasmtime 28.0: Optimizing at compile-time, booleans in the cranelift dsl, and more*, Accessed 2025-08-14, 2025. [Online]. Available: https://bytecodealliance.org/articles/wasmtime-28.0.

[43] Eunomia-bpf, *Wasm-bpf: Webassembly ebpf library, toolchain and runtime*, Accessed 2025-08-14, 2024. [Online]. Available: https://github.com/eunomia-bpf/wasm-bpf.

[44] WasmEdge, *Build with ebpf plug-in*, Accessed 2025-08-14, 2024. [Online]. Available: https://wasmedge.org/docs/contribute/source/plugin/ebpf/.