Proc. of International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA 2026)
5-7 February 2026, Boracay-Philippines

Fast Per-Request Isolation in TinyKVM with Unmodified Applications

1st Alf-André Walla
University of Oslo (UiO)
Norway
fwsgonzo@hotmail.com

2nd Laurence D. Rowe
Independent Consultant
United States
laurencerowe@gmail.com

3rd Paal E. Engelstad
University of Oslo (UiO)
Norway
paal.engelstad@its.uio.no

***Abstract*—Modern edge computing, serverless platforms, and programmable infrastructure demand efficient per-request isolation for executing untrusted code. Applications range from serverless functions and microservices to programmable CDN configurations where JavaScript has replaced domain-specific languages, requiring isolation between requests to prevent state leakage and avoid garbage collection overhead. Current approaches fail to meet these requirements adequately. V8 isolates [1], the industry standard for JavaScript sandboxing, lack any reset mechanism as they must be destroyed and recreated for each request, with instantiation times of 1-25ms that quickly become prohibitive under load. Production deployments like Cloudflare Workers therefore maintain long-lived isolates across multiple requests, sacrificing true per-request isolation for performance. WebAssembly [2] runtimes like *wasmtime* [3] offer faster instantiation but prohibit JIT compilation within guest code, severely limiting performance for computationally intensive applications and restricting framework availability.**

This paper introduces TinyKVM-based per-request isolation that combines microsecond reset times with native performance and full guest-side JIT support. TinyKVM [4] originally required custom system calls in guest applications to control request boundaries, creating a maintenance burden. Our enhanced implementation introduces automatic file descriptor tracking at the hypervisor level. This enables operation with unmodified executables while maintaining full isolation guarantees. Combined with a new reset mechanism that preserves page tables while completely restoring guest state, and leveraging JIT warmup preservation across request boundaries, the system enables practical per-request isolation for diverse workloads from edge functions to unmodified server programs like Deno. The architecture supports enterprise defense-in-depth requirements through hardware virtualization isolation combined with memory-safe runtime environments, representing the first practical solution to achieve native JIT performance, microsecond resets, and enterprise-grade security boundaries simultaneously.

***Keywords*—edge computing, sandboxing, per-request isolation, virtualization**

I. Introduction

The proliferation of cloud computing and edge services has fundamentally transformed how web applications are deployed and secured. Modern serverless platforms and content delivery networks (CDNs) routinely execute code within shared infrastructure, creating security challenges.

The core security vulnerability in shared execution environments stems from state persistence between requests. When multiple tenants share the same process or runtime instance, residual data from previous executions can leak across tenant or request boundaries through persistent runtime state.

Wiping state between requests represents an alternative approach, but proving correctness remains challenging for complex runtime environments. Modern JavaScript engines, for example, maintain intricate internal state including JIT-compiled code, heap objects, and runtime caches that are difficult to fully reset without performance penalties. Even memory-safe languages like Rust cannot eliminate all forms of information leakage through timing channels, cache behavior, or speculative execution artifacts.

Per-request isolation addresses these challenges by creating a fresh, isolated execution environment for each incoming request, regardless of tenant. This approach provides deterministic security guarantees: temporal attacks between requests become impossible, memory disclosures cannot persist across request boundaries, and resource exhaustion attacks are naturally contained. For CDNs, edge computing platforms, and multi-tenant SaaS providers, per-request isolation represents the gold standard for security while enabling the performance characteristics that modern applications demand.

Modern web services requiring per-request isolation face a fundamental performance bottleneck in existing sandboxing technologies. The industry standard approach uses V8 isolates to provide JavaScript sandboxing without request-level reset capabilities. Production deployments universally require defense-in-depth, necessitating that V8 isolates and WebAssembly runtimes operate within additional containment layers. Firecracker micro-VMs represent the current gold standard for this outer isolation boundary, but alternative approaches include process jailing, seccomp filtering, or namespace isolation. However, regardless of the containment strategy chosen, the fundamental performance bottleneck remains unchanged: V8 isolates are too expensive to create and destroy per request, while WebAssembly's prohibition of guest JIT compilation creates a performance ceiling for dynamic workloads.

979-8-3315-7191-7/26/$31.00

One security challenge with same-address-space solutions like V8 isolates and WebAssembly runtimes is their shared memory space with the host process. Despite sophisticated isolation mechanisms, both technologies remain vulnerable to speculative execution attacks, memory corruption exploits, and side-channel attacks [5] that can breach sandbox boundaries. Industry best practices therefore mandate defense-in-depth approaches where these emulators and runtimes are themselves contained within additional isolation layers. Typically process boundaries or lightweight VMs like Firecracker with a jailer on top.

V8's isolate reset mechanism becomes prohibitively slow under high request loads regardless of the underlying virtualization layer. Isolates also require custom modification of the outer application to support resetting. Process forking with V8 in order to achieve resets is even slower, and explicitly not supported. Isolates also must be instantiated in individual processes for defense-in-depth. WebAssembly runtimes like *wasmtime* solve the reset performance problem but impose a constraint: WebAssembly's security model prohibits JIT compilation within guest code, severely limiting performance for computationally intensive applications that rely on dynamic compilation. WebAssembly emulators should also be jailed in a process for defense-in-depth similar to what Firecracker already does.

The security imperative for robust sandboxing of same-address-space emulators has been reinforced by recent vulnerability disclosures. Speculative execution vulnerabilities like Spectre and Meltdown particularly impact solutions sharing address space with the host, while memory corruption bugs in JavaScript engines and WebAssembly runtimes continue to surface regularly. Even with sophisticated mitigations like Control Flow Integrity (CFI) and memory tagging, the attack surface exposed by complex JIT compilers and runtime systems necessitates additional isolation boundaries that can contain successful exploits.

This creates a problematic gap for applications requiring both practical per-request isolation and JIT performance while meeting enterprise security requirements. V8 isolates, whether deployed in dedicated processes, containers, or Firecracker VMs, does not practically support per-request isolation. WebAssembly solutions, despite requiring similar containment strategies, sacrifice the computational performance that JIT compilation provides for complex applications like modern JavaScript runtimes. Critically, no existing solution provides per-request isolation that combines hardware-level security boundaries with native JIT performance and sub-millisecond reset capabilities. The industry has been forced to choose between security (hardware isolation), performance (native JIT), or operational efficiency (fast resets), but never all three simultaneously.

II. Background

A. Defense-in-Depth Architectures

Enterprise security requirements mandate multiple isolation layers when executing untrusted code, a principle known as defense-in-depth [6]. This approach recognizes that no single security boundary is infallible. Successful attacks against one layer should be contained by additional isolation mechanisms. M.R. Stytz et al. also points out that "this conceptualization for interlocking defense would not relieve the development team of the need to maintain best practices for secure software development and software development in general." Modern cloud and edge computing environments typically implement three distinct isolation boundaries, each providing different security guarantees and performance characteristics.

The outermost layer typically consists of hardware-enforced virtualization boundaries, such as those provided by KVM, Xen, or lightweight solutions like Firecracker. These hypervisor-level boundaries leverage hardware memory management units and processor privilege levels to create strong isolation between workloads, protecting against both architectural attacks (buffer overflows, code injection) and micro-architectural attacks (cache timing, speculative execution side-channels).

The middle layer often employs process-level isolation or container technologies that leverage operating system security mechanisms. Solutions like gVisor implement user-space kernels to reduce the attack surface exposed to guest applications, while container runtimes provide namespace and cgroup isolation. These approaches offer a balance between isolation strength and performance overhead, though they remain vulnerable to kernel exploits and may not provide sufficient protection against sophisticated attacks.

The innermost layer consists of application-level sandboxing technologies such as V8 isolates, WebAssembly runtimes, or language-specific isolation mechanisms. These solutions provide fine-grained control over code execution and resource access but operate within the same address space as the host process, making them vulnerable to memory corruption exploits, speculative execution attacks, and implementation bugs in the sandbox itself.

Recent vulnerability disclosures have reinforced the importance of defense-in-depth for same-address-space emulators. Speculative execution vulnerabilities like Spectre and Meltdown particularly impact solutions sharing address space with the host, while implementation bugs in complex systems like JavaScript engines continue to surface regularly. Even with sophisticated mitigations like Control Flow Integrity (CFI), Intel CET, and ARM Pointer Authentication, the attack surface exposed by JIT compilers and dynamic runtimes necessitates containment within hardware-enforced boundaries.

B. Per-Request Isolation

Joseph M. Hellerstein et. al [7] points out in his paper about FaaS and serverless requirements that "allowing code to move fluidly toward shared data storage is po- tentially tricky: it exacerbates security management challenges related to multitenancy and the potential for rogue code to gather signals across customers".

Modern cloud and edge computing environments often used a heavy-handed approach when handling untrusted code

execution. These traditional deployment models isolated applications at the VM or container level. Emerging serverless and edge computing paradigms demand finer-grained isolation - ideally isolating each individual request to prevent both cross-request and cross-tenant attacks. Per-request isolation represents a departure from conventional isolation models. Rather than maintaining long-lived sandboxes that handle multiple requests, per-request isolation creates a fresh, isolated execution environment for each incoming request, destroying it upon completion. This approach provides strong security guarantees: temporal attacks between requests become impossible, memory disclosures cannot leak across request boundaries, and resource exhaustion attacks are naturally bounded.

1) VM-Based Isolation: Traditional virtualization technologies like KVM and Xen provide strong isolation through hardware-enforced boundaries. Firecracker [8], considered part of lightweight VMs can achieve sub-125ms cold starts and minimal memory overhead. However, Firecracker and similar solutions are designed for function-level or microVM-level isolation, where a single VM instance handles multiple requests over its lifetime. The overhead of creating and destroying a full VM for each request - including kernel boot, init system, and runtime initialization - makes per-request VM isolation impractical for latency-sensitive applications.

2) Process-Based Isolation: Operating system processes offer a lighter-weight isolation boundary, leveraging hardware memory protection and kernel-enforced separation. However, process creation overhead, while lower than VM instantiation, still imposes significant costs for per-request isolation. More critically, processes share kernel state and are vulnerable to kernel exploits, providing weaker isolation guarantees than hardware-based virtualization.

3) WebAssembly-Based Sandboxing: WebAssembly [2] has emerged as a promising technology for lightweight sandboxing, offering near-native performance with isolation guarantees within the same address space as the host process. WebAssembly runtimes like *wasmtime* [3] can instantiate new sandboxes in microseconds, making per-request isolation feasible. The isolation model relies on Software Fault Isolation (SFI) techniques, with compile-time bounds checking and capability-based security.

However, WebAssembly's isolation model comes with limitations:

- WebAssembly executes in a constrained environment with a linear memory model and restricted syscall interface (WASI). Complex applications requiring full POSIX compatibility, threading, or direct hardware access must be substantially modified or cannot run at all.
- WebAssembly does not support JIT-compilation in guest programs. Dynamic languages like JavaScript must embed their interpreters into WebAssembly modules [9], losing the performance benefits of JIT compilation that modern JavaScript engines like V8 [1] provide. This is particularly problematic for long-running or compute-intensive JavaScript applications where JIT optimization can provide 10-100x speedups on hot code paths.
- Applications must be compiled to WebAssembly, requiring toolchain support and often substantial code modifications. Native libraries with inline assembly, architecture-specific optimizations, or complex memory management patterns may not be portable to WebAssembly without significant engineering effort, and some are not portable at all, eg. CPU-heavy machine learning inference and cryptographic workloads. Very few, if any, programs compile directly to WebAssembly without modification and custom compilation and linker options specific to WebAssembly.

Note that WebAssembly requires specially crafted programs in order to be able to practically use per-request isolation (see: wasi-http [10]).

C. The Memory Safety Imperative

The push toward per-request isolation is partly driven by the persistent challenge of memory safety vulnerabilities. Recent industry analyses [11], [12] indicate that approximately 70% of severe security vulnerabilities stem from memory safety issues. Per-request isolation provides a strong defense against these vulnerabilities by ensuring that even successful exploits cannot persist across request boundaries or affect other tenants.

D. Requirements for Practical Per-Request Isolation

An ideal per-request isolation system must balance several competing requirements:

- Warmup: Per-request isolation is sometimes hosted with garbage collected languages that also have a JIT-compiler. JITs require warmup to perform predictably and fast, and in a way that avoids or prevents garbage collection during request handling.
- Minimal Memory Overhead: The memory footprint per isolated instance must be small enough to allow high density on edge nodes with limited resources.
- Application Compatibility: Support for unmodified applications, including JIT-compiled languages, full syscall interfaces, and native libraries.
- Strong Isolation Guarantees: Hardware-backed isolation comparable to VM-level security, protecting against both architectural and micro-architectural attacks.
- Efficient Resource Reclamation: Complete cleanup of resources after each request, preventing resource leaks and ensuring consistent performance.
- Failure Recovery: Crashes or execution timeouts should gracefully reset the VM such that future requests can succeed, minimizing error rates.

Modern enterprise deployments cannot rely solely on software-based isolation for untrusted code execution. Regulatory compliance frameworks and security best practices mandate that same-address-space emulators operate within hardware-backed isolation boundaries, creating a containment hierarchy where hardware virtualization provides the ultimate security boundary, process or VM isolation provides the operational boundary, and application-level sandboxes (V8 isolates or WebAssembly) provide the request-level boundary.

E. Our Approach: TinyKVM

TinyKVM [4] represents a novel approach to per-request isolation that leverages KVM's hardware virtualization capabilities while avoiding the overhead of traditional VMs. Our implementation demonstrates the practical viability of enterprise-grade defense-in-depth by combining hardware isolation with memory-safe runtime environments. Specifically, we deploy Deno, a runtime written in Rust that embeds v8 for JavaScript execution, within TinyKVM's hypervisor boundary. This creates a three-layer security model: hardware virtualization isolation (TinyKVM), memory-safe runtime isolation (Rust), and application-level sandboxing (V8 isolates).

Our architecture preserves JIT compilation benefits by performing warmup self-requests at boot time. Rather than discarding expensive JIT optimizations at request boundaries, the system maintains compiled code across resets while ensuring complete state isolation. This enables TinyKVM to bridge the gap between the strong isolation of VMs and the low overhead of WebAssembly sandboxes, while providing the computational performance that enterprise applications require.

Unlike Firecracker, which optimizes for long-lived microVMs handling multiple requests, TinyKVM specifically targets the per-request model with defense-in-depth requirements. Unlike WebAssembly-based solutions, TinyKVM preserves full application compatibility and JIT optimization capabilities within a hardware-isolated boundary. This enables a new class of secure, high-performance edge computing applications with fully automatic per-request isolation that were previously impractical with existing technologies, particularly for workloads requiring both computational performance and enterprise security guarantees.

III. Related Work

A. gVisor: User-Space Kernel Emulation

gVisor represents Google's approach to container sandboxing through user-space kernel emulation [13]. Unlike traditional container runtimes that share the host kernel, gVisor interposes a user-space kernel (the Sentry) between the application and host, intercepting and re-implementing system calls. This architecture provides defense-in-depth by minimizing the host kernel attack surface exposed to untrusted workloads.

The progenitor implementation to gVisor, primarily written in Go, demonstrated that high-level language implementations could achieve acceptable performance for sandboxing, with the original 2018 research showing only 5-15% overhead compared to native C/C++ implementations as shown by Cutler et. al [14]. However, this architectural choice introduces trade-offs. It implements over 200 system calls in userspace, each representing potential attack surface, while TinyKVM's minimalist approach implements approximately 50 system calls, a reduced attack surface. Specifically, TinyKVM focuses on the system calls needed to host a modern epoll-based server, especially avoiding dangerous things like `inotify` and `io_uring`. In fairness, system calls in gVisor has been extensively fuzzed by G-Fuzz, a directed fuzzing framework for gVisor [15]. This framework claims it is general enough to be used on other OS kernels, and might also be a candidate for TinyKVM. A study on the performace of gVisor and other container run-times [16] shows it has the best isolation, but lower performance.

gVisor supports two isolation modes: ptrace and KVM. The ptrace mode operates entirely in user-space but suffers from performance overhead due to context switching costs. The KVM mode leverages hardware virtualization for stronger isolation and better performance, particularly for CPU-bound workloads. This KVM mode more closely aligns with TinyKVM's approach, though with different design goals: gVisor aims for container compatibility while TinyKVM optimizes for fast per-request isolation.

gVisor's architecture is optimized for long-running container workloads rather than per-request isolation. The overhead of Sentry initialization and state management makes it unsuitable for a full state reset after each request. Additionally, the Go runtime's garbage collection introduces unpredictable latency spikes that can be problematic for latency-sensitive edge computing applications. While gVisor excels at providing compatible container sandboxing with reduced kernel exposure, it does not address the specific requirements of high-performance per-request isolation that TinyKVM targets.

Other container run-times such as RunC and Kata Containers also cannot realistically support per-request isolation.

B. JavaScript Sandboxing and Isolation

V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others [1]. V8 isolates represent the current industry standard for JavaScript sandboxing and per-request isolation in web services [17]. An isolate provides a sandboxed execution environment with its own heap and context, enabling relatively secure execution of untrusted JavaScript code. The isolation mechanism allows multiple isolates to coexist within a single V8 instance while maintaining memory separation between contexts. However, V8 isolates face significant performance challenges in reset operations. When an isolate completes execution, restoring it to a pristine state requires extensive cleanup of heap objects, compiled code caches, and execution contexts. In practice this is done by creating a new isolate from a heap snapshot. This method becomes particularly pronounced under high request loads, where the cost of isolate re-initialization can dominate total execution time. Additionally, V8's use of W+X memory segments for JIT-compiled code, while mitigated through various security measures, necessitates additional isolation layers in high-security deployments. In short, there is never just V8 isolates and there is always another layer that adds some overhead. For this reason, you cannot find any public cloud vendor that supports per-request isolation with V8 isolates. For example, Cloudflare Workers use V8 isolates within their custom *workerd* runtime. Each isolate handles multiple requests from the same tenant until it times out due to inactivity, there

is no isolation between individual requests. This architectural choice prioritizes performance and cost efficiency over per-request security boundaries, making it unsuitable for scenarios where requests must be fully isolated from each other.

C. WebAssembly

WebAssembly runtimes like *wasmtime* have emerged as alternatives that address the reset performance limitations of V8 isolates [3] [18]. WebAssembly's design enables efficient instantiation and teardown of execution environments, achieving reset performance that significantly outpaces V8 isolates. The WebAssembly security model provides some isolation guarantees through its sandboxed execution environment and capability-based security approach. However, WebAssembly still uses the same address space as the host and has to be meticulously tested for side channels. Further, the security model explicitly prohibits JIT compilation within guest code. This restriction negatively impacts performance for applications that rely on dynamic recompilation, such as modern JavaScript engines running complex applications and other dynamic languages (LuaJIT, Java, Ruby, Python 3.13+). Consequently, while WebAssembly runtimes excel at fast resets, they cannot match the computational performance that JIT-enabled environments provide for dynamic workloads. There are also madvise-related performance issues for larger working sets, where inter-processor-interrupts (IPIs) caused by mapping changes incur performance loss. Horizontal scaling of *wasmtime* is problematic and starts to take seconds when there are thousands of instances. [19] At high instance counts, the memory map operations experience some serialization due to kernel page table locks.

While WebAssembly can be fast for some computations, it lacks many native instructions required for heavy processing, such as the AVX-512 family of instructions, and especially cryptographic instructions. A common marketing term used is "near-native", however, in practice it is somewhere between 1/3 to 1/2 of native at best. Its low latency when exiting to the host makes it a very good candidate for configuration-like programming, such as serverless lambdas, where the real work happens elsewhere. However, if you need to verify a signature, a host function is needed to compensate for the potential 80x performance loss. [20]

WebAssembly emulators often support fast in-and-out function calls into the VM, which means that for small functions there will be a threshold where a WebAssembly (or other CPU emulator, like libriscv [21]) will be faster than TinyKVM. In practice, as complexity increases, the head start from the lower call overhead will start to diminish and eventually cross a point where native performance wins out again. In a simplified calculation where TinyKVM has a 1µs function call overhead and *wasmtime* JIT runs at half-native, the run-time threshold would be 3µs.

D. Micro-VM Approaches

Firecracker represents a significant advancement in lightweight virtualization for serverless and microservice environments [8]. Designed specifically for multitenant cloud environments, Firecracker provides micro-VMs that boot in milliseconds while maintaining full hardware virtualization isolation. The system achieves minimal performance overhead during runtime execution through paravirtual devices and a minimal guest kernel (relative to distro kernels). However, Firecracker provides full-system virtualization and does not provide application-level request isolation capabilities. In practice, Firecracker serves as an additional security layer beneath application-level sandboxing solutions like V8 isolates, creating a defense-in-depth strategy where Firecracker provides the outer virtualization boundary while V8 isolates handle request lifecycle management. This layered approach adds some performance overhead due to the overheads related to virtualized I/O, and also does not address the reset performance limitations inherent in V8 isolates. Firecracker does, however, have the lowest attack surface when using no esoteric paravirtual devices.

E. TinyKVM and Userspace Emulation

TinyKVM introduced a novel approach to virtualization focused on single-process isolation rather than full-system virtualization [4]. Unlike traditional hypervisors that virtualize entire operating systems, TinyKVM sandboxes individual processes while handling system calls in the host environment. This architecture enables fine-grained control over process execution and state management while maintaining native performance for computational workloads. The original TinyKVM implementation demonstrated promising performance characteristics, but could only run statically linked programs, and larger programs had to be specially written to avoid hitting unimplemented system calls. In addition, applications needed to incorporate custom system calls to signal request boundaries, enabling the hypervisor to provide isolation guarantees and the best performance and latency. Although this approach proved effective for controlled environments, the requirement for custom builds created deployment friction and limited practical adoption, particularly for complex applications like modern JavaScript runtimes that benefit from standard distribution channels and existing tooling ecosystems. It should be noted that specially crafted applications will always have the best latency and time-to-first-byte (TTFB), however, only by a few microseconds. In the original paper, it was shown that it was possible to run heavy compute workloads with only a single KVM exit at the very end to signal completion, and this is still possible but impractical.

IV. TinyKVM Architecture and Enhancements

A. File Descriptor Tracking

TinyKVM will generally track file descriptors that are created by the guest, managing them both in the main VM and in forked VMs separately. Any rule can be applied at fork-time, such as to close files and sockets, or make all file descriptors read-only for VM forks.

Epoll systems and file descriptors that meaningfully contribute to the epoll systems are specially tracked such that they

can be reconstructed on each fork, per request. This makes it possible to faithfully recreate the original state of the main VM. As an example, the Rust-based Deno runtime uses several epoll systems and eventfd's to manage its multiple event loops. TinyKVM will track all of it and faithfully recreate it on each reset. This event loop is created from scratch on each reset avoiding race conditions and other problems. Old file descriptors are always closed.

Security-wise the file-descriptor implementation should be very robust as only very few system calls are actually implemented, and those that are are limited in scope: Very few flags and modes of operation are allowed. Practically, the minimum required to operate applications like Deno, Node.js, Python WSGI, and other runtimes/frameworks like them. Permissions have to be explicitly allow-listed. TinyKVM is by default in a fully closed state.

While it is possible to experiment with opening up 8 connections to a Redis server and then sharing those connections with 8 VM instances so that each one can talk to Redis without reconnecting during resets, the possibility of losing synchronization with the remote or disconnecting due to an error is always there, and so our experiments are only done with new connections on each reset.

B. Optimized Reset Mechanism

In the original TinyKVM implementation, the page tables were reconstructed on each reset, which also had the effect of releasing memory down to the bare minimum after each request. For large and complex request workloads this became expensive and so a new method that did not make pagetable changes at all was developed. Instead, this method resets each page to the original state in the main VM by copying memory upfront, which was happening lazily/on-demand previously. The new method turns out to be much faster as requests generally touch much of the same memory, and it does not require a pagetable reload shooting down TLB entries and lots of inter-processor-interrupts (IPIs). As a drawback the pagetables can grow over time on each VM fork.

The old method is still kept intact as an option called hard reset. If the forked VM uses too much memory or fails spectacularly, a hard reset can be a big hammer that reconstructs the VM while reducing the memory footprint. Combining both reset methods is a way to keep the system stable over long periods by keeping memory usage in check and reacting to request memory usage spikes. Our experimentation has shown that long-running services with stable memory usage generally do not need to be hard reset at all. Finally, it is important to note that both methods have the same effect: A full reset is observed in the guest program. All state including executable code, thread-local storage, multi-threading state and everything else is restored back to its original state.

C. JIT Preservation Strategy

In server mode, TinyKVM will wait for the guest to listen on a port using either blocking *accept*, *poll* or *epoll*. After listening, the guest is considered ready for requests. Since the server has full knowledge of all file descriptors, it can connect to the guest's listener and send enough HTTP requests to activate JIT compilation. Modern dynamic language runtimes usually have several tiered JITs, and activating the highest tier will require the user to experiment to find out where the threshold is for a particular program.

After enough requests have been sent to generate JIT-compiled code segments for the fast/happy path, TinyKVM will step in and prepare the VM guest for forking and per-request isolation. It will go through each user-writable page and tag it as copy-on-write. After this, it is possible to fork out tiny VM instances from the main VM, as a way to horizontally scale up compute.

Edd Barrett et al. have shown that many microbenchmarks towards JIT runtimes fail to reach peak performance [22]. JITs are typically unpredictable and may need a prolonged warm-up phase in order to guarantee the best performance. The v8 JIT compiler can be made predictable with the `--predictable` argument at startup, which eliminates performance inconsistencies. We observed a warmed-up JIT obtained predictable performance within just a few requests with this flag enabled.

In our benchmark suite, we generally warm the main VM with around 1000 requests, usually adding less than 1ms to the startup time. If too many requests are used there's a chance that the guest will be very close to invoking garbage collection during a request (especially a complex one), which can be a hard-to-debug pessimization in production. Warming up the guest too much will cause a GC invocation, which will execute cold paths in the guest, add to startup time and has never been seen to improve performance.

TinyKVM preserves compiled machine code across resets while completely restoring all mutable state including heap contents, stack state, and register values. This enables applications to benefit from JIT warmup without state leakage between requests.

D. Limitations

TinyKVM's approach has several constraints that affect its applicability:

- KVM API and initialization adds latency compared to process-based sandboxing for short-lived functions (future work).
- Each instance requires additional memory for page tables and hypervisor structures, although the program and all dependencies are mapped from file, enabling safe read-only memory sharing with the host and other guests.
- File descriptor tracking complexity as well as extra file descriptors per instance places a limit on maximum concurrent instances.
- Scalability comparisons is future work. We know that WebAssembly (and other CPU-) emulators struggle with scalability due to kernel pagetable locking and IPIs with concurrent madvise. However, for KVM VMs there is an upper limit on the total number. As an example, unmodified programs often use multiple file descriptors

for event loops (usually at least two), which doesn't scale well with thousands of VMs.
- Overhead from Spectre/Meltdown mitigations is not separately quantified, although they especially apply to solutions within the same address space as the host (e.g. V8, WebAssembly emulators and other CPU emulators providing sandboxing).
- While TinyKVM provides strong isolation through hardware virtualization with separate PCIDs that prevent timing attacks, the system remains vulnerable to hypervisor escape exploits that could compromise the host system. However, the defense-in-depth architecture requires successful attacks to breach multiple independent security boundaries simultaneously.
- While supporting unmodified binaries, applications with exotic syscall requirements may need adaptation.
- TinyKVM intentionally lacks pre-emption support, which means multi-threading is cooperative. This can sometimes be challenging if a runtime has background threads that perform blocking operations. However, all big JavaScript runtimes work in TinyKVM, so there is currently no known unsupported languages or runtimes.
- Nested hardware virtualization is only up to 80% of native, and sometimes worse for older CPUs. We recommend running TinyKVM on bare metal.

V. Results

TABLE I
RUST MINIMAL HTTP SERVER (ASYNC EPOLL)

name	average	p50	p90	p99
native (reuse connection)	6 µs	6 µs	6 µs	8 µs
native	16 µs	16 µs	17 µs	20 µs
tinykvm t=1 (reuse connection)	13 µs	12 µs	14 µs	16 µs
tinykvm t=1	25 µs	26 µs	27 µs	32 µs
tinykvm ephemeral t=1	28 µs	27 µs	28 µs	36 µs
tinykvm ephemeral t=2	30 µs	28 µs	33 µs	45 µs
tinykvm ephemeral t=4	32 µs	31 µs	35 µs	52 µs
tinykvm ephemeral t=2 no-tail	25 µs	25 µs	28 µs	32 µs

TABLE II
RUST MINIMAL HTTP SERVER (SYNC BLOCKING)

name	average	p50	p90	p99
native (reuse connection)	5 µs	5 µs	6 µs	6 µs
native	10 µs	10 µs	10 µs	13 µs
tinykvm t=1 (reuse connection)	8 µs	8 µs	10 µs	14 µs
tinykvm t=1	13 µs	12 µs	13 µs	19 µs
tinykvm ephemeral t=1	13 µs	12 µs	13 µs	19 µs
tinykvm ephemeral t=2	15 µs	13 µs	20 µs	26 µs
tinykvm ephemeral t=4	18 µs	16 µs	24 µs	30 µs
tinykvm ephemeral t=2 no-tail	13 µs	12 µs	17 µs	19 µs

TABLE III
DENO HELLOWORLD

name	average	p50	p90	p99
native (reuse conn.)	14 µs	14 µs	15 µs	16 µs
native	17 µs	15 µs	23 µs	33 µs
tinykvm t=1 (reuse conn.)	14 µs	13 µs	14 µs	20 µs
tinykvm t=1	26 µs	25 µs	26 µs	42 µs
tinykvm ephemeral t=1	52 µs	50 µs	52 µs	89 µs
tinykvm ephemeral t=2	53 µs	51 µs	55 µs	80 µs
tinykvm ephemeral t=4	57 µs	53 µs	60 µs	98 µs
tinykvm eph t=2 no-tail	38 µs	37 µs	43 µs	56 µs

TABLE IV
DENO REACT PAGE RENDERING

name	average	p50	p90	p99
native (reuse conn.)	640 µs	611 µs	680 µs	800 µs
native	640 µs	611 µs	670 µs	926 µs
tinykvm t=1 (reuse conn.)	618 µs	594 µs	657 µs	776 µs
tinykvm t=1	640 µs	614 µs	677 µs	858 µs
tinykvm ephemeral t=1	708 µs	689 µs	699 µs	809 µs
tinykvm ephemeral t=2	697 µs	679 µs	703 µs	809 µs
tinykvm ephemeral t=4	716 µs	691 µs	707 µs	820 µs
tinykvm eph t=2 no-tail	624 µs	620 µs	637 µs	740 µs

TABLE V
WASMTIME SERVE RUST HELLOWORLD

name	average	p50	p90	p99
instance reuse	11 µs	10 µs	11 µs	12 µs
ephemeral	23 µs	22 µs	23 µs	27 µs

TABLE VI
WASMTIME SERVE JS HELLOWORLD

name	average	p50	p90	p99
instance reuse	84 µs	80 µs	86 µs	106 µs
ephemeral	314 µs	316 µs	324 µs	405 µs

TABLE VII
WASMTIME SERVE REACT PAGE RENDERING

name	average	p50	p90	p99
instance reuse	10569 µs	10477 µs	10834 µs	11504 µs
ephemeral	13197 µs	13167 µs	13341 µs	13636 µs

TABLE VIII
BUN PROCESS FORKING HELLOWORLD

name	average	p50	p90	p99
bun.serve (reuse conn)	11 µs	11 µs	11 µs	13 µs
process forking	1756 µs	1727 µs	1845 µs	2478 µs

Reset mechanisms are not free, as there is a lot of work to do to fully reset a VM back to a previous state. The React benchmark average latency was 685µs, comprising 10µs connection creation, 15µs sandbox execution, 48µs reset, and 612µs application processing. See table IV. Performance is more consistent than native because VM resets avoid garbage

TABLE IX
BUN PROCESS FORKING REACT PAGE RENDERING

name	average	p50	p90	p99
bun.serve (reuse conn)	724 µs	596 µs	790 µs	2662 µs
process forking	8269 µs	8328 µs	8718 µs	9437 µs

TABLE X
NODE ISOLATES HELLO WORLD JS

name	average	p50	p90	p99
main runtime	15 µs	15 µs	15 µs	27 µs
isolate reused	20 µs	18 µs	21 µs	45 µs
isolate ephemeral	749 µs	739 µs	780 µs	879 µs

TABLE XI
NODE ISOLATES REACT PAGE RENDERING

name	average	p50	p90	p99
main runtime	562 µs	528 µs	638 µs	763 µs
isolate reused	825 µs	790 µs	899 µs	1601 µs
isolate ephemeral	22777 µs	22764 µs	22947 µs	23015 µs

TABLE XII
MICROBENCHMARK OF A MINIMAL C APPLICATION TO ISOLATE RAW HYPERVISOR RESET OVERHEAD.

name	average	p50	p90	p99
Non-ephemeral	8.54 µs	8.00 µs	9.00 µs	10.00 µs
Ephemeral	9.00 µs	9.00 µs	10.00 µs	11.00 µs

collection and JIT spikes, as per Fig. 1. In the rightmost column the reset mechanism is turned into tail latency (by running it in another thread), lowering time-to-first-byte (TTFB) to 624µs for p50, 646µs for p90 and 753µs for p99. See Table IV. Reset time is calculated as p50 ephemeral t=1 minus ephemeral t=1 with reset as tail latency, which is 672µs - 624µs = 48µs. There is also a benchmark for Deno Hello World JS in table III, and a Rust hello world benchmark in table II (blocking) and I (non-blocking). Turning the reset mechanism into tail latency does not lower overall system CPU usage. On an unsaturated system it is conceivable that the TinyKVM solution will deliver responses more consistently and overall earlier than native for garbage collected languages like JavaScript.

TinyKVM is unbeatable in per-request reset time as it does not incur any pagetable changes at all, avoiding any IPIs and potential Linux kernel page table code-paths and atomic operations that cause coherency protocol chatter. As per Fig. 3 we can see that even the smallest, simplest HTTP server is faster in TinyKVM than in *wasmtime*. As stated in the original TinyKVM paper, the entire TinyKVM guest address space is accessible in the host allowing for zero-copy I/O to the host in various system calls [4].

To isolate and quantify the absolute minimum overhead of our optimized reset mechanism, we conducted an additional

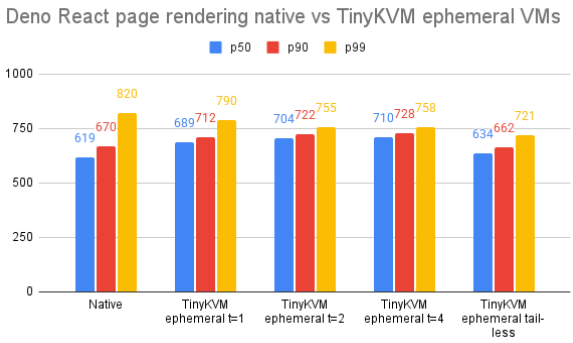

Fig. 1. Deno native vs TinyKVM ephemeral VMs

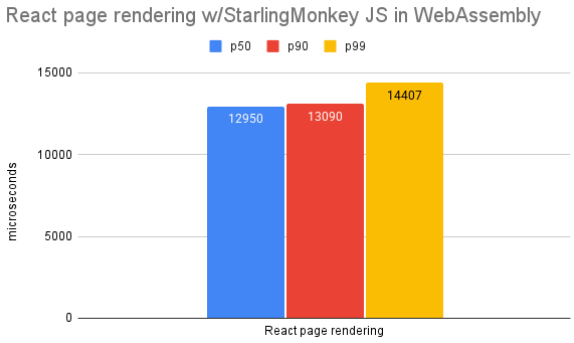

Fig. 2. React page rendering in *wasmtime* w/StarlingMonkey JS

micro-benchmark (Table XII). For this test, we used a minimal 64-bit ELF binary written in C, which responds to requests via a simple function call from the hypervisor, similar to a WebAssembly guest function. This approach eliminates the overhead from complex runtimes, JITs, and extensive file descriptor management, revealing the raw performance (and best case scenario) of the VM state reset.

When running this minimal ELF in non-ephemeral mode (one persistent VM), we measured a median latency of 8.54µs. In ephemeral mode, where a full state reset was performed for every request, the median latency was 9.00µs. The difference

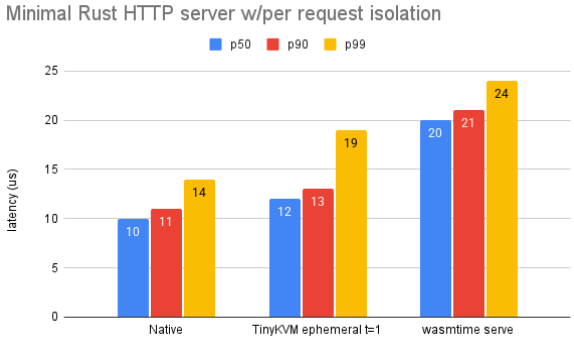

Fig. 3. Minimal Rust HTTP server benchmark

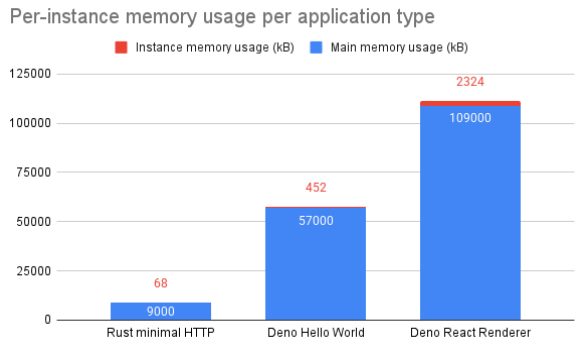

Fig. 4. Per-instance memory usage

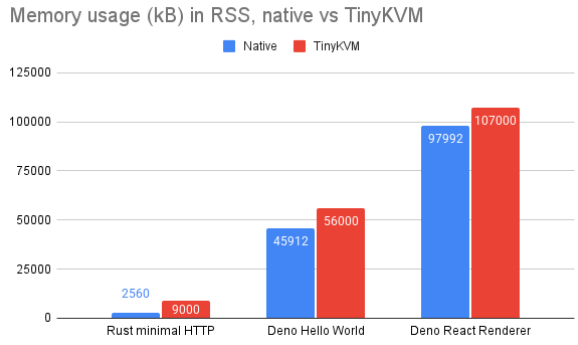

Fig. 5. Total resident memory usage at boot-time

reveals a raw reset overhead of just 0.46µs. This overhead demonstrates the lower bound of our reset mechanism.

Per-instance memory usage is low in TinyKVM. As per Fig. 4, we see that an entire React page rendering application requires only 2324 KiB per additional request handling instance. For a minimal Rust HTTP server it was 68 KiB and for a Deno Hello World it was 452 KiB. The total RSS usage after startup (Fig. 5) shows that loading large programs does not add much memory overhead. Specifically, a minimal HTTP server added 6.5 MiB (3.5x) to RSS, a Deno Hello World added 10.1 MiB (+22%), and a fully featured Deno react page rendering application added 9 MiB (+9%) compared to running the same programs natively. This boot-time memory overhead is not related to scaling, and trying to lower it further only makes sense if it can be shown to substantially reduce the cold start time. Most of the additional memory is permission structures and guest page tables. Overall, we observe that there is a small baseline memory overhead as well as some overhead related to the size of the program.

A benchmark of *wasmtime* running the state-of-the-art StarlingMonkey JS runtime [23] showed that it needed 13ms at p50 VII to render a page with the same React page rendering benchmark as per Fig 2 as was used in Deno. It needed 13.1ms at p90 and 14.4ms at p99. Lacking in native performance and JIT-compiler support, WebAssembly had a 20x longer runtime than TinyKVM w/Deno. It was not possible to run the exact same official Deno binary within *wasmtime* as it can only run custom programs specifically supporting and compiled for WebAssembly. We chose the best JavaScript framework we could find for WebAssembly for this benchmark. In order to determine the reset times of *wasmtime* we had to find a custom branch with a reuse-feature that is still a work-in-progress, however with it we were able to run programs without resets. *wasmtime* resets lazily using `madvise(MADV_DONTNEED)` which means we have to measure multiple requests, as the next request will trigger copy-on-write pagetable changes. We found that on average a simple, tiny Rust hello-world required 23µs - 11µs = 12µs of effective reset time. A hello world JS required on average 314µs - 84µs = 230µs, and the complex React page rendering benchmark required 13197µs - 10569µs = 2628µs (2.6ms). This is expected as pagetable changes are not free. It requires a trip to the kernel, requires pagetable locking and when scaled up causes IPIs and coherency protocol chatter which means it will perform even more poorly at scale.

We instantiated V8 isolates using the same Hello World JS (Table X) and found that it took on average 0.75ms. For the React page rendering benchmark (Table XI) it took 23.0ms on average, averaged over 100 runs on the same hardware. Isolates don't have a reset mechanism and can only be instantiated and destroyed, making V8 isolates unsuitable for per-request isolation. While V8 does not support process forking, we found that Bun, which uses the JavaScriptCore engine, does not actively prevent it. Measuring the Hello World JS (Table VIII) we found that it took 1756µs. Process forking using the React page rendering benchmark (Table IX) was 8269µs on average. We believe this makes Bun process forking unsuitable for per-request isolation, although it is faster than *wasmtime*.

TABLE XIII
COMPARISON OF PER-REQUEST ISOLATION METHODS

Method	Reset Time	Guest JIT	Mem/Inst	Viable
Bun forking	1-8 ms	Yes	N/A	No
Firecracker	N/A**	Yes	50+MB	No
TinyKVM	7-48 µs	Yes	2.3MB	Yes
V8 isolates	1-25 ms	Yes	N/A	No
wasmtime	12-2628 µs*	No	N/A	Limited

*Instantiation only, lacks JIT support. **Does not support per-request isolation.

All benchmarks are latency measurements from the Oha HTTP benchmarking tool running for 10 seconds, and so 634µs is around 15770 requests. The benchmarks were run on an AMD Ryzen 9 7950X workstation and they can be replicated (including *wasmtime* benchmarks) by running `make bench` in the open-source KVM server GitHub repository [24].

React page rendering exercises typical serverless workload patterns including HTML rendering, memory allocation, and JIT optimization opportunities for CPU-intensive operations.

VI. Conclusion

This paper has presented improvements to TinyKVM that address limitations in existing per-request isolation solutions while meeting enterprise defense-in-depth requirements. By extending system calls with automatic file-descriptor tracking at the hypervisor level, we have eliminated a barrier to practical adoption while maintaining strong isolation guarantees. The improved reset mechanism that preserves page tables and JIT optimizations during state restoration delivers substantial performance improvements, particularly for dynamic language workloads that previously suffered from the performance-security trade-off inherent in existing solutions.

Hardware virtualization (TinyKVM), memory-safe runtime environments (Rust-based Deno), and established application sandboxing (V8 isolates) creates a practical solution that simultaneously achieves enterprise security requirements, native computational performance, and microsecond reset capabilities. This multilayer architecture demonstrates that defense-in-depth strategies that include per-request isolation does not have to compromise on performance.

By supporting unmodified executables, developers can use official builds of complex runtimes like Deno directly without maintenance burden, eliminating deployment friction while preserving the security guarantees that make per-request isolation valuable for high-security environments.

The framework's architecture opens several avenues for future work. Investigation into optimized file-descriptor tracking mechanisms could further reduce overhead for I/O-intensive workloads. KVM's support for `io_uring` presents a promising opportunity, as direct integration could reduce I/O overhead by potentially deferring some system calls, especially during VM reset. However, `io_uring` is currently considered to be unsafe [25] and has a long list of CVEs attached to it. Google has limited `io_uring` for several of their products, and is currently exploring ways to sandbox it. Additionally, extending the approach to support multi-process guest applications would broaden applicability to more complex microservice architectures.

As microservices architectures continue to evolve toward more granular isolation requirements, TinyKVM provides a foundation for secure, high-performance per-request isolation that maintains compatibility with existing application ecosystems. The elimination of custom-build requirements while achieving native performance positions this approach as a practical solution for production deployments requiring both security and high computational efficiency.

VII. References

References

[1] What is v8 - google blog. https://v8.dev. Accessed: 2025-05-27.

[2] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 185–200, 2017.

[3] Wasmtime: A fast and secure runtime for webassembly. https://wasmtime.dev/. Accessed: 2025-5-27.

[4] Alf-André Walla and Paal E. Engelstad. Introducing tinykvm for high-performance cdn edge services. In *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 18–25, 2023.

[5] Clémentine Maurice. *Micro-architectural side channels: Studying the attack surface from hardware to browsers*. Habilitation à diriger des recherches, Université de Lille, May 2023.

[6] M.R. Stytz. Considering defense in depth for software applications. *IEEE Security Privacy*, 2(1):72–75, 2004.

[7] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2018.

[8] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: lightweight virtualization for serverless applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 419–434, USA, 2020. USENIX Association.

[9] Wasmboxc: Simple, easy, and fast vm-less sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html. Accessed: 2024-10-18.

[10] Wasi http: A proposed webassembly system interface api. https://github.com/WebAssembly/wasi-http. Accessed: 2025-8-14.

[11] Safer with google: Advancing memory safety. https://security.googleblog.com/2024. Accessed: 2024-10-17.

[12] Eliminating memory safety vulnerabilities at the source. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html. Accessed: 2024-10-17.

[13] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[14] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, October 2018. USENIX Association.

[15] Yuwei Li, Yuan Chen, Shouling Ji, Xuhong Zhang, Guanglu Yan, Alex X. Liu, Chunming Wu, Zulie Pan, and Peng Lin. G-fuzz: A directed fuzzing framework for gvisor. *IEEE Transactions on Dependable and Secure Computing*, 21(1):168–185, January 2024.

[16] Xingyu Wang, Junzhao Du, and Hui Liu. Performance and isolation analysis of runc, gvisor and kata containers runtimes. *Cluster Computing*, 25(2):1497–1513, April 2022.

[17] Mikael Siidorow. *Survey of serverless edge computing for web applications*. PhD thesis, Bachelor's thesis, Aalto University, 05 2024, 2024.

[18] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: near-zero-cost transitions for webassembly and sfi. *Proceedings of the ACM on Programming Languages*, 6:1–30, 01 2022.

[19] Wasmtime horizontal scaling results in poor performance. https://github.com/bytecodealliance/wasmtime/issues/4637. Accessed: 2025-5-27.

[20] Performance of webassembly runtimes in 2023. https://00f.net/2023/01/04/webassembly-benchmark-2023/. Accessed: 2025-5-27.

[21] libriscv: Risc-v sandboxing library. Accessed: 2025-05-27.

[22] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[23] Starlingmonkey: A spidermonkey-based js runtime on webassembly. https://github.com/bytecodealliance/StarlingMonkey. Accessed: 2025-5-27.

[24] Kvm server: Fast per-request isolation for unmodified linux server executables. Accessed: 2025-05-27.

[25] Learnings from kctf vrp's 42 linux kernel exploits submissions. https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html. Accessed: 2025-5-27.