Towards Sandboxing Untrusted Agents in Userspace

Pekka Enberg
University of Helsinki

Ashwin Rao
University of Helsinki

Abstract

Many agentic systems, such as coding agents, execute untrusted, runtime-generated code and commands on machines they share with their users and with other agents, where a single defect or misstep can corrupt the host. However, traditional sandboxing isolation using virtual machines or containers makes it hard for agents to use the files and tools that make them useful. Furthermore, many of the environments in which agents run often lack the privileges to use VMs or containers, leaving no practical way to sandbox them.

As a solution, we propose Chimera, a userspace sandbox for unprivileged agent confinement that runs unmodified Linux binaries via system call interception with same-ISA dynamic binary translation. Chimera also provides a userspace virtual filesystem that allows practical sandboxing of the host filesystem. A preliminary evaluation of our Chimera prototype suggests that the approach imposes 1.1×–3.4× overhead on translated code and 1.4×–1.7× on filesystem operations. Chimera can therefore provide a viable sandbox that runs in precisely the restricted settings where virtual machines and containers cannot run.

1 Introduction

Agentic systems, such as coding agents, can execute arbitrary code generated at runtime, invoke external tools, and manipulate their host resources in unpredictable ways. For example, asking a coding agent to bootstrap a new project may have it install a different compiler or runtime version, or rewrite per-user configuration, breaking other software on the machine. Even when the agent is not malicious, it can still be unsafe: a software defect or a misread prompt is enough to cause it to touch files, network endpoints, or operating-system interfaces, thereby damaging the system. More fundamentally, ***agents often do not run in isolation***: a coding agent shares a developer's machine with the human driving it, and increasingly with other agents and sub-agents operating on the same files, tools, and resources. ***The challenge is therefore to let an agent work freely in this shared environment without letting it corrupt what it shares.***

This creates a tradeoff: stronger constraints improve safety but reduce agent utility. Therefore, instead of restricting agents we must sandbox their execution. However, existing sandboxing solutions are often too isolated from the agent's perspective and depend on hardware or OS capabilities that may be unavailable in the agent environment. Running a coding agent in an isolated VM or container is restrictive: it cannot reach the tools and source on the local development machine. Allowing the VM or container to access the host filesystem is no solution either: the agent can then damage the host. In the cloud, VM- and container-based sandboxing is often impossible, as nested virtualization or container creation is disabled for security. ***We therefore need a sandboxing technique that does not depend on privileged virtualization support and allows safe access to the host filesystem.***

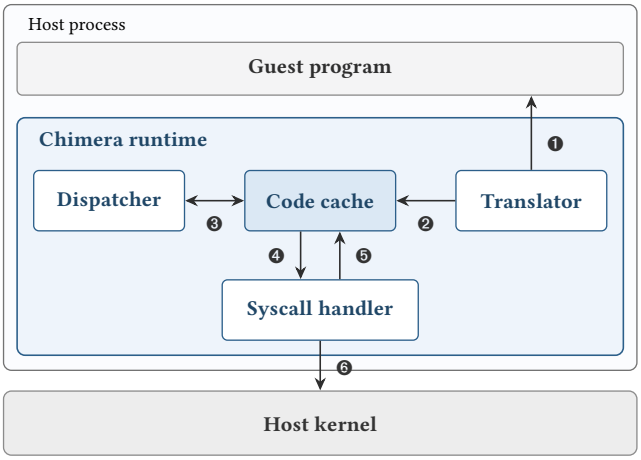

Figure 1: Chimera architecture. *The guest and the runtime share one address space. The translator reads the guest's basic blocks (1), and rewrites them into the code cache (2); the dispatcher then drives guest execution out of the cache (3). When translated code performs a system call it traps into the syscall handler (4), which either virtualizes the call inside the runtime (5) or mediates it to the host kernel (6).*

In this paper, we present Chimera, a userspace sandbox that closes the unprivileged agent confinement gap with same-ISA dynamic binary translation and system call interception (Figure 1). Chimera runs unmodified Linux binaries by rewriting system calls to trap into a runtime that shares the program's address space (§3), and our prototype already runs unmodified coding agents such as Claude Code [2] and Codex [8]. As a regular program, Chimera runs wherever agents run, including in containers and virtual machines, with no special hardware, kernel features, or administrative privileges. Chimera currently targets the Linux kernel ABI, but because Chimera intercepts system calls in userspace, the approach is portable to other host operating systems. Chimera also virtualizes the filesystem in userspace, letting the agent work on the live host without endangering it (§4). A preliminary evaluation suggests the overhead is low enough to make the approach practical (§5). As future work, the approach naturally extends to a copy-on-write overlay that captures the agent's writes, leaving the host untouched (§6).

2 Background and Motivation

Traditional isolation interfaces are challenging for agent sandboxing because agents need freedom but can lack privilege (§2.1). A userspace sandbox solves that but requires effective interception of the guest's system calls with low overhead (§2.2). A further challenge is that such a sandbox gives the guest no kernel process of its own, so the runtime must re-create the process abstraction the guest expects while sharing a single host process with it (§2.3).

2.1 Agent Sandboxing

Sandboxing cannot assume privileged virtualization support. Agents run in many different environments: developer machines, CI workers, managed platforms, containers, and virtual machines provided by an orchestration layer. In these environments, the application hosting the agent may not control the hypervisor, may not be able to create a new container, and may not have access to the operating-system features needed for stronger isolation mechanisms. A userspace sandbox is therefore an attractive solution because it runs as a regular program without requiring additional setup or administrative privilege.

Table 1 contrasts the conventional isolation primitives—micro-VMs, containers, and application kernels—with a userspace sandbox along the two axes that matter for agent deployment: the privilege needed to set the sandbox up and how much of the host the agent can still reach. A micro-VM such as Firecracker [1] runs the agent under a hardware hypervisor, and a container such as Docker [6] using kernel namespaces; both isolate the agent from the host and require privileged virtualization support to set up. An application kernel such as gVisor [15] instead re implements the system-call surface in userspace, but is still deployed as a container-style runtime over a configured root and rests on a `seccomp` or KVM platform. All three isolate the agent from the very host it must work on, whereas Chimera runs as an ordinary, unprivileged program that gives the agent confined access to the live host filesystem.

Table 1: Isolation primitives for sandboxing an agent.

Primitive	Privilege	Host access
Micro-VM	Hypervisor (VT-x/AMD-V)	Isolated
Container	Namespaces (`CAP_SYS_ADMIN`)	Bind-mounted
Application kernel	`seccomp`/KVM	Isolated
Chimera	Unprivileged	Virtualized

2.2 System Call Interception

System-call interception is the key technique for userspace sandboxing, and Table 2 summarizes the options. Library interposition through `LD_PRELOAD` intercepts only *library calls*, not system calls: a guest that issues a system call directly with a privileged instruction bypasses it entirely. Process trace (`ptrace`) intercepts every system call but delegates observation to a tracer in a separate task, so each call costs on the order of four context switches—an overhead that dominates system-call-intensive workloads [7]—and it is frequently disabled inside containers, the very environments where agents run. Syscall user dispatch (SUD) removes the second task: the kernel redirects an intercepted call to a `SIGSYS` handler on the calling thread itself, without privilege [5], but the per-call signal round trip remains a non-negligible overhead for system-call-intensive workloads [14]. Binary rewriting eliminates that cost by rewriting each `syscall` instruction in place to call a userspace hook, as in zpoline [14]; but the rewrite is a cooperative hook, not an enforcement boundary, because an adversarial guest can issue a syscall instruction the rewriter never saw.

Table 2: Comparison of system-call interception mechanisms. *Intercept is where a system call is trapped; Enforces whether that point is an enforcement boundary the guest cannot bypass.*

Mechanism	Intercept	Enforces	Overhead
Library interposition	None	No	None
Process trace	Kernel	Yes	High [7]
Syscall user dispatch	Kernel	No	Medium [14]
Binary rewriting	Userspace	No	Low [14]
DBT	Userspace	Yes	Medium (§5)

Dynamic binary translation (DBT). Dynamic binary translation intercepts system calls by rewriting the guest's code as it executes, so that each system-call instruction redirects control into a runtime that shares the guest's address space. As interception occurs entirely in userspace, DBT requires no special hardware or operating system support. Hence, it remains available even where the kernel-mediated mechanisms above are restricted or absent. Unlike in-place rewriting, DBT mediates all of the guest's control flow, so the guest cannot reach an un-vetted system-call instruction; interception is thus an enforcement boundary rather than a cooperative hook. ***This makes DBT a viable approach for sandboxing agents.*** The main challenge for DBT is keeping overhead low: even with extensive microarchitecture-specific tuning, translators such as MAMBO can still incur up to 10% overhead over native [4].

2.3 Process Virtualization

Dynamic binary translation implies a *runtime*: an ordinary, unprivileged process that loads the guest into its own address space, translates its code, and services the system calls it intercepts. This is what makes the approach deployable—the sandbox is just a program—but it exacts a price: the guest no longer has a kernel process of its own. Every service the kernel ties to the process abstraction—program loading, the address space, threads, signals, termination—is now a resource the guest shares with the runtime that confines it. The runtime must therefore *virtualize the process*: re-create the process abstraction in userspace, faithfully enough that the guest cannot tell, without ever losing control of the one real process both inhabit. This obligation has two faces.

Transparency. The guest must be unable to distinguish itself from native execution [3]. The runtime stands in for the kernel's `execve` and the dynamic linker: before the first guest instruction runs, it parses the ELF image, maps each `PT_LOAD` segment, loads the interpreter named by `PT_INTERP`, and builds the initial stack the System V ABI prescribes—`argc`, `argv`, `envp`, and the auxiliary vector. Any deviation—a missing auxiliary-vector entry, a clobbered register, an unexpected address-space layout—surfaces as a guest that crashes or, worse, behaves differently inside the sandbox than outside.

Coexistence. The guest and the runtime share every process-scoped kernel resource, so any resource the runtime does not mediate is either a correctness hazard or an escape hatch. Both link their own libc, and both heaps grow from the single program break, so unmediated `brk` lets one allocator corrupt the other's chunks.

A forwarded `exit_group` would kill the runtime along with the guest; a forwarded `execve` would replace the whole process image—runtime, translator, and code cache—with an untranslated program running natively, outside the sandbox. The design problem is therefore to decide, for each kernel service, whether to forward it, filter it, or re-implement it in userspace.

3 Chimera Design

Chimera runs an untrusted guest program using system call interception with same-ISA dynamic binary translation in a shared virtual memory space (Figure 1). The runtime virtualizes or mediates system calls (§3.1) based on whether the system calls impact execution safety. Process control (§3.2) and memory mapping (§3.3) system calls are the foundation as they allow linking and loading a program to co-exist in the runtime virtual memory space. Thread support (§3.4) and signals (§3.5) also need special support. Other system calls are mediated, allowing the runtime either to forward the system call to the host kernel or emulate it in userspace. Asynchronous system calls with `io_uring` (§3.6) are currently not supported.

3.1 System Call Virtualization

The runtime intercepts guest system calls using same-ISA dynamic binary translation in three phases: *pre-hook*, *dispatch*, and *post-hook*.

Pre-hook. The *pre-hook* runs before the call is serviced and is where the sandbox policy is applied. The hook observes every system call, validates its arguments, and may rewrite arguments or short-circuit the call entirely. For example, a pre-hook for the `open` system call can check whether a request path is in a specified allowlist and deny the call if it is not.

Dispatch. The *dispatch* phase services the call as either *virtualized* or *mediated*. A *virtualized* call is fully serviced by the runtime, bypassing the host kernel altogether; this is necessary when forwarding the call would break the runtime invariant of only executing translated code. For example, `execve` loads untranslated code into the address space and must therefore be virtualized. Other examples of virtualized system calls are `clone`, and `vfork` for process control (§3.2), and `mmap` and `mprotect` for memory management (§3.3), as well as out-of-band memory writers such as `ptrace`, `process_vm_writev`, and `/proc/self/mem`. In contrast, a *mediated* call is safe to forward to the host kernel or emulate within the sandbox. For example, the `read` and `write` system calls can be safely forwarded to the host OS, because they do not affect the runtime's control over execution. Similarly, the `open` system call can be safely forwarded to the kernel, assuming it has passed the sandbox policy defined in the pre-hook. A sandbox can also choose to handle a mediated call itself, for example, by emulating a virtual filesystem or network stack.

Post-hook. The *post-hook* runs after the call returns, allowing the runtime to record it or update the sandbox state.

3.2 Process Control

The runtime and the guest coexist within a single host process: rather than isolating the guest separately, the runtime runs guest code in its own address space and virtualizes the guest's system calls through dynamic binary translation, so it can mediate the guest's entire lifecycle—startup via `execve` and `execveat`, new flows of control via `clone`, `clone3`, and `vfork`, and termination via `exit` and `exit_group`.

Startup. A guest starts with `execve` or `execveat`, which replace the process image and transfer control to the new entry point. The runtime intercepts the call and loads the ELF executable into the address space without clobbering its own; if the executable names an interpreter, it loads that too. It then translates the first basic block of the entry point—the ELF entry or the interpreter—builds the process stack with `argv`, `envp`, and the auxiliary vector, and transfers control to the managed code.

Creating New Flows of Control. The `clone`, `clone3`, and `vfork` system calls all create a new flow of control, and the runtime services them according to the OS resource sharing they request. Without `CLONE_VM`, the call is an ordinary `fork`: the runtime forwards it to the host kernel, which duplicates the entire process, the guest program and runtime together, and the copy-on-write child resumes in translated guest code under its own copy of the runtime. A `clone` that creates a new *thread* (`CLONE_THREAD`) cannot be forwarded, because the host kernel would start the new task in untranslated guest code, outside the sandbox. The runtime instead spawns a host thread that runs the child guest through the dispatcher in the shared process and returns its kernel thread ID to the parent, giving each guest thread the backing host thread that the thread-state (§3.4) and signal (§3.5) machinery relies on. `CLONE_VM` together with `CLONE_VFORK` but without `CLONE_THREAD` is the `vfork`/`posix_spawn` pattern: a child that borrows the parent's memory only until it calls `execve`. Such a child cannot run as a host thread, since its `execve` would replace the image it shares with the parent. The runtime therefore emulates it as a fork, stripping the sharing flags so the child gets a copy-on-write image; because the `CLONE_VFORK` contract limits the child to `exec` and `_exit`, the copy is observationally equivalent. A pipe carries the child's `execve` outcome back to the parent, which blocks on it just as `CLONE_VFORK` would, so a failed exec is still reported synchronously. Plain `vfork` is emulated as a fork in the same way, though without the reporting pipe: the parent resumes immediately, and an exec failure surfaces through the child's exit status. The one remaining `clone` combination, `CLONE_VM` without `CLONE_THREAD` or `CLONE_VFORK`, requires a separate process that continues to share the caller's address space. The runtime can honor none of its readings: forwarding would run the child unsandboxed, a host thread would give it the parent's PID, and a fork would silently break the requested sharing. Therefore, the combination is refused with `EPERM`.

Termination. The `exit` and `exit_group` system calls terminate the calling thread or the entire thread group, respectively. Because the guest shares the runtime's host OS process and threads, forwarding either call to the host kernel would terminate the runtime, so the runtime virtualizes both. A thread-local `exit` stops that thread's dispatch loop at its next boundary; on the way out, the runtime zeroes the thread's `clear_child_tid` word and wakes one `futex` waiter on it, exactly as the kernel would, so a joining guest thread resumes. An `exit_group` publishes the exit status process-wide and interrupts every sibling: a thread executing translated code

observes the stop at its next safepoint poll (§3.5), and one parked in a blocking host system call is interrupted and stops at its run-loop boundary. The main thread waits for its siblings to drain, then returns the guest's exit status to the embedder. Guest mappings are not unmapped at exit; they are torn down only when a committed `execve` replaces the image.

3.3 Memory Mapping

The runtime virtualizes the memory mapping system calls. The `mmap`, `munmap`, `mremap`, and `mprotect` system calls are serviced by the runtime, which has authoritative control over the guest's address space. The runtime uses the host kernel to manipulate the address space co-owned by the runtime and the guest process, and the runtime also updates its own internal address-space metadata, adding, removing, or moving the affected region. The dispatcher always runs translated blocks from its own code cache. Together, the dispatcher and the memory-mapping subsystem maintain an invariant over the guest address space: ***the guest never executes its pages directly***. A second invariant covers writes: ***the bytes behind a translated address never change unobserved***. Every write that could reach them is either trapped, invalidating the stale translation before the new bytes can run, or refused, as the following two mechanisms describe.

Non-executable mappings. The runtime disallows any guest page from carrying execute permission in the host page tables to uphold the invariant that the guest never executes its own pages directly. As the memory mapping calls are virtualized, all the runtime has to do is clear the `PROT_EXEC` from the requested protection and substitute it with `PROT_READ` to ensure that the translator can read the pages. Therefore, any stray native jumps into guest code faults instead of silently running as untranslated. Translation is lazy, which means the runtime reads guest pages and translates only when the dispatcher first encounters them, not when the page is mapped, so memory mapping incurs no overhead. The `personality` call with `READ_IMPLIES_EXEC` is also refused by the runtime to avoid setting the executable bit on guest pages.

Self-modifying code. Non-executable mappings prevent the guest from running its own pages. However, a guest store can still rewrite code that the translator has already translated leaving a stale translation running. The runtime, therefore, write-protects every guest page it has translated code from. A guest store to such a page raises a synchronous fault that the runtime classifies as a self-modifying-code write: it drops the page's translations, restores write permission, and re-executes the store, so the write lands and the next execution of that code is translated afresh. The virtualized memory mapping calls compose with this: an `mprotect` or `mmap` over translated pages drops their translations too.

Out-of-band writers. The runtime must also observe every change to guest memory, so the guest cannot alter the bytes behind an already-translated address without the translator noticing. Therefore, the runtime refuses out-of-band write system calls with `EPERM`. The `remap_file_pages` system call rebinds the file pages under an existing mapping in place; `process_vm_writev` writes into the address space through a second kernel mapping of the target pages. (its read-only sibling `process_vm_readv` is forwarded); and `userfaultfd` lets a userspace monitor supply a page's contents on its first fault so that the guest can hand the translator one set of bytes and the faulting access another. The `shmat` and `shmdt` system calls are refused too, since a System V attach lands outside the `mmap` family's bookkeeping, which is untracked, and executable outright under `SHM_EXEC`. The `ptrace` system call is likewise refused: it reads and writes another process's registers and memory regardless of page protection and redirects its control flow, none of it through the translator.

3.4 Multithreading

A multithreaded guest runs as several host threads inside the single runtime process, one backing each guest thread (§3.2), all sharing the one guest address space and the translated code within it. A block one thread translates is thus immediately runnable by every other, and a mapping one installs or tears down is seen by all. A single lock serializes those updates, but the runtime holds it only to insert, look up, or remove a mapping or a translation—never across translated code or a blocking host system call—so threads translate and run concurrently, and one parked in a system call never stalls the others. Because a guest thread's identity is its backing host thread's kernel TID, a `futex` or `tgkill` directed at it reaches the right host thread.

Each thread also carries private state the runtime virtualizes. On x86-64 the FS segment register points to thread-local storage, so the runtime virtualizes `arch_prctl`: `ARCH_SET_FS` records the requested thread pointer in the guest's per-thread state and `ARCH_GET_FS` reads it back, without touching the kernel; a `clone` carrying `CLONE_SETTLS` gives the child its own pointer the same way. The `set_tid_address` call registers the per-thread word the kernel clears on thread exit; the runtime tracks it and, when the thread exits, performs the clear and the `futex` wakeup itself—the wakeup a joining thread waits on, so `pthread_join` completes. Since a real host thread backs each guest thread, `set_robust_list` and the remaining thread system calls are forwarded to the host unchanged.

3.5 Signals

The runtime must virtualize signals: delivered to untranslated code, a signal would fault, since guest pages are non-executable (§3.3). It therefore intercepts the signal system calls and installs its own handler for every signal the guest catches, so the kernel never delivers one directly to untranslated code.

When the guest installs a handler via `rt_sigaction`, the runtime registers its own with the host kernel and records the guest's handler address. Because the kernel can deliver a signal at any time, interrupting translated code or the runtime itself, that handler does only async-signal-safe work: it marks the signal pending in a virtualized process-wide bitmask, copies the kernel's `siginfo_t` into a pre-allocated slot, and sets the per-thread flag below. It is installed with `SA_SIGINFO`, so the `siginfo_t` reaches the guest handler intact, and *without* `SA_RESTART`, so a blocking host syscall forwarded for the guest is interrupted with `EINTR` back to the dispatcher.

The runtime delivers a pending signal at the next block boundary, where the guest's register file lives in memory rather than host registers. The dispatch loop drains the pending set on every re-entry,

but block chaining lets a loop of direct branches run in the cache without re-entering the dispatcher; so translated code also polls the per-thread flag at each loop-closing edge, re-entering within one iteration when it is set, and the signal is delivered.

3.6 Asynchronous System Calls with `io_uring`

The `io_uring` subsystem in the Linux kernel allows a thread to submit system calls that execute asynchronously. The runtime currently does *not support* the `io_uring` interface and, therefore, refuses all three syscalls: `io_uring_setup`, `io_uring_enter`, and `io_uring_register`. To support `io_uring`, the runtime needs to virtualize the submission queue mechanism to intercept system calls that require virtualization.

4 Filesystem Virtualization

While filesystem system calls never threaten the runtime's control over execution (§3.1), the runtime virtualizes them for filesystem sandboxing. The filesystem layer has two parts: a virtual filesystem interface (§4.1) that every guest filesystem call crosses, and host filesystem access provided as a userspace filesystem behind that interface (§4.2), confined and read-only by default.

4.1 Virtual Filesystem Interface

The runtime intercepts the guest's filesystem system calls and services them through a userspace virtual filesystem. A *system call personality* layer owns the Linux ABI: it marshals paths and structures out of guest memory, keeps the descriptor table together with each descriptor's offset and status flags, and lays out the buffers the guest reads back. The personality layer also has a *namespace* mapping each absolute guest path to the `Vfs` that serves it and the path within that filesystem. The `Vfs` object resolves a path to an open `File` object and services the path-addressed operations, such as `open`, `stat`, and so on. The `File` object, on the other hand, services `pread`, `pwrite`, `fstat`, `ftruncate`, and directory enumeration system calls.

In the `openat` system call, for example, the personality reads the path from guest memory, resolves it against the calling descriptor and the working directory, and hands it to the namespace, which returns the backing `Vfs` and the confined path within it; a write to a read-only mount is refused here with `EROFS`. The personality then calls `open` on that `Vfs` and installs the resulting `File` into the descriptor table, writing the descriptor number back into the guest's result register, exactly as the host kernel would. The descriptors a filesystem hands out are virtual, but each reserves a real kernel descriptor number, so the guest sees the low, dense, lowest-available numbers POSIX promises; a system call on a descriptor the table does not hold, or one the personality does not model, is forwarded to the host unchanged.

4.2 Host Filesystem Access

Chimera provides access to the host filesystem as a userspace filesystem behind the interface: a passthrough `Vfs` mounted at `/`, so the guest sees the live host tree while every operation it performs is mediated by the runtime. That mediation is what makes the access safe. All path resolution is scoped to the mounted subtree, so the guest cannot escape it by following a symbolic link, traversing to a parent directory, or naming an `execve` target outside it; in the common case of a single host mount, the scoping is delegated to the kernel with `openat2` under `RESOLVE_IN_ROOT`, in one system call. The mount is read-only by default: the guest reads the host freely, every mutating operation is refused with `EROFS`, and write access is an explicit opt-in. Because the host backend is just another filesystem behind the interface, safer backends slot in without touching the sandbox: the natural next step is a copy-on-write overlay that accepts the guest's writes while the host stays untouched—in userspace, with no FUSE mount or kernel support—which we return to as future work in §6.

5 Preliminary Evaluation

We evaluate steady-state translation overhead, system-call interception, and filesystem virtualization overheads.

5.1 Translation Overhead

We measure the per-operation cost of translated code once it is resident in the code cache, excluding process start-up and the one-time translation of each block. All measurements run on a 12-core AMD Ryzen 9 3900XT with 64 GB of RAM, under Fedora Linux 42 with kernel 6.18.12.

Methodology. We use nine self-timed microbenchmarks, each isolating a single guest execution pattern the translator has to handle. Every benchmark runs its hot loop with `CLOCK_MONOTONIC`, so the reported ns-per-operation figure reflects steady-state execution of translated code, not start-up or translation. Each workload runs natively and under Chimera. The `loop` benchmark is a tight integer dependency chain whose only control flow is the loop back-edge, a direct conditional branch that linking keeps resolved in the cache; it measures the floor of translated straight-line code with no cache exits. The `direct` benchmark adds data-dependent conditionals over a pseudo-random stream, both edges kept linked in the cache. The `indirect` benchmark calls through a 4-entry function-pointer table that stays within the inline indirect-branch lookup, while `indirect-mega` uses a 64-entry table that overflows to the fallback path; paired, they show how indirect-call cost scales with the number of distinct targets. The `ret` benchmark is a leaf called from a single site, the cheap monomorphic return, while `callsites` reaches one leaf through eight rotating callers so the return target stops being predictable and resolution falls to the slow path. The `syscall` benchmark issues a bare `getpid` per iteration, the only in-cache workload that leaves translated code on every operation—cache exit, trampoline, register-state save and restore, and re-entry—and so isolates the cost of crossing the sandbox boundary. `signal` raises `SIGUSR1` into an installed handler per iteration, exercising guest signal virtualization. The `mem` benchmark is a pointer-chasing dependent-load chain; loads run natively under same-ISA translation, so it serves as a control that should track native closely.

Results. Table 3 summarizes the measurements. Translated code whose control flow stays linked in the cache runs close to native speed: `loop` and `direct` cost 1.1×–1.2×—both edges of each conditional are kept resolved, and the guest's own branch mispredictions are paid identically either way—and the `mem` control tracks

Table 3: Steady-state per-operation cost of executing already-translated guest code, native versus under Chimera, in ns per operation.

Workload	Native	Chimera	Ratio
`loop`	1.758	1.860	1.1×
`direct`	5.604	6.978	1.2×
`indirect`	2.020	6.959	3.4×
`indirect-mega`	2.251	7.297	3.2×
`ret`	1.829	4.344	2.4×
`callsites`	4.475	14.348	3.2×
`syscall`	142.552	228.157	1.6×
`signal`	2543.168	3645.028	1.4×
`mem`	1.364	1.921	1.4×

native at 1.4×. The overhead concentrates where the translator must resolve a target that is not statically linked. Indirect calls cost 3.4× and 3.2×, and returns range from 2.4× when monomorphic (`ret`) to 3.2× when the return site varies (`callsites`), reflecting the code-cache lookup that maps a guest target to its translation; even so, the absolute cost stays below 15 ns per operation. Crossing the sandbox boundary is the most expensive in-cache operation: `syscall` rises from 142.6 ns to 228.2 ns (1.6×), the cost of the cache exit, trampoline, and register-state save and restore around the syscall handler. `signal` adds 1.4× because the per-operation cost is dominated by the kernel's own signal-delivery path, which both configurations pay; Chimera's virtualization of the signal frame contributes roughly 1.1 µs on top.

5.2 Filesystem Overhead

Methodology. The steady-state measurements isolate translator overhead, but the userspace virtual filesystem (§4) adds cost to every filesystem system call. We decompose that cost with three microbenchmarks, each self-timed the same way over `/usr/lib/os-release` for 200 000 iterations. `stat` exercises path resolution and confinement, `open` adds the virtual descriptor-table install, and `read` isolates the per-descriptor dispatch with no path resolution.

Results. Table 4 reports the results. `stat` and `read` stay closest to native at 1.4× each, showing that confined path resolution and the per-descriptor dispatch are both cheap. `open` carries the most virtualization work at 1.7×, combining path resolution with virtual descriptor-table install and teardown.

Table 4: Filesystem virtualization overhead: per-operation cost of guest filesystem system calls over `/usr/lib/os-release`, native versus under Chimera, in ns per operation.

Operation	Native	Chimera	Ratio
`stat`	1153.9	1639.1	1.4×
`open`	2906.0	5030.2	1.7×
`read`	627.4	890.7	1.4×

6 Related Work and Discussion

Unprivileged agent confinement. Sandlock shares Chimera's goal of running untrusted AI-agent code on developer machines [13], but builds on Landlock, seccomp-bpf, and a seccomp-notify supervisor rather than dynamic binary modification. Its seccomp-based copy-on-write workspace with commit/abort and FUSE-based BranchFS parallel Chimera's userspace filesystem virtualization, and beyond the filesystem Sandlock also confines network, IPC, and HTTP-level access. The most mature userspace sandbox, gVisor, is also deployed to confine untrusted and model-generated code, but as a heavyweight application kernel: it re-implements the Linux system-call surface in a separate address space using `seccomp` or KVM [15]. That separate address space makes its memory isolation strictly stronger than Chimera's, which shares one address space with the guest; in return, gVisor depends on kernel features and a container-style setup that Chimera does not, and it confines the guest to a configured root rather than running it against the live host. Where the hardware is available, that shared-address-space boundary could be hardened with Intel Memory Protection Keys, as in ERIM [10], whose soundness rests on the untrusted code never executing a `WRPKRU`, which is a guarantee Chimera's translation of all guest code can provide.

Copy-on-write filesystems. Chimera's read-only host mounts keep the host safe at the cost of making the agent's writes fail; the natural extension is a copy-on-write overlay that redirects guest writes into a writable upper layer over the live host tree, so the agent works freely while its changes can be inspected, committed, or discarded wholesale. The virtual filesystem interface is built for this: an overlay is just another filesystem stacked in the namespace (§4.2). Recent systems underline the demand for such branchable agent filesystems. BranchFS gives agents copy-on-write filesystem branches with atomic commit through a FUSE filesystem, and proposes a `branch()` kernel primitive for isolating whole exploration contexts [12]. AgentFS implements a copy-on-write filesystem inside a single SQLite file, mounted through FUSE on Linux, so an agent's session can be snapshotted, forked, and shared as one file [9]. Both, however, inherit FUSE's limitations: it is unavailable in the restricted environments Chimera targets, and it routes every request through a cross-process daemon [11]. A Chimera overlay would instead live in the same in-process userspace VFS as the rest of the sandbox, with no such round-trip.

7 Conclusions

Agents on machines shared with users and other agents are hard to sandbox: VMs and containers either restrict the agent too much or are unavailable where agents run. Chimera offers an alternative by running unmodified binaries through same-ISA dynamic binary translation, making system-call interception an enforcement boundary that needs no hardware, kernel feature, or privilege. It also virtualizes the filesystem in userspace, confining the agent to a read-only view of the live host, with a copy-on-write overlay as the natural next step. A prototype and preliminary evaluation suggest the approach is practical: an unprivileged userspace sandbox that runs where VMs and containers cannot. Broadening guest coverage and hardening the boundary remain, but the result points toward sandboxing both safe and useful where agents work.

References

[1] Alexandru Agache, Marc Brooker, Andreea Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 419–434.

[2] Anthropic. 2025. Claude Code. https://www.anthropic.com/claude-code.

[3] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. Association for Computing Machinery, New York, NY, USA, 133–144. doi:10.1145/2151024.2151043

[4] Guillermo Callaghan, Cosmin Gorgovan, and Mikel Luján. 2020. Optimising Dynamic Binary Modification Across 64-bit Arm Microarchitectures. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. Association for Computing Machinery, New York, NY, USA, 185–197. doi:10.1145/3381052.3381322

[5] Gabriel Krisman Bertazi. 2021. Syscall User Dispatch. Linux kernel documentation. https://docs.kernel.org/admin-guide/syscall-user-dispatch.html.

[6] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014, 239 (2014).

[7] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 167–182. doi:10.1145/3373376.3378519

[8] OpenAI. 2025. OpenAI Codex. https://openai.com/codex/.

[9] Turso. 2026. AgentFS: Isolated Filesystem Access for AI Agents. https://www.agentfs.ai/.

[10] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association, 1221–1238. https://www.usenix.org/system/files/sec19-vahldiek-oberwagner_0.pdf

[11] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*. USENIX Association, 59–72. https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[12] Cong Wang and Yusheng Zheng. 2026. Fork, Explore, Commit: OS Primitives for Agentic Exploration. arXiv:2602.08199 https://arxiv.org/abs/2602.08199.

[13] Cong Wang and Yusheng Zheng. 2026. Sandlock: Confining AI Agent Code with Unprivileged Linux Primitives. arXiv:2605.26298 https://arxiv.org/abs/2605.26298.

[14] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: A System Call Hook Mechanism Based on Binary Rewriting. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Boston, MA, USA, 293–300.

[15] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association.