

Sandlock: Unprivileged Process Sandboxing

Cong Wang

Founder and CEO, Multikernel Technologies, Inc.

The AI Agent Threat Model

- Agents are **not adversaries**, they follow prompts
- Threats: prompt injection, misinterpreted instructions, tool misuse
- `rm -rf /` happens because of a bad prompt, not a kernel exploit

"You do not need a hypervisor to stop `rm -rf /`. You need a filesystem allowlist."

Isolation Is Not Security

- A microVM agent can still read your **SSH keys** because you mounted them
- A container agent reaches **169.254.169.254** and steals IAM creds
- All tools share the **same permissions** inside one big sandbox

What we actually need: deny-by-default policy, per-tool-call granularity, no root

Per-Tool-Call Policy

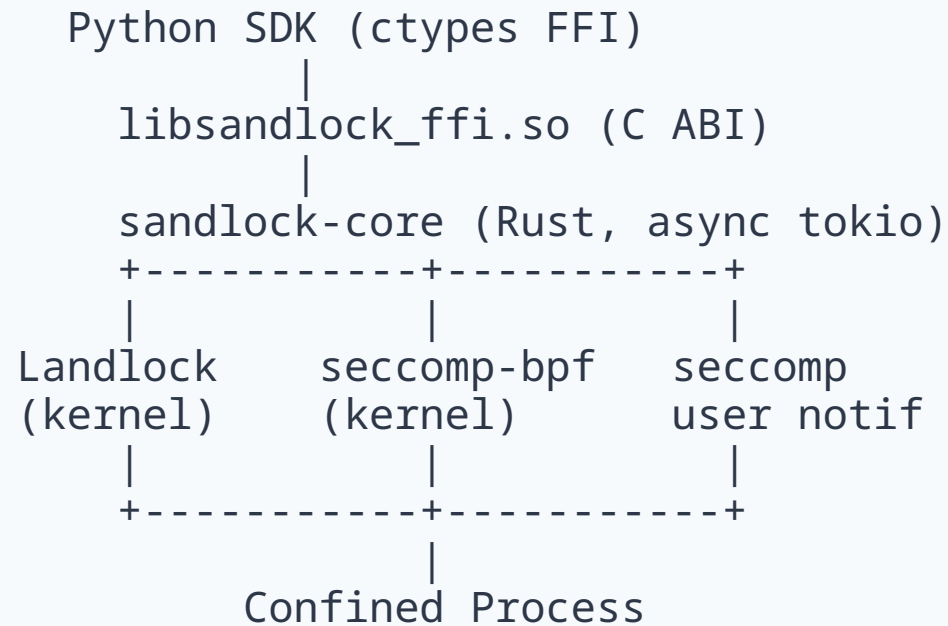
The problem: one sandbox for all tools = union of all permissions

Sandlock's approach: each tool invocation gets its own policy

- `web_fetch` → fork + sandbox with network only, no filesystem write
- `file_write` → fork + sandbox with filesystem only, no network
- `run_tests` → fork + sandbox with read-only source, no network

Least privilege at the **right granularity**: per-tool, not per-agent.

Architecture Overview



- **Layer 1:** Landlock = filesystem + network (kernel-enforced, zero overhead)
- **Layer 2:** seccomp-bpf = syscall gating (~1KB cBPF, kernel-space)

- **Layer 3:** seccomp user notif = async supervisor (userspace policy engine)



Confinement Flow: between fork() and exec()

1. `PR_SET_NO_NEW_PRIVS` (irreversible)
2. Landlock ruleset created + `landlock_restrict_self()`
3. seccomp-bpf filter installed with `SECCOMP_FILTER_FLAG_NEW_LISTENER`
4. Notification fd sent to parent via Unix socket
5. Parent starts **async supervisor** (tokio event loop)
6. Optional: vDSO patched for time/random virtualization

Total setup: 7ms. No namespace, no cgroup, no mount.

Landlock: ABI Versions

ABI	Capabilities Added
v1	EXECUTE, READ_FILE, READ_DIR, WRITE_FILE, REMOVE_, MAKE_
v2	+ REFER (cross-mount renames/links)
v3	+ TRUNCATE
v4	+ TCP bind/connect (LANDLOCK_RULE_NET_PORT)
v5	+ IOCTL_DEV
v6	+ Abstract Unix socket scoping, signal scoping

Landlock: Confinement

1. `landlock_create_ruleset()` with handled access masks
2. `landlock_add_rule()` for each allowed path and TCP port
3. `landlock_restrict_self()` to enforce

Paths and ports not in the ruleset are **denied by default**. Kernel enforcement means **zero steady-state overhead**.

seccomp-bpf: Filter Layout

Hand-assembled cBPF filter:

1. **Arch check:** verify `AUDIT_ARCH_X86_64`, `KILL_PROCESS` on mismatch
2. **Argument filters:** pre-built filters on syscall args
3. **NOTIF chain:** syscalls routed to `USER_NOTIF` (connect, openat, clone, ...)
4. **DENY chain:** return `ERRNO(EPERM)` for ~90+ dangerous syscalls
5. **Default:** `ALLOW`

seccomp-bpf: Flags and Denied Syscalls

Key flags:

- `SECCOMP_FILTER_FLAG_NEW_LISTENER` (5.0+): returns notification fd
- `SECCOMP_FILTER_FLAG_WAIT_KILLABLE_RECV` (5.19+): child killable during recv

Always denied: mount, umount2, pivot_root, unshare, setns, ptrace, bpf, perf_event_open, init_module, reboot, kexec_load, ...

The Async Supervisor

The parent runs a **tokio event loop** receiving seccomp notifications:

1. `ioctl(notif_fd, SECCOMP_IOCTL_NOTIF_RECV)` blocks until a syscall is intercepted
2. Look up the syscall number in a **dispatch table** (registered per policy)
3. Run the handler chain; produce an action
4. `ioctl(notif_fd, SECCOMP_IOCTL_NOTIF_SEND)` to respond

Only syscalls relevant to the active policy are registered. Unintercepted syscalls hit zero supervisor overhead.

Supervisor: Actions

The supervisor can respond to an intercepted syscall with:

- **Continue:** let the syscall proceed normally
- **Errno:** return an error to the child
- **ReturnValue:** synthetic return value (e.g., for getrandom)
- **InjectFd:** give the child a file descriptor (COW opens, /proc, etc.)
- **Hold:** freeze the notification (for checkpoints)
- **Kill:** send SIGKILL (OOM enforcement)

Supervisor: TOCTOU Safety

On-behalf execution for network syscalls (`connect`, `sendto`, `sendmsg`):

- Supervisor copies the `sockaddr` into its own memory, validates, then issues the syscall itself — the kernel never re-reads the child's (mutable) buffer

Atomic argument reads: `process_vm_readv` pulls paths and sockaddrs in a single syscall, immune to mid-read mutation

`SECCOMP_IOCTL_NOTIF_ID_VALID` **bracketing:** every `read_child_mem` / `write_child_mem` is guarded *before and after* the transfer, so a child that exits or is replaced mid-operation is detected

Path canonicalization: `AT_FDCWD` resolved via `/proc/[pid]/cwd`, then `canonicalize()` — closes symlink-swap windows before the policy check

`pidfd_getfd`: guards fd duplication against PID recycling



Supervisor: File Descriptor Injection

`SECCOMP_IOCTL_NOTIF_ADDFD` (5.9+): inject a supervisor fd into the child's fd table

- **Inject + continue:** place fd at a specific slot, then let the original syscall proceed
- **Inject + send** (`SECCOMP_ADDFD_FLAG_SEND`, 5.14+): inject and complete in one ioctl, the injected fd becomes the syscall's return value

Used for: COW file opens, /proc virtualization, `/dev/urandom` replacement, network connect on behalf

Network: Domain Allowlist

Problem: Landlock v4 only does TCP port filtering, not IP/domain filtering

Solution: intercept `connect(2)` via seccomp user notification

1. Copy `sockaddr` from child memory (`process_vm_readv`)
2. Parse IP from `AF_INET` / `AF_INET6`
3. Check against pre-resolved allowlist (DNS done at sandbox creation)
4. Duplicate child's socket via `pidfd_getfd` (Linux 5.6+)
5. Call `connect()` in supervisor with validated address
6. Return result to child

Network: HTTP-Level ACL

```
sandbox run \  
  --http-allow "POST api.openai.com/v1/chat/completions" \  
  --http-deny  "* */admin/*" \  
  -- python3 agent.py
```

Transparent MITM proxy:

- Supervisor redirects `connect()` to a local proxy
- Proxy terminates TLS (self-signed CA injected into sandbox)
- Inspects HTTP method + Host + path against allow/deny rules

Anti-spoofing: original dest IP recorded before redirect; proxy verifies the `Host` header resolves to that IP (DNS cache with 30s TTL)

Network: Port Mapping

Problem: two sandboxes both want to bind `6379` on the host

Inside the sandbox: every child keeps using its native port. The supervisor transparently assigns a unique real port per sandbox, and the child never sees the remap.

Outside the sandbox: each sandbox's `{virtual → real}` map is published to a shared registry keyed by sandbox hostname.

Discovery: `sandbox network ls` lists every sandbox's real ports; peer sandboxes look each other up by hostname to connect.

COW Filesystem (No Root)

Key insight: overlayfs requires `CAP_SYS_ADMIN`. We build COW in userspace via seccomp notification instead.

Write path: intercept `openat` with write flags

- If file already in upper → open it, inject fd
- Otherwise → copy from lower to upper, then open and inject fd

Delete path: intercept `unlinkat`, add path to an in-memory deleted set

Resolution: upper first, then lower (same semantics as overlayfs)

COW Filesystem: Details

- `upper/` directory keyed by UUID per sandbox instance
- Disk quota: track bytes written per sandbox, deny on exceed
- On exit: **commit** changes to workdir or **discard** (dry-run mode)
- Intercepted syscalls: `openat`, `unlinkat`, `mkdirat`, `renameat2`, `statx`, `getdents64`

COW: Three Backends

Backend	Root	Mechanism	Use Case
Seccomp	No	User notification intercept	Default, unprivileged
OverlayFS	Yes	Kernel overlayfs mount	Performance-critical
BranchFS	No	Custom kernel FS (ioctl)	Experimental

Trade-off: higher per-write overhead than kernel overlayfs, but no root, no mount namespace, and composable with chroot.

Chroot Without Root

Landlock-based path restriction + seccomp path rewriting:

- Supervisor intercepts `openat` and resolves paths within a configured root
- Symlink following with bounds checking prevents `/../` escapes
- Opens the resolved path in supervisor, injects fd into child
- Mount mappings via longest-prefix matching for overlapping mounts

Composable with seccomp COW in the same sandbox. No `chroot(2)`, no `CAP_SYS_CHROOT`, no mount namespace.

COW fork(): Fast Parallel Workloads

Pattern: initialize once, fork many

1. Parent loads expensive state (ML model, JIT warmup)
2. `fork()` gives each child a COW copy of the address space
3. Each child processes a shard independently
4. Results collected via pipes

Performance: ~530 us per fork, ~1,900 forks/sec sustained

Uses raw `fork()` (syscall NR 57 on x86-64) bypassing libc wrappers to avoid seccomp clone interception overhead. No full process restart, no VM overhead.

vDSO Patching

Problem: vDSO calls (`clock_gettime`, `gettimeofday`) execute in userspace via the kernel's shared page, **bypassing seccomp entirely**

Solution: overwrite vDSO symbols with real syscall stubs

1. Find `[vdso]` base via `/proc/{pid}/maps`
2. Read vDSO ELF via `/proc/{pid}/mem`, parse dynsyms
3. Generate x86-64 stubs (8 bytes: `mov eax, NR; syscall; ret`)
4. Write stubs back via `/proc/{pid}/mem`

Now all time calls go through the real syscall path, and seccomp can intercept them.

vDSO: Time Offset Stubs

For `--time-start`, a longer stub adds an offset to `timespec.tv_sec` after the real syscall:

- Shifts `CLOCK_REALTIME` and `CLOCK_REALTIME_COARSE` in the stub itself
- Monotonic clocks and absolute timers (`TIMER_ABSTIME`) are handled by the supervisor
- Offset calculated as `desired_time - current_real_time`

Deterministic Execution

Seeded randomness (ChaCha8 PRNG):

- `getrandom()`: supervisor fills child's buffer via `process_vm_writev` with PRNG output, returns synthetic success
- `/dev/urandom`: supervisor creates a `memfd` filled with 1 MiB of PRNG bytes, injects it via `SECCOMP_ADDFD_FLAG_SEND`

Frozen time:

- vDSO patched to route time calls through seccomp
- Offset applied: `virtual_time = real_time + (desired - actual)`

```
sandbox run --time-start "2000-01-01T00:00:00Z" \  
--random-seed 42 -- ./build.sh
```

Same seed = same output, every time. Reproducible builds, deterministic tests.



/proc Virtualization

The sandbox sees an isolated view of the system:

Virtual File	Content
<code>/proc/cpuinfo</code>	N processor entries (configurable)
<code>/proc/meminfo</code>	MemTotal = sandbox memory limit
<code>/proc/loadavg</code>	Sandbox-local EWMA (kernel algorithm, 5s sampling)
<code>/proc/uptime</code>	Time since sandbox start
<code>/proc/[pid]/*</code>	Only sandbox PIDs visible

Mechanism: intercept `openat()`, create `memfd` with generated content, inject via `SECCOMP_ADDFD_FLAG_SEND`

/proc: Sensitive Path Blocking

Denied with EACCES (not just invisible):

- `/proc/kcore`, `/proc/kmsg`, `/proc/kallsyms`
- `/proc/keys`, `/proc/key-users`
- `/proc/sysrq-trigger`
- `/sys/firmware/*`, `/sys/kernel/security/*`

Resource Limits: Memory

No cgroups, no root:

- Intercept `mmap`, `munmap`, `brk`, `mremap`, `shmget`
- Track `mem_used` vs. configured limit
- On exceed: `SIGKILL` to process group + `Errno(ENOMEM)`

Trade-off: memory limits are approximate (mmap tracking, not kernel-enforced OOM). But no root, no cgroupfs mount, no systemd dependency.

Resource Limits: Processes

- Intercept `clone` / `clone3` / `vfork`, track count
- `CLONE_THREAD` (threads) do not count
- Namespace flags (`CLONE_NEWNS`, `CLONE_NEWPID`, ...) always denied with `EPERM`

Namespace flags are always blocked. This prevents sandbox escape via `unshare()` equivalent through clone flags.

Pipelines: Multi-Stage Sandboxing

Compose sandboxed stages with different policies:

```
[Planner: network, no fs] —pipe—> [Executor: fs, no network]
```

- Each stage gets its own sandbox policy
- Data flows through kernel pipes only, preventing exfiltration
- N-1 pipes for N stages; parent closes write ends for EOF propagation

Use case: XOA mode, where the reasoning stage can be networked while the action stage is isolated from the network.

Checkpoints and Freeze

Checkpoint: capture full process state via `ptrace(PTRACE_SEIZE)`

- Registers, memory maps, writable memory segments

Freeze: hold all fork/clone notifications until resumed

- Use case: pause sandbox for policy inspection, then resume

Dynamic Policy Callbacks

- Per-syscall decisions with rich metadata (path, IP, argv)
- Runtime policy modification: expand or shrink allowed IPs, reduce memory/process limits, deny paths
- Expansions are reversible (up to a ceiling); restrictions are permanent

Performance

Metric	Bare Metal	Sandlock	Docker
<code>/bin/echo</code> startup	2 ms	7 ms	307 ms
Redis SET	82K rps	80K rps	52K rps
Redis GET	79K rps	77K rps	53K rps
p99 latency	0.5 ms	0.6 ms	1.5 ms

44x faster startup than Docker. **97% native throughput, 2.5x lower tail latency.**

Kernel Version Requirements

Feature	Minimum Kernel
seccomp user notification	5.0
<code>pidfd_getfd</code> (fd dup across processes)	5.6
<code>SECCOMP_IOCTL_NOTIF_ADDFD</code> (fd injection)	5.9
Landlock filesystem	5.13
Atomic fd inject + respond	5.14
Killable supervisor recv	5.19
Landlock network (TCP bind/connect)	6.7
Landlock IPC + signal scoping (ABI v6)	6.12

Design Trade-offs

Choice	Trade-off
No namespaces	Cannot virtualize UID/GID or hostname
No cgroups	Memory limits are approximate (mmap tracking)
seccomp COW	Higher per-write overhead than kernel overlays
vDSO patching	Architecture-specific (x86-64 today)
MITM proxy	Requires CA injection, adds HTTPS latency
Process-based	No thread-level sandboxing

Philosophy: favor unprivileged operation and simplicity over maximum isolation. The threat model (AI agents, not adversaries) makes this trade-off correct.

Comparison: Sandlock vs. Alternatives

Feature	Sandlock	Docker	gVisor	Firecracker
Root required	No	Yes	No*	Yes
Startup time	7 ms	300 ms	150 ms	125 ms
Steady-state overhead	~3%	~35%	~20%	~5%
Per-tool-call sandbox	Yes	No	No	No
HTTP-level ACL	Yes	No	No	No
Deterministic execution	Yes	No	No	No
Dynamic policy	Yes	No	No	No

*gVisor rootless mode has limitations



Key Takeaways

- **Three-layer enforcement:** Landlock + seccomp-bpf + seccomp user notification
- **Unprivileged:** no root, no cgroup, no namespace, no daemon
- **Per-tool-call granularity:** fork + confine in < 1ms
- **seccomp user notification is underrated:** enables COW, /proc virtualization, HTTP ACL, deterministic execution, all without root
- **44x faster startup, 97% native throughput**

"The right abstraction for AI sandboxing is policy, not isolation."

Questions and Discussion

Contact: cwang@multikernel.io

Open Source: github.com/multikernel/sandlock