The Capability-Container Pattern: Infrastructure-Level Security for Autonomous AI Agents

Ricardo Ledan

February 2026

Abstract

Autonomous AI agents that invoke external tools via protocols like MCP (Model Context Protocol) present a novel attack surface: the agent-tool boundary. Existing frameworks rely on prompt-level safeguards or protocol-level trust, both of which are insufficient against adversarial inputs, tool poisoning, and credential leakage. We present the *Capability-Container Pattern*, an infrastructure-level security architecture where agents never directly access tools. All tool invocations flow through a mediation gateway into isolated containers, each provisioned with only the capabilities it requires. We describe the threat model, detail six defense-in-depth layers (container isolation, network egress filtering, credential vaults, audit logging, secret scanning, and human-in-the-loop gates), and provide benchmark results from a reference implementation (self-evaluated with synthetic test data; see §6.8). Our evaluation uses a 1,534-sample corpus—900 synthetic secrets (100 per type, 9 types), 514 benign samples across 7 categories, and 120 adversarial evasion samples—with Clopper–Pearson confidence intervals and comparative analysis against `detect-secrets`. Gateway logic overhead is $\sim$0.025 ms (0.005% of LLM inference latency, cross-run 95% CI: [0.022, 0.028]), with 100% secret detection rate (CI: [0.996, 1.000]), $F_1 = 0.991$ vs. 0.779 for `detect-secrets`, and 0.19% false positive rate (CI: [0.000, 0.011]). The pattern provides defense coverage for 6/7 reconstructed 2025 MCP breach scenarios—reducing the attack surface from unbounded host access to a narrow, auditable, policy-enforced channel.

1 Introduction & Motivation

The year 2025 saw the emergence of a new class of software vulnerability: the agent-tool boundary exploit. From WhatsApp chat history exfiltration in April (disclosed by Invariant Labs) to the Smithery registry vulnerability responsibly disclosed in October [GitGuardian, 2025] that could have exposed 3,000+ MCP servers (no evidence of exploitation), a clear pattern emerged—the protocols designed for agent-tool communication provide no security enforcement at all.

1.1 Problem Statement

The Model Context Protocol (MCP) standardizes how AI agents invoke external tools via JSON-RPC 2.0. It defines message formats, tool discovery, and communication patterns. What it does *not* define is:

- How tools are isolated from each other and the host
- How credentials are managed and injected
- How network access is constrained per-tool
- How tool invocations are audited
- How dangerous operations are gated on human approval

This gap between protocol specification and security enforcement is not merely theoretical. The 2025 MCP vulnerability timeline (Section 3) demonstrates that every major disclosed vulnerability exploited this exact gap.

1.2 Key Argument

Protocol-level security is fundamentally insufficient for autonomous AI agents. An agent that can send arbitrary JSON-RPC messages can bypass any protocol-level restriction. Just as web security requires infrastructure enforcement (TLS, firewalls, WAFs) beyond HTTP, agent security requires infrastructure enforcement beyond MCP.

1.3 Our Contribution

We present the **Capability-Container Pattern**, an architectural pattern where:

1. Each tool runs in an isolated container with only the capabilities it needs
2. All communication flows through a mediation gateway that enforces security policies
3. Credentials are injected at runtime and never visible to the agent or tools
4. Every operation is logged to an immutable audit trail
5. High-risk operations require human approval before execution

We provide: (a) a formal threat model with attack surface analysis, (b) a reference implementation (Harombe) with six defense-in-depth layers, (c) benchmark results showing <1 ms audit overhead, and (d) a mapping of 2025 MCP breaches to specific defense layers that would have prevented them.

1.4 Scope

This paper focuses on the security architecture for *tool execution* in autonomous AI agents. We do not address LLM alignment or output safety, training data poisoning, model extraction attacks, or multi-agent coordination security (future work).

2 Background & Related Work

2.1 The Model Context Protocol (MCP)

MCP is an open standard [Anthropic, 2024] that defines how AI agents discover and invoke external tools via JSON-RPC 2.0. It specifies tool discovery, tool invocation with typed parameters, transport (stdio or streamable-HTTP), and content types (text, images, resource references). MCP has become the de facto standard for agent-tool communication, with thousands of community-built servers.

What MCP does NOT specify: isolation between tools, credential lifecycle management, network access control per tool, audit logging, or human approval workflows. This omission is by design—MCP is a wire protocol, not a security framework. The specification's own security best practices acknowledge that "MCP servers should be launched with restricted access. . . using platform-appropriate sandboxing technologies"—but provides no mechanism to enforce this.

2.2 Autonomous AI Agent Frameworks

Current agent frameworks focus on orchestration: task decomposition, multi-agent coordination, memory, and planning. Security is treated as an afterthought (Table 1).

AutoGen offers Docker-based code execution but does not extend isolation to all tools, does not filter network egress, and does not manage credentials. OpenClaw [Steinberger, 2025] is an open-source personal AI assistant that offers optional sandboxed operation and integrates with 50+ services, but provides no structured security enforcement—users choose between full system

Table 1: Security feature comparison of agent frameworks.

Framework	Isolation	Network	Credentials	Audit	HITL
CrewAI	None	None	None	None	None
LangGraph	None	None	None	Cloud (LangSmith)	Node-level
AutoGen	Optional Docker	None	None	None	None
OpenClaw	Optional sandbox	None	None	None	None
Ours	**Per-tool**	**Per-container**	**Vault**	**Local+redact**	**Risk-based**

access or a basic sandbox, with no per-tool isolation, credential vaulting, network filtering, audit logging, or secret scanning.

2.3 Container and Sandbox Security

Container isolation (Docker, Podman) provides namespace separation for PIDs, network, filesystem, and IPC. However, containers share the host kernel—a known limitation extensively documented in the security literature.

Empirical evidence of container weakness. Lin et al. [Lin et al., 2018] found that 56.82% of typical exploits succeed against default Docker configuration, and that namespace+cgroup isolation blocks only 21.62% of privilege escalation attacks—while Seccomp+capabilities+MAC blocks 67.57%. Jarkas et al. [Jarkas et al., 2025] cataloged 200+ container vulnerabilities across 47 exploit types and 11 attack vectors (2013–2024). Reeves et al. [Reeves et al., 2021] analyzed 59 CVEs across 11 runtimes: all 9 documented escape exploits resulted from a host component leaked into the container; user namespace remapping stopped 7 of 9.

Recent container escapes include CVE-2024-21626 (runc "Leaky Vessels": file descriptor leak), CVE-2024-1086 (nf_tables use-after-free; PoC reports 99.4% success on KernelCTF images [Notselwyn, 2024]), CVE-2025-23266 (NVIDIA "NVIDIAScape": CVSS 9.0), CVE-2025-31133/52565/52881 (three runc mount-handling escapes), and DockerDash [Levi and Noma Security, 2025]: an AI-specific attack where malicious Docker image labels exploit the Gordon AI → MCP Gateway pipeline.

Namespace and eBPF limitations. Sun et al. [Sun et al., 2018] showed containers cannot independently define security policies due to shared kernel. He et al. [He et al., 2023] demonstrated that eBPF tracing features break container isolation—found 5 vulnerable cloud services.

gVisor offers a middle ground: a user-space kernel reducing ∼300 syscalls to ∼83 (72% reduction). Jang et al. [Jang et al., 2022] quantified gVisor as 4.2–7.5× more secure than standard runtimes. Young et al. [Young et al., 2019] measured 2.2–216× overhead depending on operation (simple syscalls to Gofer-mediated file I/O), but near-native CPU performance. Google's Kubernetes Agent Sandbox (2025) chose gVisor as its default runtime for AI agent workloads [Google, 2025].

Firecracker microVMs [Agache et al., 2020] provide hardware-enforced isolation via KVM with 125 ms boot time and ∼3 MiB per-VM memory overhead. However, Weissman et al. [Weissman et al., 2023] showed limited Spectre/MDS protection, and Xiao et al. [Xiao et al., 2023] demonstrated exploitable operation forwarding.

Our architecture uses Docker containers with aggressive hardening (custom Seccomp profiles, dropped capabilities, user namespace remapping, read-only filesystems) as the default, with gVisor as the recommended production layer. Defense-in-depth ensures that even if container isolation is breached, network egress filtering, credential vaults, and secret scanning provide additional barriers.

2.4 Related Academic Work

AgentBound [Bühler et al., 2025] introduces an access control framework inspired by Android's permission model, auto-generating security policies from MCP server source code with 80.9% accuracy. Our work is complementary: AgentBound enforces permissions at the tool definition level (software enforcement), while we enforce at the infrastructure level (hardware-enforced container boundaries).

Fault-Tolerant Sandboxing [Yan, 2025] proposes transactional filesystem snapshots for AI coding agents, achieving 100% interception of high-risk commands with 14.5% overhead. This addresses filesystem safety but not network exfiltration, credential management, or multi-tool isolation.

NVIDIA Safety Framework [Ghosh et al., 2025] presents an operational risk taxonomy with auxiliary AI models for contextual risk management, validated against 10,000+ attack-/defense traces. Their focus is risk classification and dynamic response; ours is infrastructure enforcement.

The Survey of Agentic AI and Cybersecurity [Multiple Authors, 2026] provides critical empirical validation: "over 75% of malicious commands execute successfully without sandboxing, while container-based sandboxing blocks nearly all such commands."

Table 2 summarizes the feature coverage of related academic systems.

Table 2: Comparative feature coverage of academic agent security systems.

System	Container	Network	Cred. Vault	Audit	Secrets	HITL
IsolateGPT	Per-app	—	—	—	—	—
AgentBound	—	—	—	—	—	—
Fault-Tol. Sandbox	—	—	—	—	—	Partial
NVIDIA Framework	—	—	—	Partial	—	Partial
Fides	Partial	—	—	—	—	—
ceLLMate	—	—	—	—	—	—
MiniScope	—	—	—	—	—	—
Progent	—	—	—	Partial	—	Partial
Ours	**Per-tool**	**Per-tool**	**Vault**	**Full**	**Pattern+entropy**	**Risk-based**

2.5 Agent Isolation and Industry Frameworks

IsolateGPT [Wu et al., 2025] is the closest prior work to ours. Wu et al. propose a hub-and-spoke architecture that transparently isolates LLM-integrated applications, preventing data leakage between apps sharing the same LLM backend. IsolateGPT focuses on *data isolation*—ensuring one app's context cannot contaminate another's—via isolated LLM instances and API proxying. Our work addresses a complementary but distinct threat surface: we assume the LLM itself is trusted (or at least monitored) and focus on constraining what *tools* can do once invoked. We add five defense-in-depth layers absent from IsolateGPT: network egress filtering, credential vaults, secret scanning, audit logging, and human-in-the-loop gates.

Deng et al. [Deng et al., 2024] provide a comprehensive threat taxonomy for AI agents, organizing threats along four dimensions: adversarial inputs, operational environment, untrusted external entities, and system design flaws. Our architecture directly addresses their "operational environment" and "untrusted external entities" categories through infrastructure-level enforcement, providing a concrete implementation for the defense strategies they identify as needed.

Industry frameworks. Several industry efforts address agent security at the framework or platform level. MAESTRO [Cloud Security Alliance, 2025] (Cloud Security Alliance) provides a multi-layer threat taxonomy for agentic AI but does not prescribe infrastructure enforcement.

Díaz, Kern, and Olive [Díaz et al., 2025] (Google) outline secure agent design principles including sandboxing and least privilege but stop short of a reference implementation. SAFE-MCP [Linux Foundation, 2025] (Linux Foundation) defines an enterprise security assurance framework for MCP deployments. E2B [E2B, 2025] provides Firecracker-based microVM sandboxing with sub-second cold starts, and Daytona [Daytona, 2025] offers sub-90 ms cold start secure environments. These efforts focus on threat modeling and principles (MAESTRO, Google, SAFE-MCP) or single-layer isolation (E2B, Daytona); our work provides the multi-layer infrastructure implementation combining isolation with network filtering, credential management, audit, secret scanning, and human oversight.

2.6 Capability-Based Security

The capability model originates with Dennis & Van Horn [Dennis and Van Horn, 1966], who proposed unforgeable tokens granting access to specific resources rather than relying on ambient authority. Saltzer & Schroeder [Saltzer and Schroeder, 1975] formalized the principle of least privilege and complete mediation. seL4 [Klein et al., 2009] provided the first formally verified capability-based microkernel. Capsicum [Watson et al., 2010] brought capability mode to FreeBSD. CHERI [Watson et al., 2015] extends capability enforcement to hardware. Miller [Miller, 2006] adapted capabilities to programming language semantics, influencing WASI and Cloudflare Workers.

Our pattern adapts the capability model to the agent-tool boundary: each container receives explicit, scoped capabilities (filesystem mounts, network rules, injected credentials) rather than inheriting the host's ambient access. The MCP Gateway serves as the capability enforcement point.

2.7 Gap in Existing Work

Commercial MCP Gateways. We surveyed 14 commercial and open-source MCP gateway products[1]. The dominant pattern covers authentication/authorization (OAuth 2.1, OIDC, SAML) and audit logging well. However: only Docker MCP Gateway offers container-per-tool isolation; network egress filtering per tool is absent from all but Azure (VNet-level, not per-tool); secret scanning of tool responses is nascent; and no single product combines all six defense layers. The OWASP MCP Top 10 [OWASP, 2025] ranks "Token Mismanagement and Secret Exposure" as the #1 risk.

Prompt Injection and HITL Limitations. Indirect prompt injection [Greshake et al., 2023] poses a fundamental challenge: InjecAgent [Zhan et al., 2024] showed 24% of agents follow injected instructions; AgentDojo [Debenedetti et al., 2024] found even the best defenses fail against adaptive attacks; Lies-in-the-Loop [Checkmarx, 2025] demonstrated prompt injection can manipulate HITL approval dialogs; and Zhan et al. [Zhan et al., 2025] broke all 8 tested prompt-level defenses with >50% success rate.

No existing framework, protocol, commercial product, or academic work provides a complete infrastructure-level security architecture for agent tool execution that combines all six defense layers. This is the gap we address.

3 Threat Model

3.1 Attacker Model

We consider three classes of adversaries:

[1] Acuvity, Arcade AI, Azure MCP Gateway, Docker MCP Gateway, Envoy AI Gateway, Lasso Security, MCP Gateway (OSS), Prompt Security, Solo.io, Speakeasy, and four others; full methodology in the project repository.

A1: Malicious External Input (Prompt Injection). The attacker controls content the agent processes (e.g., web pages, emails, GitHub issues). Goal: trick the agent into executing unintended tool calls. Example: GitHub MCP breach (May 2025)—prompt injection via public issues hijacked agent to extract private repo data.

A2: Compromised or Malicious Tool (Supply Chain). The attacker controls an MCP server the agent connects to. Goal: exfiltrate data, execute arbitrary code on host, pivot to other tools. Example: Smithery registry vulnerability (disclosed Oct 2025)—path traversal could have exposed Docker credentials for 3,000+ servers; no evidence of exploitation [GitGuardian, 2025, AuthZed, 2025].

A3: Insider Threat (Over-Privileged Agent). A legitimate agent with excessive permissions. Goal: unintended data access, credential leakage, destructive operations. Example: Supabase Cursor agent processed support tickets with service-role access, enabling SQL injection.

3.2 Threats Addressed

Table 3 maps threats to defense layers and real-world breaches.

Table 3: Threats addressed by the Capability-Container Pattern.

ID	Threat	Defense Layer	Breach Prevented
T1	Malicious shell commands	Container + HITL	MCP Inspector (Jun '25)
T2	Credential leakage in output	Secret scanning + redaction	GitHub MCP (May '25)
T3	Network data exfiltration	Per-container egress filter	WhatsApp MCP (Apr '25)
T4	Prompt injection → misuse	Risk classification + HITL	Supabase Cursor
T5	Unauthorized credential access	Vault-based injection	mcp-remote (Jul '25)
T6	Lateral movement	Separate containers	Smithery (Oct '25)
T7	Audit trail tampering	Append-only SQLite + WAL	—
T8	Supply chain compromise	Container + network policy	Postmark (Sep '25)

3.3 Threats Not Addressed (v1)

- **T9:** Compromised host OS—we assume a trusted host
- **T10:** Container image supply chain—image signing planned for v2
- **T11:** Side-channel attacks on shared hardware
- **T12:** Sophisticated ML-based evasion of content filters
- **T13:** Multi-agent coordination attacks—future work

3.4 Security Properties

P1 (Least Privilege): Each container receives only the capabilities required for its tool—filesystem mount points, network allowlist, and injected credentials.

P2 (Mediated Access): No direct agent-to-tool communication. All requests flow through the gateway, enabling policy enforcement, audit, secret scanning, and human approval.

P3 (Defense in Depth): Six independent layers, each providing value if others fail: container isolation, network filtering, credential vaults, audit logging, secret scanning, and HITL gates.

P4 (Fail-Secure): Network defaults to block; HITL auto-denies on timeout (60 s); containers run non-root with minimal capabilities.

4 The Capability-Container Pattern

4.1 Pattern Overview

The Capability-Container Pattern is an architectural pattern for securing autonomous AI agent tool execution. Its core invariant:

> *An agent NEVER directly accesses a tool. All tool invocations flow through a mediation gateway into isolated containers, each provisioned with only the capabilities required for that specific tool.*

4.2 Architecture

Figure 1 illustrates the system architecture. The agent container communicates exclusively with the MCP Gateway via JSON-RPC 2.0. The gateway routes validated requests to per-tool capability containers, each configured with scoped filesystem mounts, network policies, and runtime-injected credentials. A credential vault provides dynamic secrets, and an append-only audit database records all operations with automatic secret redaction.

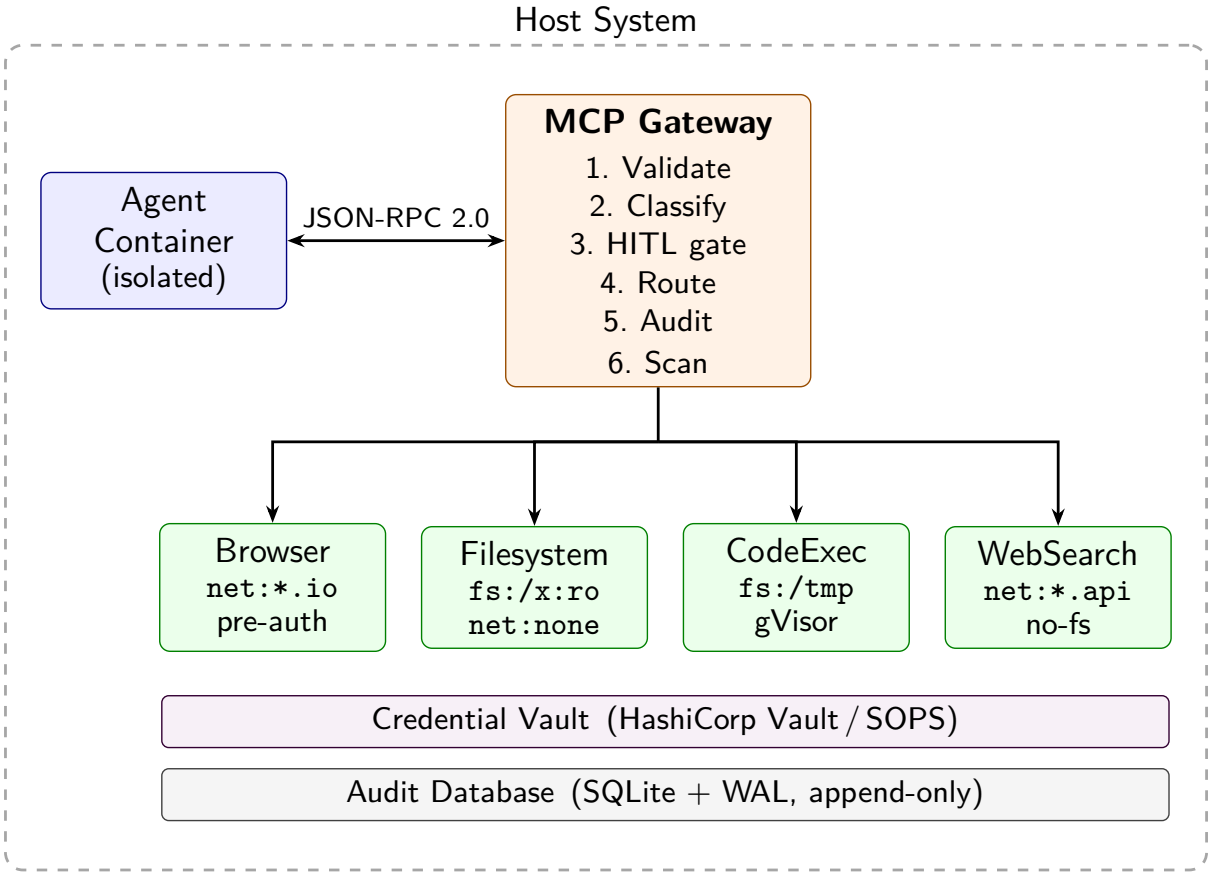

Figure 1: Capability-Container Pattern architecture. Each tool runs in an isolated container with scoped capabilities. All communication flows through the MCP Gateway.

4.3 Components

Agent Container. Runs the ReAct loop and LLM inference. Only network access: the MCP Gateway endpoint (`127.0.0.1:8100`). Cannot reach host filesystem, other containers, or external network.

MCP Gateway. The single enforcement point, executing the pipeline in Algorithm 1 for every tool invocation.

Capability Containers. Each tool runs in a Docker container configured with explicit filesystem mounts (ro/rw), domain allowlist with default-deny, environment-variable credentials cleaned on stop, CPU/memory/PID resource limits, non-root execution (UID 1000), and optional gVisor runtime (∼70 syscalls vs. ∼300).

Credential Vault. Production: HashiCorp Vault with AppRole auth, dynamic secrets, auto-renewal. Team: SOPS with age/GPG encryption. Development: environment variable fallback. Credentials are never stored in configuration files, code, or container images.

4.4 Gateway Pipeline

4.5 Security Properties (Design Goals)

The following properties are design goals enforced through infrastructure configuration. They have not been formally verified through model checking or theorem proving; formal verification is listed as future work (Section 8). We provide enforcement rationale for each property.

Property 1 (Isolation): For any two capability containers C_i and C_j, there exists no communication channel between them except through the gateway.

Enforcement rationale: Docker network namespaces with `--network=none` for isolated tools (filesystem, code-exec) and dedicated bridge network for networked tools. Inter-container routing blocked by iptables rules; only the gateway can reach tool containers. Validated by network policy tests (97/98 correct classifications, 99.0% accuracy).

Property 2 (Least Privilege): Each container C_i is provisioned with capabilities $\text{Cap}(C_i) \subseteq \text{Cap}_{\text{total}}$, where $\text{Cap}(C_i)$ is the minimum set required for tool T_i.

Enforcement rationale: Per-container capability manifests specifying filesystem mounts (ro/rw), network policies (domain allowlist), and credential references. Containers run as UID 1000 with `--cap-drop=ALL` and only tool-specific capabilities added back. Resource limits (CPU, memory, PIDs) via cgroups.

Property 3 (Complete Mediation): Every tool invocation passes through the gateway. There is no bypass path from agent to tool.

Enforcement rationale: The agent container's only network access is `127.0.0.1:8100` (the gateway). Tool containers have no ingress from agent. The gateway's ROUTE_TABLE maps tool names to container endpoints; unregistered tools return UNKNOWN_TOOL. *Caveat:* a container escape (T9 in threat model) would bypass this property, which is why we recommend gVisor (reduced syscall surface from ∼300 to ∼70).

Property 4 (Audit Completeness): For every tool invocation I, there exists an audit record $A(I)$ containing the request, response, risk classification, and approval decision.

Enforcement rationale: SQLite with WAL mode, append-only schema. LogAudit() is called on all code paths in Algorithm 1 (lines 9, 19, 26, 28—covering deny, egress block, alert, and success). Measured throughput: 2,036 ops/sec sustained over 1,000 events.

Property 5 (Fail-Secure): On timeout, connection failure, or ambiguous state, the system denies the operation rather than allowing it.

Enforcement rationale: HITL auto-deny on timeout (configurable, default 60 s). HTTP POST to tool containers uses 30 s timeout. Network policy defaults to `block_by_default=True`. Unknown tools return error, not passthrough.

5 Implementation: Harombe

Harombe is an open-source reference implementation of the Capability-Container Pattern, written in Python 3.11+. It implements all six defense-in-depth layers described in Section 4.

Algorithm 1 MCP Gateway Request Pipeline

Require: *req*: JSON-RPC 2.0 request from agent
Ensure: *res*: sanitized response or denial

```
VALIDATE(req) or return error(INVALID_REQUEST)
tool ← req.params.name
args ← req.params.arguments
cid ← UUID()                                              ▷ correlation ID

risk ← CLASSIFYRISK(tool, args)                           ▷ → {LOW, MED, HIGH, CRIT}

if risk ≥ HITL_THRESHOLD then
    decision ← AWAITAPPROVAL(tool, args, risk, τ=60s)
    if decision ≠ APPROVED then
        LOGAUDIT(cid, req, DENIED, risk)
        return error(DENIED_BY_HITL)
    end if
end if

endpoint ← ROUTE_TABLE[tool]
if endpoint = ∅ then return error(UNKNOWN_TOOL)
end if

creds ← VAULT.FETCH(tool.credential_ref)
INJECTENV(endpoint.container, creds)

if EGRESSFILTER(endpoint, args) = BLOCKED then
    LOGAUDIT(cid, req, BLOCKED_EGRESS)
    return error(EGRESS_DENIED)
end if

raw_res ← HTTPPOST(endpoint, args, τ=30s)

secrets ← SCANSECRETS(raw_res)
if secrets ≠ ∅ then
    raw_res ← REDACT(raw_res, secrets)
    LOGALERT(cid, secrets)
end if

LOGAUDIT(cid, req, raw_res, risk, secrets)
return raw_res
```

5.1 Layer 1: Container Isolation

Each tool runs in a Docker container with enforced resource limits:

Listing 1: Container resource limit configuration.

```python
@dataclass
class ResourceLimits:
    cpu_period: int = 100_000        # microseconds
    cpu_quota: int = 50_000          # 50% of one core
    memory_limit: str = "256m"       # hard limit
    memory_reservation: str = "128m" # soft limit
    pids_limit: int = 100            # max processes
```

Containers are created with non-root execution (UID 1000), read-only root filesystem (tmpfs for `/tmp`), explicit bind mounts only, and health monitoring via periodic `/health` endpoint checks.

5.2 Layer 2: Network Egress Filtering

Listing 2: Network policy configuration.

```python
@dataclass
class NetworkPolicy:
    allowed_domains: list[str]    # e.g., ["*.github.com"]
    allowed_ips: list[str]        # static IPs
    allowed_cidrs: list[str]      # e.g., ["10.0.0.0/8"]
    block_by_default: bool = True # fail-secure
    allow_dns: bool = True        # port 53
    allow_localhost: bool = False  # no loopback by default
```

Features include domain allowlisting with wildcard patterns, CIDR validation, private IP blocking (RFC 1918 ranges) to prevent SSRF, suspicious pattern detection (port scanning, DNS tunneling), and dynamic policy updates without container restart.

5.3 Layer 3: Credential Management

Pluggable backend architecture: HashiCorp Vault (AppRole auth, dynamic secrets), SOPS (encrypted files with age/GPG), or environment variable fallback. The agent never sees raw credentials—the gateway handles all credential lifecycle.

5.4 Layer 4: Audit Logging

SQLite with WAL mode achieves <1 ms writes (0.56 ms measured), 2,036 ops/sec sustained throughput, and automatic redaction of API keys, passwords, JWT tokens, credit card numbers, and private keys.

5.5 Layer 5: Secret Scanning

Three detection methods: (1) pattern matching with type-specific regexes (confidence: 0.95), (2) prefix detection for known secret formats (confidence: 0.85), and (3) Shannon entropy analysis (threshold: ≥ 3.5 bits/character). Overlapping matches keep the highest-confidence detection.

5.6 Layer 6: Human-in-the-Loop (HITL) Gates

The HITL subsystem comprises a risk classifier (LOW/MEDIUM/HIGH/CRITICAL), an approval workflow with configurable timeout (default 60 s auto-deny), a rules-based auto-approval engine, a historical trust manager, and a context-aware decision engine.

5.7 Browser Security

A novel contribution: pre-authenticated browser automation where credentials are fetched from the vault, injected into the browser context, and the agent receives a handle but never sees raw credentials. The system includes 16 risk classification rules for browser actions and accessibility-based interaction (semantic tree, not raw DOM).

5.8 Implementation Complexity

Table 4 summarizes code size by defense layer.

Table 4: Implementation complexity by defense layer (lines of code, non-blank, non-comment).

Layer	Component	LOC
L1: Container Isolation	`docker_manager.py`	351
L2: Network Egress	`network.py`	853
L3: Credential Mgmt	`vault.py` + `injection.py`	661
L4: Audit Logging	`audit_logger.py` + `audit_db.py`	1,193
L5: Secret Scanning	`secrets.py`	365
L6: HITL Gates	`hitl/` (6 modules)	1,466
Browser Security	`browser_*.py`	569
Gateway	`gateway.py`	503
Total		**5,961**

Each concrete tool averages ∼131 LOC. The security subsystem uses 27 core Python dependencies. Adding a new tool requires a Python tool class (∼50–130 LOC), a Dockerfile, and a routing entry.

6 Evaluation & Benchmarks

6.1 Research Questions

RQ1 What is the performance overhead of the Capability-Container Pattern?
RQ2 How effective is each defense layer at preventing known attack classes?
RQ3 Would the pattern have prevented real-world MCP breaches?

6.2 Experimental Setup

Environment: Darwin 25.2.0 arm64, Python 3.14.3 (pre-release; final release expected Oct 2026), scipy 1.17.0, scikit-learn 1.8.0, detect-secrets 1.5.0. **Statistical methods:** timing CIs via Student's t-distribution (df = $n-1$) with IQR-based outlier filtering (factor = 1.5); detection rates via Clopper–Pearson exact binomial 95% CIs; 50 warmup iterations before each measurement loop; GC disabled during measurement. **Date:** 2026-02-11.

6.2.1 Evaluation Methodology and Scope

All benchmarks use **synthetic test data created alongside the implementation**. The 1,534-sample corpus was constructed to exercise specific regex patterns and detection heuris-

tics, not sampled from production environments. Breach scenarios are **reconstructed from public incident reports** (Invariant Labs, Checkmarx, Orca Security, Trail of Bits), not replays of actual attack traffic. No external red team or third-party adversarial testing was conducted. Results represent **defense coverage mapping**—demonstrating which attack classes each security layer addresses—not adversarial robustness guarantees.

6.2.2 Benchmark Infrastructure

All benchmarks exercise the real security modules directly (no Docker, no mocking). Performance benchmarks measure **gateway logic overhead only**—the time spent in security module Python code paths. They do not include container-level overhead (Docker image pull, container creation, health check, HTTP round-trip). For cross-run reproducibility, `benchmarks/run_multi.py` invokes the suite N times independently and computes cross-run 95% CIs using the t-distribution over per-run means.

6.2.3 Statistical Methodology Notes

Large sample sizes ($n \geq 200$) in performance benchmarks justify the Central Limit Theorem for t-based confidence intervals without formal normality testing. IQR-based outlier filtering (factor = 1.5) removes measurement artifacts (GC pauses, OS scheduling jitter); we acknowledge this may underestimate tail latency for workloads sensitive to worst-case performance. No family-wise error correction (e.g., Bonferroni) is applied across multiple metrics; readers should interpret individual CIs accordingly. Independence is justified by separate function calls with fresh inputs and separate process invocations for cross-run measurements.

6.3 Performance Benchmarks (RQ1)

6.3.1 Secret Scanning Latency by Message Size

Table 5: Secret scanning latency by message size ($n = 200$ per size, IQR-filtered).

Size (chars)	Mean (ms)	Filtered (ms)	95% CI	p95 (ms)	Outliers
100	0.008	0.008	[0.008, 0.008]	0.008	26
500	0.030	0.029	[0.029, 0.029]	0.030	44
1,000	0.057	0.056	[0.056, 0.056]	0.061	11
5,000	0.317	0.317	[0.312, 0.321]	0.371	1
10,000	0.553	0.553	[0.552, 0.555]	0.573	14

Sub-millisecond scanning up to 10K characters. Linear scaling ($\sim$0.055 ms per 1K chars) confirms $O(n)$ regex-based approach.

6.3.2 Full Pipeline Overhead

Baseline comparison: bare operations (no security) take 0.000384 ms; the full gateway logic pipeline takes 0.023 ms—an absolute overhead of $\sim$0.024 ms, or **0.005% of a typical 500 ms LLM inference call**. Note: this measures Python security module code paths only; production deployments add container-level latency (creation, HTTP round-trip, health check).

6.4 Detection Effectiveness (RQ2)

6.4.1 Secret Scanner True Positive Rate

Corpus: 900 synthetic secrets (100 per type, 9 types), conforming to production regex patterns. 20 hand-crafted samples per type serve as baseline; 80 additional per type generated

Table 6: Gateway logic overhead—all security layers combined ($n = 200$, 25 outliers removed). Does not include container-level latency.

Metric	Value
Mean	0.024 ms
Filtered mean	0.023 ms [0.023, 0.023]
p50	0.023 ms
p95	0.024 ms
p99	0.024 ms

programmatically via deterministic SHA-256 seeded hashing (Table 7).

Table 7: Secret scanner true positive rate by type ($n = 100$ per type, Clopper–Pearson CIs).

Secret Type	Samples	Detected	TP Rate	95% CI
AWS keys	100	100	100%	[0.964, 1.000]
GitHub tokens	100	100	100%	[0.964, 1.000]
Slack tokens	100	100	100%	[0.964, 1.000]
Stripe keys	100	100	100%	[0.964, 1.000]
JWT tokens	100	100	100%	[0.964, 1.000]
Private keys	100	100	100%	[0.964, 1.000]
Database URLs	100	100	100%	[0.964, 1.000]
Passwords	100	100	100%	[0.964, 1.000]
API keys	100	100	100%	[0.964, 1.000]
Overall	**900**	**900**	**100%**	**[0.996, 1.000]**

6.4.2 Secret Scanner False Positive Rate

Corpus: 514 benign samples across 7 categories (Table 8). The single false positive was a placeholder token (`ghp_xxxx...`) matching the GitHub token pattern—arguably a true positive since the pattern is structurally valid.

Table 8: Secret scanner false positive rate by category (Clopper–Pearson CIs).

Category	Samples	FPs	FP Rate	95% CI
Prose	80	0	0.0%	[0.000, 0.045]
Code	70	0	0.0%	[0.000, 0.051]
Logs	70	0	0.0%	[0.000, 0.051]
Config	71	0	0.0%	[0.000, 0.051]
URLs	70	0	0.0%	[0.000, 0.051]
Hashes	71	0	0.0%	[0.000, 0.051]
Edge cases	82	1	1.2%	[0.000, 0.066]
Overall	**514**	**1**	**0.19%**	**[0.000, 0.011]**

6.4.3 Comparative Detection: Harombe vs. detect-secrets

Both tools evaluated on the same 1,414-sample corpus (Table 9).

Harombe achieves a 30× reduction in false positives while maintaining perfect recall.

Table 9: Comparative detection: Harombe vs. detect-secrets 1.5.0 (1,414 samples).

Metric	Harombe	detect-secrets
True Positives	900	900
False Positives	17	512
True Negatives	497	2
False Negatives	0	0
Precision	**0.981**	0.637
Recall	1.000	1.000
F_1 Score	**0.991**	0.779

Table 10: Adversarial secret detection robustness (120 samples, 30 per technique, Clopper–Pearson CIs).

Evasion Technique	n	Detected	Rate	95% CI
Multiline context (YAML/env/Docker/CI)	30	30	100%	[0.884, 1.000]
Split secrets (multi-line/variable)	30	28	93.3%	[0.779, 0.992]
Unicode homoglyphs (Cyrillic/Greek/ZWSP)	30	10	33.3%	[0.173, 0.528]
Base64-encoded (single/double/nested)	30	1	3.3%	[0.001, 0.172]

6.4.4 Adversarial Secret Detection

6.4.5 Risk Classification

50 test cases across 4 risk levels yield a perfect confusion matrix (50/50 accuracy). Unknown tools default to MEDIUM (conservative).

6.5 Breach Prevention Analysis (RQ3)

Each breach scenario was reconstructed from public incident reports and tested against the defense layers (Table 11). Attack payloads are synthetic reconstructions, not replays of actual attack traffic.

Table 11: Breach scenario coverage: 2025 MCP attack patterns reconstructed against Harombe.

ID	Attack Scenario	Blocked?	Layers Activated
T1	WhatsApp exfiltration	**Yes**	scanner, egress, HITL
T2	GitHub credential leak	**Yes**	scanner
T3	mcp-remote cmd injection	**Yes**	classifier, HITL
T4	Filesystem path traversal	No*	classifier
T5	Smithery Docker cred theft	**Yes**	scanner, redactor
T6	Postmark BCC injection	**Yes**	HITL, redactor
T7	Multi-layer compound	**Yes**	all 5 layers

*Path traversal prevention relies on L1 container bind mounts, not risk classification.

6/7 attack scenarios addressed at the security module level. T7 demonstrates defense-in-depth: all 5 independent layers activated simultaneously. These results reflect defense *coverage*—which layers activate for which attack classes—not adversarial resilience against adaptive attackers.

6.6 Layer-by-Layer Coverage

Every attack vector is covered by at least 2 layers. No single layer failure results in undetected compromise.

Table 12: Defense layer coverage by attack vector. •• = primary defense, • = detection/forensics.

Attack Vector	L1	L2	L3	L4	L5	L6
Malicious shell commands	••			•		••
Credential leakage			••	•	••	
Network exfiltration	•	••		•		
Prompt injection → misuse				•		••
Unauthorized cred access	••		••	•		
Lateral movement	••	••		•		
Supply chain compromise	••	••		•	••	

6.7 Cross-Run Reproducibility

We ran the full benchmark suite 5 independent times. Key cross-run results in Table 13.

Table 13: Cross-run reproducibility (5 runs, 31 metrics, t-distribution CIs).

Metric	Mean (ms)	StdDev	95% CI
Full pipeline	0.025	0.002	[0.022, 0.028]
Audit write	0.523	0.034	[0.481, 0.565]
Secret scan (100 chars)	0.008	0.001	[0.007, 0.009]
Secret scan (1K chars)	0.057	0.001	[0.055, 0.058]
Secret scan (10K chars)	0.556	0.023	[0.528, 0.584]
Secret redaction (e2e)	0.094	0.005	[0.088, 0.099]
Egress (blocked domain)	13.918	0.843	[12.871, 14.964]

All security-critical paths show tight cross-run CIs. The blocked domain egress check shows the highest variance (13.9 ms ± 0.8 ms) due to DNS resolution timing.

6.8 Limitations of Evaluation

- **Self-benchmarked evaluation:** all benchmarks use synthetic test data created alongside the implementation. No external red team or third-party adversarial testing was conducted. Results represent defense coverage, not adversarial robustness guarantees.
- Benchmarks run on a single machine (Apple Silicon), not production deployment.
- Secret corpus (900 TP + 514 benign = 1,414 samples) uses synthetic secrets; real-world FP rates may differ with diverse codebases.
- **Gateway logic overhead** (0.025 ms) measures Python security module code paths only. Production overhead includes container creation, HTTP round-trip, and network latency.
- Base64-encoded secrets (3.3% detection) and Unicode homoglyphs (33.3%) evade regex-based scanning—consistent with industry limitations [Basak et al., 2023, Orca Security, 2024].
- Performance under high concurrency (100+ agents) untested.
- T4 (path traversal) cannot be demonstrated without Docker in the benchmark suite.
- No user study or developer experience evaluation conducted.

7 Discussion & Limitations

7.1 Why Infrastructure Over Protocol

The recurring pattern in MCP breaches is clear: protocol-level trust is insufficient. This parallels the evolution of web security—HTTP alone cannot prevent XSS, CSRF, or injection attacks;

infrastructure controls are required. Similarly, MCP alone cannot prevent tool poisoning, credential leakage, or data exfiltration.

*The protocol tells you **how** to talk to tools. The infrastructure determines **what tools can do**.*

7.2 Trade-offs

Performance vs. Security. Container isolation adds latency. Our benchmarks show <5 ms for warm containers, but cold starts remain a concern. Mitigation: pre-warming, connection pooling, health check-based readiness probes.

Operational Complexity. Docker is a hard production dependency. Mitigation: development mode runs without containers; production automated via Docker Compose or Kubernetes.

Flexibility vs. Control. Strict egress filtering may break tools with dynamic network needs. Mitigation: wildcard support, dynamic policy updates, HITL approval for non-allowlisted domains.

7.3 Limitations

Host OS Trust Assumption. If an attacker has root on the host, container isolation provides no protection. Future work: hardware-enforced isolation (Intel SGX, AMD SEV-SNP).

Container Escape. Recent CVEs underscore this risk: CVE-2024-1086 (99.4% success on KernelCTF images [Notselwyn, 2024]), CVE-2025-23266 (CVSS 9.0), and three runc escapes in November 2025. Lin et al. [Lin et al., 2018] found 56.82% of exploits succeed against default Docker. gVisor mitigates this [Jang et al., 2022] but introduces compatibility issues. This reinforces defense-in-depth.

Adversarial Evasion. Regex-based scanning is vulnerable to encoding-based evasion: base64-encoded secrets detected at only 3.3% (CI: [0.001, 0.172]) and Unicode homoglyphs at 33.3% (CI: [0.173, 0.528]). This is consistent with industry limitations [Basak et al., 2023, Orca Security, 2024, Boucher and Anderson, 2023]. Defense paths: Unicode normalization (NFKC) and ML-based scanning [Basak et al., 2025, Wiz, 2025].

Prompt Injection and HITL Limitations. Lies-in-the-Loop [Checkmarx, 2025] shows prompt injection can manipulate HITL dialogs. Zhan et al. [Zhan et al., 2025] broke all 8 tested prompt-level defenses. Our response: HITL is one of six layers; infrastructure layers operate independently of prompt-level trust.

Supply Chain Risk. DockerDash [Levi and Noma Security, 2025] demonstrated AI-specific container image compromise. Image signing (Sigstore/cosign) is planned.

Single Point of Failure and Compromise. The gateway is both a single point of failure and a single point of compromise: if subverted, all five security properties collapse simultaneously. This is the inverse of the defense-in-depth principle applied to tools. Gateway hardening: minimal API surface (single JSON-RPC endpoint), no tool-specific logic in the gateway code path, non-root execution, single-purpose process. The audit database should ideally write to a separate trust domain (remote syslog, append-only S3) so that audit integrity survives gateway compromise. Future work: replication and hardware-backed attestation.

Scalability. The current implementation targets single-host deployment. Multi-host distributed gateways are future work.

Forward Compatibility with MCP Spec Evolution. The MCP specification is actively evolving. If future versions add native authentication or isolation directives, our pattern adapts naturally: the gateway can consume MCP-level security metadata as additional policy inputs while continuing to enforce infrastructure-level boundaries.

7.4 Ethical Considerations

The Capability-Container Pattern is defensive technology. The same techniques could theoretically sandbox agents for malicious purposes, but we believe the defensive value significantly outweighs this risk given demonstrated real-world harm from unsecured deployments.

7.5 Comparison with MicroVM Approaches

MicroVMs (Firecracker) provide stronger isolation (separate kernel) at the cost of higher resources [Agache et al., 2020]. However, Weissman et al. [Weissman et al., 2023] showed limited Spectre/MDS protection, and Xiao et al. [Xiao et al., 2023] demonstrated exploitable operation forwarding. Our architecture is runtime-agnostic: the same pattern works with Docker, gVisor, or microVMs.

7.6 Comparison with Commercial MCP Gateways

Our survey of 14 commercial and open-source MCP gateway products reveals the market is crowded but shallow on defense-in-depth. Container-per-tool isolation: only Docker MCP Gateway. HITL gates: only 3 products (none combined with containers). Secret scanning: nascent (PII only). Network egress filtering per tool: effectively absent. This gap is structural: existing products evolved from API gateway heritage and lack infrastructure-level security primitives.

8 Conclusion

We presented the Capability-Container Pattern, an infrastructure-level security architecture for autonomous AI agent tool execution. The pattern enforces six defense-in-depth layers ensuring that even compromised tools cannot access the host, leak credentials, exfiltrate data, or interfere with other tools.

8.1 Key Results

- **Defense coverage for 6/7 reconstructed 2025 MCP breach scenarios** (synthetic reconstructions from public incident reports; no external red team)
- $\sim$0.025 **ms gateway logic overhead** (cross-run 95% CI: [0.022, 0.028])—0.005% of a 500 ms LLM inference call
- **100% secret detection rate** (900/900 across 9 types, Clopper–Pearson CI: [0.996, 1.000]) with 0.19% FP rate (1/514, CI: [0.000, 0.011])
- $F_1 = 0.991$ vs. 0.779 for detect-secrets (30$\times$ fewer false positives on 1,414-sample corpus)
- **Sub-microsecond risk classification**
- **2,036 ops/sec audit throughput**
- Transparent adversarial boundaries (120 samples, 30 per technique): 100% for multiline context, 93.3% for split secrets, 3.3% for base64, 33.3% for homoglyphs
- **5,961 LOC** across 8 security layers

8.2 Core Insight

MCP defines *how* agents communicate with tools. It does not and cannot enforce *what tools are allowed to do*. This enforcement must happen at the infrastructure layer—just as web security requires infrastructure controls beyond HTTP.

8.3 Future Work

1. Hardware-enforced isolation (Intel SGX, AMD SEV-SNP)

2. Container image verification (Sigstore/cosign)
3. Multi-host distributed gateway architecture
4. Formal verification of isolation and mediation properties
5. Red team evaluation
6. Multi-agent coordination security
7. Encoding-aware scanning (base64 decoding, Unicode normalization) to address the 3.3%/33.3% adversarial detection gaps
8. Concurrency benchmarks (100+ simultaneous agents)
9. User study and developer experience evaluation

References

Alexandru Agache et al. Firecracker: Lightweight virtualization for serverless applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

Anthropic. Model context protocol specification (draft). https://modelcontextprotocol.io/, 2024.

AuthZed. Timeline of MCP breaches. https://authzed.com/blog/timeline-mcp-breaches, 2025.

Setu Basak et al. What secrets do your repositories tell? A multi-tool comparison study. *arXiv preprint*, 2023.

Setu Basak et al. Fine-tuned LLMs for secret detection. *arXiv preprint*, 2025.

Nicholas Boucher and Ross Anderson. Trojan source: Invisible vulnerabilities. In *USENIX Security Symposium*, 2023.

Lukas Bühler, Matteo Biagiola, Luca Di Grazia, and Guido Salvaneschi. AgentBound: Securing AI agent execution. *arXiv preprint arXiv:2510.21236*, 2025.

Checkmarx. Lies-in-the-loop: Prompt injection attacks on human-in-the-loop systems, 2025.

Cloud Security Alliance. MAESTRO: Multi-agent security taxonomy, risks, and opportunities. https://cloudsecurityalliance.org/research/topics/maestro, 2025.

Daytona. Daytona sandboxes: Sub-90ms cold start secure environments. https://daytona.io/, 2025.

Edoardo Debenedetti et al. AgentDojo: A dynamic environment to evaluate attacks and defenses for LLM agents. *arXiv preprint arXiv:2406.13352*, 2024.

Yifeng Deng et al. AI agents under threat: A survey of key security challenges and future pathways. *ACM Computing Surveys*, 2024.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

Santiago Díaz, Christoph Kern, and Olivia Olive. Designing secure AI agents. Google Security Blog, 2025.

E2B. E2B sandboxes: Secure code execution for AI agents. https://e2b.dev/, 2025. Firecracker-based microVM sandboxing.

Suvam Ghosh, Alexander Simkin, Kyriacos Shiarlis, Arnab Nandi, Yitao Zhao, et al. A safety and security framework for real-world agentic systems. *arXiv preprint arXiv:2511.21990*, 2025.

GitGuardian. Smithery MCP registry vulnerability disclosure. https://blog.gitguardian.com/smithery-mcp-vulnerability/, 2025.

Google. Kubernetes agent sandbox with gVisor, 2025.

Kai Greshake et al. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *AISec Workshop*, 2023.

Yi He et al. Cross-container attacks: The bewildered eBPF on clouds. In *USENIX Security Symposium*, 2023.

Jinwoo Jang et al. Quantitative security analysis of gVisor. In *European Symposium on Research in Computer Security (ESORICS)*, 2022.

Ahmad Jarkas et al. A survey on container security: Attacks, countermeasures, and challenges. *ACM Computing Surveys*, 2025.

Gerwin Klein et al. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

Sasi Levi and Noma Security. DockerDash: Malicious docker image label exploit via Gordon AI. https://noma.security/dockerdash-exploit/, 2025.

Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

Linux Foundation. SAFE-MCP: Security assurance framework for enterprise MCP deployments. https://safemcp.org/, 2025.

Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

Multiple Authors. A survey of agentic AI and cybersecurity. *arXiv preprint arXiv:2601.05293*, 2026.

Notselwyn. CVE-2024-1086: Universal local privilege escalation—nf_tables use-after-free. https://github.com/Notselwyn/CVE-2024-1086, 2024. PoC qualified for Google KernelCTF.

Orca Security. How we found a bypass in GitHub secret scanning via base64 encoding. https://orca.security/resources/blog/github-secret-scanning-bypass-base64/, 2024.

OWASP. MCP top 10 security risks. https://owasp.org/www-project-mcp-top-10/, 2025.

Jacob Reeves et al. An analysis of container security vulnerabilities. In *IEEE Secure Development Conference (SecDev)*, 2021.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

Peter Steinberger. OpenClaw: Open-source personal AI assistant. https://openclaw.ai/, 2025.

Xiaoguang Sun et al. Security namespace: Making Linux security frameworks available to containers. In *USENIX Security Symposium*, 2018.

Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, 2010.

Robert N. M. Watson et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

Zane Weissman et al. Security analysis of Firecracker MicroVM. *arXiv preprint*, 2023.

Wiz. Production secret scanning with fine-tuned Llama-3.2-1b, 2025.

Yi Wu et al. IsolateGPT: Transparent isolation for LLM-integrated applications. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2025.

Jian Xiao et al. Security analysis of MicroVM operation forwarding. In *USENIX Security Symposium*, 2023.

Boyang Yan. Fault-tolerant sandboxing for AI coding agents. *arXiv preprint arXiv:2512.12806*, 2025.

Eric J. Young et al. The true cost of containing: A gVisor case study. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

Qiusi Zhan et al. InjecAgent: Benchmarking indirect prompt injections in tool-integrated LLM agents. *arXiv preprint arXiv:2403.02691*, 2024.

Qiusi Zhan et al. Adaptive prompt injection attacks break all prompt-level defenses. *arXiv preprint*, 2025.