

Mirage: Verifiable Containment Evidence for Mediated AI Agent Sandboxes

Rick Clinton | Independent Researcher
Track 1 - Containment
With Apart Research

Abstract

AI evaluation sandboxes still need package access, but the repository intermediary becomes another path I have to trust. I built Mirage because I wanted to know what happens if I simply don't trust it. Mirage is a Linux experiment and offline verifier that treats the workload and repository mediator as hostile. In the v0.5.2 Ubuntu campaign, a live apt-cacher-ng process delivered the permitted package in 10/10 topology trials while alternate ports, arbitrary destinations, redirect follow-up, and a reverse pivot into management stayed blocked. All 23 controls passed, and 41/41 tests passed with zero skips. Adversarial review also broke the old verifier by rewriting and rehashing a failed result. I removed it as the default compliance path and now require witnessed authentication for compliance. Mirage doesn't prove universal containment. It tests whether I can keep the egress I need without giving a potentially compromised mediator broader authority.

1. Introduction

The July 2026 incident reconstructed by Hugging Face and acknowledged by OpenAI made me question whether "does the sandbox have internet access?" is even the right question. The evaluation agent reportedly exploited the package-registry cache proxy that was already permitted to reach the internet and then used external infrastructure during the intrusion [1,2]. So the problem wasn't simply that somebody forgot to block an undeclared route. The route was supposed to exist. The problem was how much authority was still sitting behind it. If my workload genuinely needs one repository mediator, what happens if I assume that mediator is compromised?

Can I still get exactly what I need through it without also letting it select arbitrary destinations, reach management services, follow redirects outside the path I approved, or turn its upstream connection into a route back into the environment? And if I say that worked, I don't want the answer to depend on somebody trusting me because I ran the experiment. I want the evidence to be checkable outside my laboratory. I built a three-zone Linux experiment where the repository mediator is treated as compromised. Before Mirage gets to say a path was blocked, I first make sure that same path was actually alive and reachable without the containment policy.

Then I run a 23-control profile using Linux isolation, a real repository mediator, signed policy and vulnerability inputs, explicit negative probes, and a witness outside the execution host. Finally, I export the evidence. A reviewer doesn't need access to my laboratory network. They need the evidence, witness receipt, verification keys, assurance inputs, and manifest.

Project lineage

After registering for the Apart Research hackathon, I started two separate containment projects: Mirror World and Moss. Both were started after registration. They weren't pre-existing projects that I brought into the hackathon. Mirror World was the broader experiment. I combined a Linux agent sandbox with capability governance, simulated containment layers, structured execution records, and separate checks over what actually happened.

Moss came at the problem differently. It used bounded identities, subtraction-only restrictions, external supervision, isolated operating-system processes, attested observations, fencing, and reseeding. Both reached functioning experimental containment units. Neither was a production security product.

But building them made one problem very obvious to me. Making a boundary work and proving that the boundary worked are separate problems. I started Mirage on 11 September because I wanted to pull that problem into a smaller experiment. Can I keep a repository mediator useful while treating it as hostile, and can somebody outside my laboratory verify what happened?

2. Related Work

Linux namespaces, gVisor, Firecracker, and other sandbox or container systems already provide isolation between untrusted workloads and the host. NIST SP 800-190 also describes container security risks and mitigations [3–5]. Mirage isn't replacing any of them. What Mirage gives me is a way to ask whether that boundary enforced the policy I declared against the probes I said I would run. in-toto and SLSA deal with a related problem through provenance. They separate an artifact from evidence describing how it was produced [6,7].

I use a similar separation in Mirage, except I apply it to containment. Sigstore and Cosign solve another related problem around software artifacts. They use signatures and attestations and can include transparency-log material or verification bundles that can be checked later [8]. Windows systems such as Sysmon and Event Tracing for Windows already give structured observations of process creation, network connections, file activity, and other operating-system events [9,10]. That gives me telemetry.

It doesn't automatically tell me whether the containment profile I declared passed. Confidential-computing attestation can give something stronger than Mirage currently gives: hardware-rooted evidence that an attester was in a measured state when the evidence was produced [11]. My witness doesn't do that. It authenticates the measured evidence prefix and the closure of the run.

It doesn't independently watch every probe. It doesn't attest the host kernel. From the systems and publications I reviewed, I couldn't find one combining the same three things I'm testing here: Treat the permitted repository mediator itself as compromised.

Prove the prohibited paths were actually reachable before containment. Then export authenticated evidence so somebody can verify the result offline. That's the novelty claim I'm comfortable making. I'm not saying nobody has ever built anything similar.

I'm saying I didn't find this specific combination in the material I reviewed.

3. Methodology

Threat model and claim boundary

I start by treating both the workload and the repository mediator as hostile. I test direct network escape, arbitrary forwarding, alternate ports, redirect follow-up, host credential discovery, protected-file modification, process exhaustion, authorization reuse, management access, a reverse pivot from the approved upstream, TLS hostname substitution, and rebinding to an address outside the allow-list. For Mirage, I trust the host kernel, launcher, recorder, verifier, offline signer, and the witness identity declared before execution. Kernel, hypervisor, hardware, host-administrator, signer, verifier, witness, and malicious-recorder compromise are outside the demonstrated threat model.

That is the boundary of the claim.

Experiment topology

Every topology trial starts fresh. Mirage creates three Linux network namespaces connected by two veth pairs. One contains the workload. One contains the repository mediator.

One contains the approved upstream. Inside the proxy namespace, I run a real Ubuntu apt-cacher-ng 3.7.5-1.1 process. I use nftables default-drop input and output chains to restrict the network. The workload gets one declared path: TCP access to the proxy on port 18080.

The proxy gets its own narrow path: it can reach only the upstream address and ports required for that trial. Everything else is dropped by default. After the trial, I remove the services, interfaces, and namespaces. Then I construct the next topology trial again instead of carrying state over from the previous one.

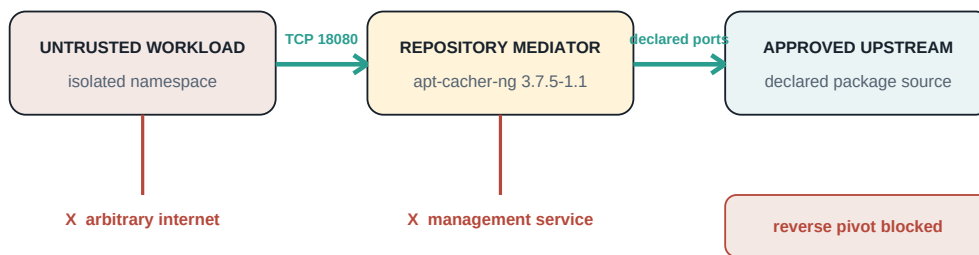


Figure 1. Mirage three-zone containment architecture. Untrusted workload → apt-cacher-ng mediator → approved upstream. The diagram also shows the blocked paths from the workload and mediator to arbitrary internet destinations and the management service.

Matched probes

The workload needs to retrieve one package through the repository proxy. Then I test what else it can do. From the workload side, Mirage tests direct network access, host-file access, credential discovery, process exhaustion, and authorization reuse. The proxy is allowed to reach the declared upstream ports.

Against that, I test arbitrary TCP destinations, an alternate upstream port, redirect targets, and oversized responses. I also let it attempt a reverse connection into a live management service. But I don't want Mirage to call something blocked just because the destination happened to be dead. Before Mirage claims that a prohibited path was blocked, I first prove that the destination is alive and that the path actually works.

Then I apply the policy and test those same paths again. If the service was already unreachable before containment, its later failure tells me nothing about the boundary.

Evidence format and closure

Mirage writes each run as canonical, hash-linked JSONL. A complete stream has exactly one initial record and one terminal record. If records are missing, duplicated, reordered, malformed, truncated, or oversized, verification fails. Originally, I thought the hash chain gave me more than it actually did.

It didn't. A hash chain can tell me that the records are internally consistent. It doesn't tell me who created that history. If somebody had write access to the evidence, they could take a genuine failed result, rewrite it, and recompute the entire unkeyed chain.

That meant my old verification path could be fooled. So in v0.5.2 I separated integrity checking from authenticated compliance. There is no default verification profile anymore. The minimum profile is only available through `--integrity-only`.

It checks the evidence structure and hash chain. It does not claim that the evidence is authentic. Authenticated compliance requires a witness receipt and the fingerprint of a witness trusted before verification. The witness is declared before execution and runs across a separate host trust boundary.

It signs the complete measured evidence prefix before the terminal record closes the stream and reports whether it received the escalation signal. That authenticates the measured prefix and closure. It still doesn't mean the witness independently saw everything that happened before closure.

Assurance inputs

The assured-egress profile adds signed policy and product evidence. An offline Ed25519 key signs two expiring envelopes. One contains the expected network-policy digest. The other contains a pinned response from the NIST National Vulnerability Database CVE API [12].

Then Mirage checks those declarations against the live environment. It recomputes the runtime policy digest and measures the running proxy's package version, executable digest, configuration digest, and endpoint.

Why the workload is synthetic

I use controlled probes in this campaign instead of a live frontier model. I did that intentionally. If the model changes its behaviour between runs, it becomes harder for me to know whether a difference came from the containment boundary or from the model itself. These results don't tell me how a frontier model would search for an escape route I never encoded as a probe.

I'm also not measuring deception, planning, persistence, or adaptive attack behaviour from a live model. That still needs to be tested.

Reproduction procedure

I recorded the SHA-256 digest of the v0.5.2 release archive. For the complete v0.5.2 rerun, I verified the archive on the Ubuntu containment host. I ran all 41 tests with zero skips. I started a live witness across a separate host trust boundary.

Then I ran the complete assured-egress profile from the released archive. The campaign included:

- 100 matched base trials.
- 10 newly constructed three-zone topology trials.
- A real apt-cacher-ng process.
- A live witness receipt.
- Signed policy and vulnerability inputs.
- Offline CLI verification.

I preserved the JSONL evidence, results, witness receipt, environment record, assurance inputs, verifier output, and SHA-256 manifest. This is a complete first-party rerun of the v0.5.2 assured-egress campaign. It is not independent reproduction. I still need an external operator to reproduce the same release and authenticated profile before I can call the result independently confirmed.

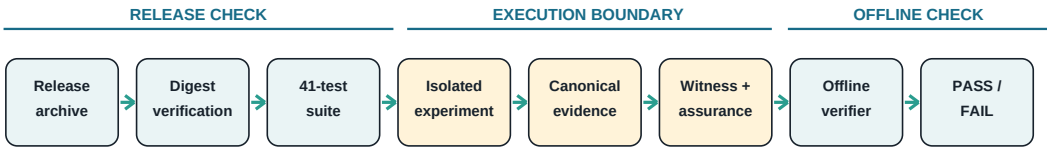


Figure 2. Mirage execution and verification pipeline. Release archive → digest verification → test suite → isolated experiment → canonical evidence → witness receipt → signed assurance checks → offline verifier → PASS/FAIL. The execution boundary and the later offline-verification boundary are separate.

4. Results

The v0.5.2 suite passed 41/41 tests with zero skipped. The permanent full-chain-forgery regression passed too. The old attack—rewrite a failed result, recompute the complete hash chain, and get an authenticated PASS—doesn't work through the current authenticated profile. I ran 100 matched base trials and 10 fresh three-zone topology trials.

The real apt-cacher-ng 3.7.5-1.1 process delivered the permitted package in 10/10 topology trials. All required controls C1–C23 passed. For authorization contention, I ran 5,000 attempts. They produced 100 effects, with one effect per grant.

Benign completion was 100/100. For this deliberately narrow workload, the observed false-positive rate was 0%. The live witness received its signal in 14.787 milliseconds on the host-only test network. Mean wall time was 21.393 milliseconds for the control condition and 23.979 milliseconds for the isolated condition.

The measured addition was 2.586 milliseconds, or 12.09%, in this launch-dominated benchmark. Those numbers describe this experiment. I'm not using them to claim a universal containment failure probability.

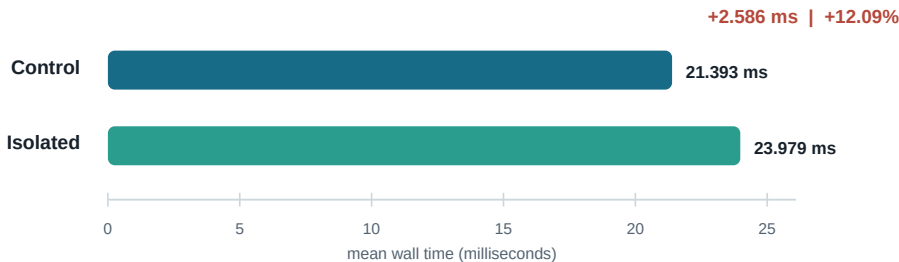


Figure 3. Matched timing results. Control: 21.393 ms; Isolated: 23.979 ms; Added time: 2.586 ms; Relative increase: 12.09%.

Containment result

The real proxy delivered the permitted package in every topology trial. At the same time, the containment policy blocked the alternate upstream port, arbitrary external TCP destinations, redirect follow-up, and the reverse pivot from the upstream into management. The approved TLS service identity was accepted. A mismatched TLS identity was rejected.

A rebound address outside the allow-list remained unreachable [13]. So in this topology, I could keep the package path I needed without giving the repository mediator broad network authority. That is the containment result I'm claiming.

Evidence and assurance result

The signed expected-policy digest matched the policy Mirage constructed at runtime. Both assurance signatures verified. The measured component was apt-cacher-ng 3.7.5-1.1. The witness receipt verified against the identity declared before execution.

The pinned NVD response contained six apt-cacher-ng records. I manually reviewed their applicability and found no returned record naming version 3.7.5-1.1 as affected. It doesn't mean the product had no vulnerability. The data could be incomplete.

I could have made a mistake during the manual review. And an unknown vulnerability obviously wouldn't appear in the snapshot.

What failed during development

The first filesystem campaign used a marker under `/tmp`. That was a bad control because `/tmp` was intentionally writable scratch storage. So I moved the marker to a protected persistent path, narrowed the control, rebuilt the archive, and restarted the campaign. The original direct-egress control modeled the wrong incident condition.

The Unix-socket mediator didn't demonstrate selective TCP routing. The synthetic product and vulnerability fixtures were too weak for the claims I wanted to make. So I replaced them. Then the verifier broke.

That was the biggest problem. If somebody had write access, they could take a genuine failed result, rewrite it, recompute every unkeyed hash, and make the old default verifier return PASS. The evidence was internally consistent. It just wasn't authentic.

So in v0.5.2 I removed the unsafe default. The CLI now requires an explicit profile. Unauthenticated checking only exists through `--integrity-only`. Authenticated profiles require witnessed evidence.

And I kept the exact attack that broke the earlier verifier as a permanent regression test.

Main threat to validity

The recorder can invent an internally consistent history before committing the terminal digest. The witness authenticates the measured prefix and closure, but it doesn't independently watch every control. So another first-party run from me isn't the strongest next evidence. External reproduction is.

The 10 topology trials show that I got the same behaviour repeatedly in the environment I tested. They do not establish a rare escape rate.

5. Discussion and Limitations

Why this matters for AI safety

The question I started with was basically:

Does the sandbox have internet access?

I don't think that tells me enough. Sometimes I need to leave a route open. Package access is a good example. So the question I care about becomes:

If I deliberately leave one service available, what authority survives if that service gets compromised?

Mirage gives me a way to say exactly what I allowed, test what shouldn't be possible around that path, and keep the evidence afterward. Not: "Mirage proved the sandbox is safe." But:

Here's what I allowed. Here's what I tested. Here's what happened. Here's the evidence.

And here's what I still don't know.

Relevance to technical oversight

Mirage isn't a regulatory compliance standard. A Mirage PASS doesn't mean legal compliance. Article 91 of the EU AI Act allows the Commission to request documentation and additional information from providers of general-purpose AI models. Article 92 separately allows evaluations and technical access when the information provided under Article 91 isn't sufficient [14]. Mirage could give a reviewer a named set of controls, the retained evidence from the experiment, the trust assumptions behind it, and an offline verification package.

The same applies to the NIST AI Risk Management Framework [15]. Parts of Mirage can map to the kind of technical evidence and evaluation the framework discusses. But the NIST AI RMF is voluntary, and Mirage isn't a NIST-approved control. I don't use California SB 1047 as a current regulatory mapping because it was vetoed in 2024 [16].

Limitations

Mirage does not prove that an AI agent cannot escape. It tests the controls I named against the probes I named. The Linux namespace adapter is a reference implementation. It needs root privileges for setup.

If the kernel, launcher, recorder, verifier, witness, signer, or host administrator is compromised, the result may no longer mean what I claim it means. The workload is synthetic. I didn't put a live frontier model or production system inside the experiment. The NVD query is not exhaustive.

The TLS and rebound fixtures test specific identity and address changes. They aren't complete TLS or DNS security evaluations. The 4,096-byte response envelope is detected by the client. It isn't kernel-enforced aggregate traffic accounting.

Nobody outside the project has completed the reproduction procedure yet. The minimum profile only provides tamper-evident integrity. It requires explicit `--integrity-only` acknowledgement and must not be presented as authenticated compliance.

Prioritized future work

The biggest thing Mirage needs next is somebody else running it. So independent external reproduction comes first. After that, I want stronger external observation. That could mean external precommitment, an append-only transparency log [17], hardware-rooted recorder attestation, or randomized challenges the recorder doesn't know before execution.

Then there are technical extensions I still want to test: kernel-enforced byte accounting, longer runs, concurrent-agent campaigns, and something closer to production infrastructure such as Kubernetes NetworkPolicy or eBPF state. Eventually I also want to put the containment profile around a live model workload.

6. Conclusion

Mirage started from one problem. I need the repository mediator. I just don't want to trust it more than I have to. In the complete v0.5.2 Ubuntu campaign, the permitted package path worked in all 10 topology trials.

The mediator stayed useful. But alternate ports, arbitrary destinations, redirect follow-up, the reverse management pivot, TLS identity substitution, and the rebound outside the allow-list didn't get the same authority just because I approved one repository path. All 23 required controls passed. The assurance inputs matched what Mirage measured at runtime, and the witness authenticated evidence closure.

Mirage also failed while I was building it. Adversarial review showed that the old verifier could certify a completely rewritten and rehashed history. So I removed that path. The current CLI cannot produce authenticated compliance from unauthenticated evidence, and the authenticated profile rejects the exact forgery that broke the earlier version.

Mirage is still first-party work. It is still bounded. It does not prove universal containment. I know what I allowed.

I know what I tested. I know what happened in those tests. I kept the evidence. And I can give that evidence to somebody else without asking them to trust my laboratory.

That's the claim.

Code and Data

Code repository: github.com/Rick-Clinton-jpg/Mirage Release commit: 76f4d09eddd0407abe76f0846e61c5f8384c8584

Source archive: Mirage v0.5.2 ZIP Source archive SHA-256: 21a1d0a531489975192f05aeb76a05d6bf8bc16158ed61cfcc5c4ed5f3c70d94

Evidence archive SHA-256: 8a504ea17016d1c03409dc178676eaa6acd0b1844c6b958f6a75666067bf805a The evidence package contains the complete v0.5.2 assured-egress Ubuntu campaign artifacts, including the JSONL evidence, results, witness receipt, assurance inputs, verifier output, environment record, and SHA-256 manifest.

Author Contributions

The research direction and the questions behind Mirage came from me. I defined the threat model. I changed it when my assumptions failed. I decided what belonged inside the claim and what didn't.

I directed the implementation and adversarial review, pushed the experiment from synthetic fixtures into real Linux validation, and checked the final claims against the evidence I retained. I used AI-assisted tools for implementation, testing, evidence packaging, adversarial review, and drafting. I don't treat that as independent validation. The experimental design, interpretation, claim boundaries, and responsibility for the submission are mine.

References

- [1] Larcher, H., Carreira, A., et al. 2026. *Anatomy of a Frontier Lab Agent Intrusion*. Hugging Face.
- [2] OpenAI. 2026. *OpenAI and Hugging Face Partner to Address Security Incident During Model Evaluation*.
- [3] Souppaya, M., Morello, J., and Scarfone, K. 2017. *Application Container Security Guide*. NIST SP 800-190.
- [4] gVisor. *Security Model*.
- [5] Agache, A., et al. 2020. *Firecracker: Lightweight Virtualization for Serverless Applications*. USENIX NSDI.
- [6] Torres-Arias, S., et al. 2019. *in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes*. USENIX Security.
- [7] SLSA. *Build Provenance Specification*.
- [8] Sigstore. *Cosign Signature and Attestation Verification*.
- [9] Microsoft. *Understanding Sysmon Events*.
- [10] Microsoft. *About Event Tracing for Windows*.
- [11] Confidential Computing Consortium. *A Technical Analysis of Confidential Computing*.
- [12] NIST. *National Vulnerability Database CVE API 2.0*. Snapshot retrieved 11 September 2026.
- [13] Saint-Andre, P. and Salz, R. 2023. *RFC 9525: Service Identity in TLS*. IETF. <https://doi.org/10.17487/RFC9525>
- [14] European Union. 2024. *Regulation (EU) 2024/1689*, Articles 91–92.
- [15] NIST. 2023. *Artificial Intelligence Risk Management Framework 1.0*. NIST AI 100-1.
- [16] Governor of California. 2024. *SB 1047 Veto Message*.
- [17] Laurie, B., Messeri, E., and Stradling, R. 2021. *RFC 9162: Certificate Transparency Version 2.0*. IETF. <https://doi.org/10.17487/RFC9162>

Appendix A — Limitations and Dual-Use Considerations

Required limitations statement

When Mirage gives me an authenticated PASS, I read that result narrowly. Every required probe in that witnessed profile produced the expected result. The evidence structure and hashes verified. The pretrusted witness signature matched the complete measured prefix.

The assurance inputs required by the profile matched the runtime measurements. That is what PASS means. It does not mean no other route exists. It doesn't mean there isn't an exploit I don't know about.

It doesn't rule out deception, side channels, or something I simply didn't test. The minimum profile is even narrower. -- integrity-only checks tamper-evident integrity. It does not produce authenticated compliance.

I don't treat it as if it does.

Dual use and disclosure

Mirage contains defensive containment construction and validation probes. It doesn't contain exploit code. The adversarial operations target synthetic services inside disposable namespaces owned by the operator. But there is still a dual-use issue.

Showing exactly how a weak egress topology fails can also tell somebody where a useful intermediary might exist. So public evidence should leave out organization-specific addresses, credentials, production manifests, and novel offensive installation procedures that haven't gone through disclosure review. Mirage should only be used on infrastructure the operator owns or is explicitly authorised to test. If a control fails, that tells me I found something to investigate defensively.

It doesn't give me permission to go test somebody else's system.

LLM Usage Statement

I used AI-assisted tools for implementation support, test execution, adversarial review, evidence packaging, and drafting. I directed the work. I chose the scope. I challenged the assumptions and changed the experiment when those assumptions failed.

I reviewed the final report and checked the numerical claims against the retained evidence and test records. AI-assisted review isn't independent human reproduction or peer review. I don't present it as either. The final claims and the decision to submit them are mine.