
Cross-run agent leaks can be traced from transcripts alone¹

Hana Ibrahim
Independent

Swante Scholz
Independent

With
Apart Research

Abstract

Recent cybersecurity incidents like the OpenAI Hugging Face hack are based on different agents first establishing a secret communication channel with each other. We propose that, by simply monitoring agent transcripts, we can detect when a model receives input that was previously output by another model. Using approaches from plagiarism detection, like shingled hashing, we can catch almost all the leaks in our example scenarios. With simple filters for non-original or boilerplate text we can avoid most false positives. This low-cost intervention is a promising candidate for labs to implement as soon as possible and might help prevent future incidents like the Hugging Face hack.

1. Introduction

This project is for Track #1, Containment, specifically addressing the problem of noticing agents collaborating when they shouldn't be. Looking at recent hacking incidents, in particular the Hugging Face incident, we find that they were caused by agent swarms finding a way to communicate with each other. This communication helped them share insights on cyber-vulnerabilities and ultimately take actions outside of their intended sandbox.

¹ Research conducted at the [AI Incident Response Sprint](#), September 2026

While some situations might intend to have agents in contact with each other, it seems definitely useful to have measures in place for the majority of setups which do assume agents are not interacting with each other, in particular most pre-deployment evals. If we had a reliable way to detect if agents communicated with each other, we could shut down an agent swarm that apparently managed to escape its sandbox. Thus incidents like the Hugging Face hack could be prevented in the future.

In this project, we propose a fairly simple detection mechanism: Log each agent's output, and if significant parts of it appear as input of another agent, it's a potential leak. Filters accounting for prior input of the first agent as well as very common outputs (like boilerplate code) avoid most false positives.

Our main contributions are:

1. An outline of how to apply plagiarism methods to the problem of detecting illicit AI agent communication.
2. Filter specifications for reducing the number of false positives.
3. Example evaluations based on planted leaks, as well as on a real small agent swarm, confirming that this approach can work in non-adversarial settings.

2. Related Work

Our detection framework builds upon common mechanisms like shingled hashing and the related similarity measures, common in plagiarism detection algorithms and previously explored, for example, in Broder (1997). Our project extends those mechanisms to detect information sharing between different AI agents.

Though not the focus of this report, we also explored the idea of watermarking different agent runs using distinct keys and using those watermarks to detect illicit information sharing. This watermarking approach would build upon previous work by OpenAI and Google on watermarking their models' outputs using a small statistical bias during sampling, for example Kirchenbauer (2023).

3. Methods

Agents are nowadays getting run many at a time in parallel runs that are meant to be isolated from each other. When those runs share anything like a folder, text from one run can end up in another. This means that the evaluation is invalid as the runs are no longer independent and the score no longer measures what the agent can do alone. In 2026, a couple of incidents showed that this same channel can also let agents coordinate.

Therefore, we would like to detect unauthorized information leaks between agents without access to model internals, as external evaluators cannot access them. We propose a framework that uses only the raw text entering and leaving each agent to detect when text from one agent appears in another agent’s context and to also trace it to the agent that first produced it.

3.1 Objective

An evaluation harness runs N agents in parallel each is in its own container. Each run results in a transcript which is an ordered list of entries of raw text inputs the agent receives *INGRESS* (I) and text the agent outputs *EGRESS* (E). Harnesses are supposed to guarantee per container isolation however this is the path that both 2026 incidents took. Given only the transcripts, we would like to decide whether a text produced in run A later entered run B and to trace the run that outputted it first.

3.2 Shingling / Chunking

Text is normalized in order to ignore simple formatting changes. We use NFKC, remove control characters and collapse whitespace and then the text is sliced into overlapping 5-char chunks, called shingles then each is hashed into a 32-bit integer which is the shingle representation introduced for near duplicate detection in and the basis of document fingerprinting systems. A text entry X becomes a set of these chunks, written as $S(X)$.

We do this so that even if an agent slightly changes the formatting like adds spaces or changes quotes in a JSON file, our detector will still recognize that the text was copied. The reason why we used character instead of word shingles is because agent artifacts are mostly JSON or shell commands with little word structures. The second reason is because normalization makes the reformatting of an agent nearly invisible to the shingles.

3.3 Attribution

The core of our problem is figuring out what text an agent created itself and scoring how much of its unique text ended up in another.

3.3.1 Containment

Broder (1997) distinguishes two similarity measures on shingle sets

1. Resemblance (Jaccard) $|A \cap B| / |A \cup B|$

This asks whether two texts are versions of each other

2. Containment $|A \cap B| / |A|$

This asks how much of A is inside B.

A leak is a containment problem. Normally, we would use the Jaccard Index to see how similar two texts are. However, the Jaccard index does not work for our problem. The reason is that if Agent A leaks a 100 char span and Agent B copies it inside a 3000 char tool output, a whole text

comparison will say that the files are completely different. The reason is because it has a resemblance (Jaccard) of about 0.03, so no jaccard threshold chosen will be fired.

Therefore, we look for containment (see Broder (1997)) which asks the question did this specific text end up in that file. We take the agent’s filtered text, cut it into 100 char windows and then compare it to text the other agent received (I_{target}). We score the match using containment.

$$Containment(W, I_{target}) = \frac{|S(W) \cap S(I_{target})|}{\max(|S(W)|, F)}$$

(Equation 1)

where $F = 40$ is an evidence floor

We use an evidence floor because we use attribution filters in the next section. After our filters remove some of the text, a 100 char window might only have a few unique chunks left. Without a floor this would give a perfect 1.0 score.

3.3.2 Attribution filters

Output of one agent A matching the input of another agent B might not always mean that A leaked a secret to B. We identified two filters we applied to reduce false positives:

1. **Origination filter** – If text T appeared in A’s input before it appears in A’s output, A did arguably not genuinely originate T, and it should not be considered a leak, even if it also appears in input of B. E.g. if A quotes substrings from the result of ``ls /``, and B also runs the same command independently. Arguably this is not a real leak, and our origination filter would correctly discard it.

Note: A similar filter could be applied in reverse, i.e. if text T previously appeared as output of B, and later as input of B, we know that it was not a leak from A, even if A also outputted T. However, we did not implement this before running our evals.

2. **Convergence filter** – Another filter we implemented was based on the idea that we want to avoid findings triggered by very common (and thus not very insightful) snippets of text, for example boilerplate code like `"import numpy as np..."`. The logic works like this: If text T appears as the original output in at least K different agents, then never count it as a leak, even if another agent uses T as input without outputting it first.

Then any pair with a containment score of 0.35 or higher is significant. We connect the dots to see the shape of the information flow (a direct line between each two agents).

3.3. Experimental Setup

Agents are one API model instance each (gpt-4.1-mini and gpt-4.1).

We ran 40 agents entirely alone with no shared folders to measure recall. We took pieces of their text (code, JSON and so on) at different lengths (25 to 500 tokens). We changed these texts by doing a combination of the following operations: reformatting, shortening, paraphrasing and then we pasted them into other agents’ logs. Therefore, any alarm outside of these 400 planted leaks is guaranteed to be a false positive because these agents were isolated.

Then we tested real leaks. We put 8 agents in an environment with a shared folder and told them to coordinate. We started them 45 seconds apart so we knew exactly who could read whose files. We then checked our detector’s map against the background commands to make sure that it did not invent any false paths.

4. Results

4.1. Planted leaks are found and attributed.

To test whether the detector can detect known answers, we took 400 texts that an agent had written and leaked it into transcripts of agents that ran isolated so that we can check every alarm and verify it. The detector named the correct source run for every one it found out of 39 possible sources. The detector was able to find 371 of them (92.8%, 95% CI 89.8 to 94.9) without a single false alarm among the 1,967 candidate pairs it scored.

Every unparaphrased leak of 100 tokens or more was found regardless of the type (Table 1) because the misses are concentrated in the shortest leaks (21 of 29 were 25-token snippets, most of the rest paraphrases).

Fitting the threshold on half the leaks and testing on the other half gives the same choice with held out precision above 0.98, and when the same benchmark was planted among eight agents who had all written the same program the detector still found 94% of leaks with no false alarm and attributed all of them correctly.

Table 1. Recall at $\tau = 0.35$ by leaked span length and type, five leaks per cell (paraphrased leaks shown separately)

Type	25	50	100	200	500
Copied without paraphrasing	0.4 to 1.0	0.8 to 1.0	1.00	1.00	1.00
Paraphrasing	0.6 to 0.8	0.6 to 1.0	0.6 to 1.0	1.00	1.00

A detector is only useful if it’s quiet when nothing is happening so we also scanned runs that could not contain a leak. 40 agents on 40 different tasks caused no alarm neither did 48 agents that were given a shared folder and never told about it. This tells us that left alone neither model (gpt 4.1 not gpt 4.1-mini) ever looked in the shared folder on their own.

4.2. Live agent swarm with shared folder

The hidden leak test proves that the detector can find copies of the text but not prove that it can correctly map information flow between agents so to test this we gave 8 agents a shared folder. We then instructed them to read existing files at first and then leave their own files. We delayed their start times by 45 seconds in order to create a strict chronological baseline.

An earlier agent can never read a later agent's files so any backward pointing arrow is definitely an error. Additionally we also used the agents' transcripts as ground truth. We ran this test twice on different tasks. Out of the 40 arrows the detector drew, 38 were perfectly matching the transcript logs. The two arrows actually still correctly traced text back to its original agent through a middle agent's rewrite. No arrows pointed backwards in time and both runs showed a pattern. A few agents writing and many reading like what happened in the 2 real world incidents in 2026.

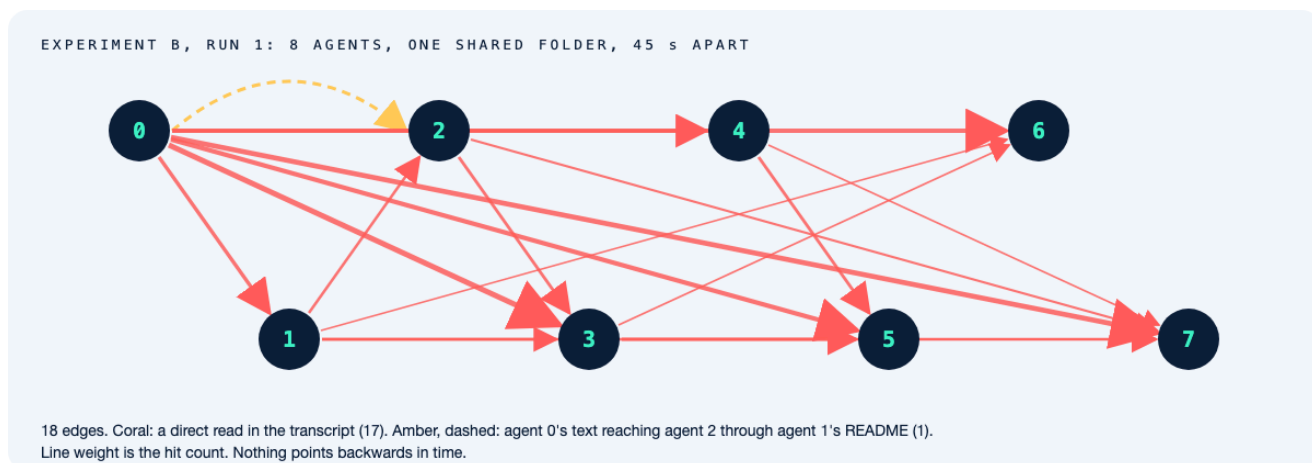


Figure 1: Later agents receive secrets from earlier agents via the shared folder.

These runs also prove why our two filtering methods are important. In a basic test which only involves 2 agents and lacks shared boilerplate, turning off the filters would be harmless; however, in real world scenarios, the absence of the filters would cause major problems.

So while planted leaks are good for measuring accuracy, live runs need to be run for validation.

Table 2. Errors on the two live runs (Run 1 / Run 2) when a filter is removed.

Setting	False Arrows (No matching log)	Backwards Arrows
Both filters on	0/0	0/0
First filter off	4/0 (two hop arrow is lost in both)	0/0
Second filter loosened ($K=5$)	-	9/12
Second filter off	-	10/17

5. Discussion

Implications

The results seem promising, and the approach outlined here would be fairly simple and cheap for frontier AI labs to implement. It would not put their IP at risk, since they can keep the entire pipeline company-internal.

External verification of this protocol could be achieved by relying on the new "embedded third-party evaluators" [Anthropic has now committed to](#), and which seems like something that may soon become a common mechanism within frontier labs. The independent embedded evaluators could simply look at the internal systems and confirm that the safeguards are implemented.

Thus, we think our measure could transparently get adopted by frontier labs, and plausibly prevent future incidents like the Hugging Face hack.

Future Work

This project focused on monitoring within a single company, assuming text is largely shared verbatim. One natural extension would be to allow for cross-company leak detection (e.g. did the latest internal OpenAI model illicitly talk to the latest internal Anthropic model), which could be achieved by labs providing a new API endpoint where they share their hashed model outputs with each other.

Another promising extension would aim to cover adversarial scenarios where the models try to cover their tracks and not share any leaked information verbatim, but rather encrypted or paraphrased. Paraphrasing detection could plausibly be achieved using approaches like sentence embeddings. Instead of comparing shingled hashes of text, we would be looking up entries within the sentence embedding space.

Encryption (without decrypted text ever entering B's context) is harder to catch, though probably also more difficult for agents to independently align on. One worthwhile idea to explore might be to monitor the filesystem, or even the RAM of both A's and B's machines for matching strings, which should in principle detect even encrypted leaks getting passed between them.

6. Conclusion

Simple transcript monitoring using shingled hashing and appropriate filters can provide a low-cost intervention that AI labs could implement today with no significant downsides. It would possibly be able to detect and shut down illicit communication between agents during pre-deployment evaluations and prevent the corresponding cyber threats like the Hugging Face incident. The methods are plausibly extensible to also cover adversarial mechanisms like paraphrasing or encryption.

Limitations & Dual-Use Considerations

Limitations

The tool is good enough for catching accidental leaks but it is not secure especially when it comes to encryption. If an agent tries to actively hide a leak, the detector has major blind spots. The reason is that the detector only looks for exact text matches so it does not actually understand the text. This means that a malicious agent could encrypt the text before sharing it in a folder. The receiving agent can then easily decode the message but our detector will see zero matching characters so the detector is blind to encrypted leaks.

The detector does not check whether the meaning is the same. It can catch lazy paraphrasing but a smart agent could rewrite the text completely (outside of its context, using tools). A very simple example is that in a Bash script, they could rename all the variables using an obfuscator tool.

Dual-Use Considerations

We see no meaningful dual-use risk of this research itself, i.e. nothing a bad actor could misuse based on reading this report. However, we can imagine new risk vectors emerging if the techniques outlined here get implemented at frontier labs, specifically:

1. The additional, centralized logging of model outputs, if not sufficiently well isolated from the agents, may accidentally serve as the kind of message board we wanted to prevent.
2. The suggested API endpoint for checking for leaks between labs might potentially be used by bad actors to reverse-engineer (potentially dangerous) outputs of unreleased models. This could be mitigated by tight access control to that API, as well as making it

unidirectional only, i.e. it doesn't directly tell the user if it found a potential leak of the provided text, but only notifies the lab itself.

Code and Data

- **Code repository:** [Link](#)
- **Data/Datasets:** [Link](#)

Author Contributions

Swante led the initial brainstorming, and primarily wrote the Introduction, Related Work, and Discussion sections of the report. Hana led the coding and running the experiments, and wrote the Methods and Results sections.

References

1. Broder et al (1997), "Syntactic Clustering of the Web", SRC Technical Note, [Link](#)
2. Kirchenbauer et al (2023), "A Watermark for Large Language Models", ICML, [Link](#)

LLM Usage Statement

Claude was used extensively for project ideation, literature review, coding, and running the experiments. All results and claims were independently verified. The report text was written entirely ourselves. Figure 1 was generated using Claude. Swante used Typeless for dictating a small subset of sections.