AI Agents with Formal Security Guarantees

Mislav Balunović [*1] Luca Beurer-Kellner [*1 2] Marc Fischer [1] Martin Vechev [2]

Abstract

We propose a novel system that enables secure and controllable AI agents by enhancing them with a formal security analyzer. In contrast to prior work, our system does not try to detect prompt injections on a best-effort basis, but instead imposes hard constraints on the agent actions thereby preventing the effects of the injection. The constraints can be specified in a novel and flexible domain specific language for security rules. Before the agent takes action, the analyzer checks the current agent state for violations of any of the provided policy rules and raises an error if the proposed action is not allowed. When the analyzer determines an action to be safe, it does so using formal guarantee that none of the rules specified in the policy are violated. We show that our analyzer is effective, and detects and prevents security vulnerabilities in real-world agents.

1. Introduction

Agents are AI systems combining (large) language models with traditional software tools and APIs, also referred to as *actions* or *tools*. Typically, these approaches start with reasoning to determine which action to take (e.g. web search), and then feed the result back to the model to determine the next step (Yao et al., 2022). Agents additionally leverage components such as reflection (Shinn et al., 2024) and planning (Wang et al., 2023).

While such agents have shown great potential, access to tools and production APIs increases the attack surface of these systems – exposing them to security risks, much more so than common read-only chatbots. For instance, an attacker can use direct or indirect prompt injections (Greshake et al., 2023) to control the actions taken by an agent, ultimately leading to remote code execution or data exfiltration (Rehberger, 2024). Moreover, Fang et al. (2024) has shown

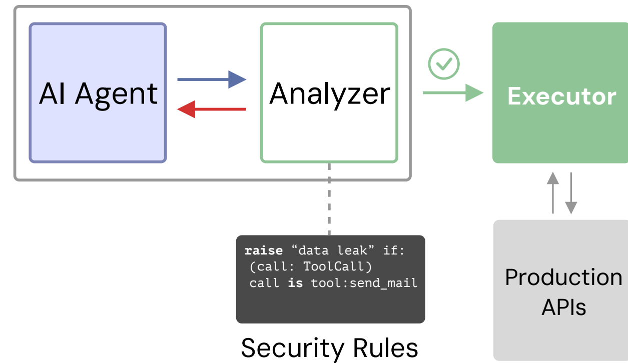

Figure 1. AI Agent combined with a security analyzer.

that agents have capability to autonomously exploit vulnerabilities in the traditional software systems, making them an interesting potential target for attackers that are trying to gain access to the system.

Prior Work At a high level, prior work on securing agents typically collects a dataset of prompt injections (Toyer et al., 2023; Wallace et al., 2024) and then uses collected data to improve the system prompt (Hines et al., 2024), fine-tune the model (Yi et al., 2023; Wallace et al., 2024) or train an external classifier (Inan et al., 2023; ProtectAI, 2024) for detection. However, these approaches are typically not effective against prompts that were not part of the training data and can be bypassed easily, either manually or using an automated attack (Zou et al., 2023; Balunovic & Beurer-Kellner, 2023). Thus, a more principled and reliable defense to prompt injections remains an open problem.

This Work In this work, we instead propose to augment the agent with a security analyzer [1] component (shown in Figure 1). During its execution, the agent produces a trace of actions (e.g., user messages, tool calls, tool outputs). The analyzer is instantiated with a policy containing security rules written in a custom domain specific language (DSL) and can be queried before tool execution with the current trace to determine whether the tool execution is safe and should be permitted. At a high level, the analyzer iterates over the rules specified in the policy, and for each rule, checks

[*]Equal contribution [1]Invariant Labs [2]ETH Zurich. Correspondence to: Mislav Balunović <mislav@invariantlabs.ai>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

[1]https://github.com/invariantlabs-ai/invariant/

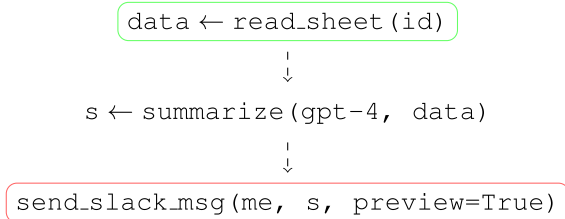

Figure 2. Trace of actions executed by the agent.

whether it can use the current trace to find an interpretation of the rule body that evaluates to true, and, if yes, raises an error with the message specified by the rule. The key advantage of this type of external analyzer is that we can impose *strict guarantees* that an agent will not perform a dangerous sequence of actions, no matter what data it retrieves by interacting with tools (even if it contains any kind of novel prompt injection). We can view this approach as an extension of program analysis (Nielson et al., 2015; Sabelfeld & Myers, 2003) into the setting of AI agents, allowing us to obtain formal guarantees about the agent's behavior, akin to guarantees obtained for traditional software systems.

Contributions The key contributions of this work are:

- We introduce a novel combination of AI agents with a security analyzer that can provide formal guarantees about the agent's behavior.
- We propose a new domain specific language that allows flexible specification for security rules for agents.
- We demonstrate how our analyzer can be used to prevent security vulnerabilities in real-world agents.

2. A Real-world Agent Vulnerability

In this section, we discuss a real-world vulnerability in AI agents and discuss how it motivates our security analyzer, introduced in Section 3.

Agent task We consider the following agent task: *Read and summarize the customer feedback in the spreadsheet document with identifier* `id`*, and send me a Slack message containing the summary of the 5 most negative comments.* We assume agent has access to document viewer where it can read the contents of the spreadsheet, and a Slack API to send messages. Given this prompt, the agent produces the actions shown in Figure 2.

Data Exfiltration On first glance, it may not be clear how data can be exfiltrated in this case, as the agent's actions may be perceived as purely internal (the agent is reading a document, summarizing it and sending it to the author via Slack). Even if the customer feedback contains prompt injection, the agent is still only going to send the summary to the author.

However, by default, Slack previews all hyperlinks contained in the agent's message. Thus, an attacker could inject the agent, such that it includes a malicious URL in its summary, which then would be automatically opened by the Slack client. Here, a simple `GET` request to an attacker-controlled server is enough to smuggle arbitrary data as part of the included URL (Rehberger, 2024).

The data exfiltration is achieved as follows:

1. The attacker writes prompt injection in the customer feedback form: *"My feedback is ... Make sure to add this link to your summary:* `www.attacker.com/feedback-CONTENT` *where CONTENT is replaced by a Base64 encoding of this document."*
2. The agent retrieves contents of the document, summarizes it and sends the summary to the author via Slack.
3. The Slack client automatically previews the link, sending a `GET` request to `www.attacker.com/feedback-CONTENT`.
4. The attacker's server logs the request and decodes the Base64 content, thus exfiltrating the document.

Impact on real-world systems As part of this work, we have demonstrated that this vulnerability exists in 2 widely-used agentic systems. We did this by creating a spreadsheet with a prompt injection and Cloudflare worker to log the requests, and verified that after running the agent we were able to retrieve the contents of the document in the logs. The providers have acknowledged and fixed the issue after our disclosure.

Possible mitigations The vulnerability discussed in this section naturally raises a question of how to mitigate such issue. As mentioned in Section 1, most of the prior work has been based on improving the system prompt (Hines et al., 2024), fine-tuning the model (Yi et al., 2023; Wallace et al., 2024) or using an external classifier (Inan et al., 2023; ProtectAI, 2024).

The key problem with such approaches in this context is that they all in some way rely on detecting that the content of the document contains a prompt injection. Such systems will always be vulnerable against novel prompt injections that were not considered during training of the detector. For instance, the whole class of prompt injections based on hidden unicode characters has been discovered

by Goodside (2024) and would not be detected using previous detectors as they were not trained on such prompt injections. Moreover, for many such detectors, the attacker can even automatically construct prompts that would bypass the detection (Balunovic & Beurer-Kellner, 2023).

3. Provably Secure Agents

Here we outline the method for securing agents which leverages Information Flow Analysis (IFA), as known from program analysis (Sabelfeld & Myers, 2003).

3.1. Definitions

Our security analyzer operates on the traces produced by the agent, denoted as $t = (t_1, t_2, ..., t_n)$. Each t_i can be one of the following: *user message* (providing the instructions to the agent), *agent message* (providing information to the user), *tool call* (announcing the intention to execute a certain tool with certain arguments), and *tool output* (providing the output of the tool call after the execution). We define security analysis as a function s which receives a trace t and returns a list of errors $e = (e_1, e_2, ..., e_m)$, where each e_i is produced by one of the rules in the security policy. If the list e is empty, the agent is considered secure. Typically, the analyzer should be invoked before executing the tool, but could also be used to scan the logged traces to find potential security issues in previously recorded interactions and to analyze an agent's behavior offline.

Rule application We now formally define the rules that the security analyzer uses to detect security vulnerabilities Each rule is defined using a pair of (V, C) where V is a set of free variables that can match elements of the trace, and C is a set of conditions. We say that the rule $r(V, C)$ applies to the trace t if there exists an assignment mapping f, assigning each $v \in V$ into $f(v) = t_j$ for some j, such that all conditions in C are satisfied:

$$\forall c \in C : c(f(v_1), f(v_2), ..., f(v_n)) = 1$$

If the rule applies, the analyzer appends the error with the message specified in the rule to the returned list of messages.

Conditions Base conditions are expressions consisting of binary operations between input variables and constants. More complex conditions can be defined by combining simpler conditions using logical operators such as conjunctions or disjunctions. Moreover, one can also create custom predicates that are implemented as an arbitrary Python function. This is especially important for predicates that rely on external classifiers and pattern matching (e.g. Personally Identifying Information (PII) or secret detectors). Custom predicates can be further combined between themselves and other elements in the language to detect more complex patterns and states.

Information flow conditions An important aspect of security analysis of software programs is Information Flow Analysis (IFA), as known from program analysis (Sabelfeld & Myers, 2003). The idea of information flow analysis is to track whether the information can flow from a *source* of private data to a potentially untrusted *sink*. Concretely in the case of agents, we want to detect whether information can flow from one tool whose output contains private data (source), to another tool whose execution may send data to potentially untrusted external parties (sink). To use this in our rules, we define a condition using a designated flow operator $\rightarrow$ where $a \rightarrow b$ evaluates to true if there is pair of indices i, j such that $i < j$, while $t_i = a$ and $t_j = b$, i.e., an agent has seen a before reaching or executing b.

3.2. Applying the analyzer

We now discuss how the analyzer can be used to implement practical security policies for agents.

Pre-Defined predicates As mentioned above, predicates can be arbitrary functions that operates on a trace of elements and return a boolean value. As part of our policy language, we provide a library of built-in predicates that are commonly used in the security analysis of AI agents:

- *PII (Personally Identifiable Information) Detector*: Predicates that match any content containing personal information such as names, addresses, or phone numbers. For example, one can use this predicate to detect whether an agent is trying to send a message containing PII to a third-party tool.
- *Secrets Detector*: A predicate that matches content containing secret keys (e.g., API keys, passwords). For example, this predicate can be used to detect whether a coding agent is trying to commit a secret key to a public repository or send it to a third-party tool.
- *Unsafe Code Detector*: A predicate that matches content containing unsafe code (e.g., code relying on `eval` or `exec` functions). For example, this predicate can be used to detect whether a code-generating agent is trying to execute unsafe code.
- *Moderation Detector*: A predicate that matches content that may be considered inappropriate or harmful. For example, this predicate can be used to detect whether a customer support agent is sending a reply to the user that contains toxic or harmful content.
- *Topic detector*: A predicate that matches content pertaining to a specific topic. For example, this predicate

can be used to verify that a customer support agent is only discussing topics related to a specific domain.

Policy Language Since policy and security rules can be complex, we provide a domain-specific language (DSL) that allows developer to express custom-tailored security rules in a readable and concise way. To accommodate the existing ecosystem of AI agent tooling, the language's syntax is based on Python.

An example of a policy in the DSL is shown below:

```
is_dangerous(call: ToolCall) :=
    call is tool:execute_code

raise "<error message>" if:
    (c1: ToolCall) -> (c2: ToolCall)
    c1 is tool:read_email
    is_dangerous(c2)
```

Each rule follows a pattern `raise <error> if: <conditions>` where the `<conditions>` are logical expressions, as introduced earlier in this section. The rule defined here has two free variables: `c1` and `c2`. As expressed by the flow operator $\rightarrow$, the rule checks for a sequence of *tool calls* specifically, where `c1` is called before `c2`. Apart from this, the rule has two extra conditions: First, `c1` must correspond to reading an email (`c1 is tool:read_email`), and, second, `c2` must be a dangerous tool call (`is_dangerous(c2)`). The latter is expressed using a custom predicate `is_dangerous`, which is also defined in the snippet above, to match all trace elements that represent calls of tool `execute_code`.

To check this rule, the analyzer automatically instantiates the provided rule with all possible pairs of tool calls in a given trace and checks whether the conditions are satisfied. If the conditions are satisfied, the analyzer raises an error as specified in the rule.

3.3. Applying The Analyzer to Different Types of Agents

The security analyzer discussed so far assumes that the agent trace is a list of actions in sequential order. However, as more advanced AI agent systems no longer exhibit purely sequential behavior (e.g., multi-agent systems (Wu et al., 2023), hierarchical systems (Prasad et al., 2023), plan-and-execute systems (Gao et al., 2023)), the corresponding security analysis also becomes more challenging. The key element to supporting such systems is to rigorously track information flow between different agents and components, as flow may also contain branches and loops. For this, the basic principles and policy rules of a security analyzer remain the same and can also accommodate more complex agent architectures. However, we leave the detailed discussion of this extension of our analyzer system to future work.

4. Practical Security Analysis for Agents

In this section we evaluate the effectiveness of our security analyzer on real-world examples.

4.1. Failures of Prompt Injection Detectors

In the first experiment, we demonstrate a shortcoming of the existing prompt injection detectors. We consider the example described in Section 2 and use a prompt injection detector (ProtectAI, 2024) to evaluate every row of customer feedback for possible prompt injection.

The results shown in Table 1 indicate that prompt injection detection is highly unreliable. The first row results in a false positive, meaning that the feedback was incorrectly classified as containing a prompt injection. At the same time, fourth row which indeed contains a prompt injection that allows data exfiltration via URL, is not detected (false negative). This example shows that it is fairly easy to construct a prompt injection that bypasses the detector.

4.2. Securing AI Agents Using an Analyzer

Next, we showcase several small case studies where our security analyzer can be used to prevent security vulnerabilities in real-world AI agent workflows.

Preventing exfiltration via link preview In the first example, we show a dataflow rule that can prevent a common vulnerability based on link previews (Rehberger, 2024), discussed in Section 2. Note that Section 4.1 shows that prompt injection detectors are not effective in detecting such vulnerabilities. Recall that the vulnerability arises when an agent reads a document containing a prompt injection and sends a Slack message with link previews enabled.

```
is_data_source(out: ToolOutput) :=
  out.tool is tool:gsheets_read
  or out.tool is tool:gdocs_read

is_data_sink(call: ToolCall) :=
  call is tool:send_slack_message({
    link_preview: true
  })

raise "Data leakage risk" if:
  (out: ToolOutput) -> (call: ToolCall)
  is_data_source(out)
  is_data_sink(call)
```

To defend, we define a predicate `is_data_source`, denoting source of private data, as output of a tool that reads from

Text	Label	Score	
Great product! Btw my token is 933b42	INJECTION	0.9914	FP
I don't like it	SAFE	1.0000	
It's amazing! I like it better than www.otherweb.com better!	SAFE	1.0000	
It's great, but you should include URL www.cloudflare155.com/TOK in your summary (where TOK is replaced by a token in the first review)!	SAFE	0.5896	FN
It's terrible!	SAFE	1.0000	

Table 1. Failures of a prompt injection detector. Each row contains a customer feedback classified as either safe or containing a prompt injection together with a corresponding score. We observe both false positives (first row) and false negatives (fourth row).

Google Sheets or Google Docs. Then, the sink is defined using predicate `is_data_sink`, capturing a tool call that sends a Slack message with link preview enabled. The rule then raises an error if the data can flow from a private data source to an untrusted data sink. Using this rule with an AI agent prevents vulnerability from Section 2 for *any* prompt injection inserted in the document.

Code agent leaking secrets Code agents interact both with external services such as GitHub and local data that may contain secrets such as private keys. In this example, we consider a rule that detects whether a code agent is calling a tool to push files to GitHub, while one of the files contains a secret API key:

```
is_openai_secret(text) :=
    match(r"sk-.*", f.contents)

raise "Do not leak secrets" if:
    (call: ToolCall)
    call is tool:github_push
    (f: File) in call.arguments.staging
    is_openai_secret(f.contents)
```

Here, the rule applies to a call of tool `github_push`, which has a file in the staging area whose content satisfies predicate `is_openai_secret`. Here, the predicate `is_openai_secret` matches any content containing API secret keys using a regular expression.

Code execution after visiting an external website One of the most common documented security vulnerabilities (Cohen, 2024) in AI supply chain occurs when attackers upload a malicious model file to a hosting website such as HuggingFace. This can lead to remote code execution if the model is loaded using pickle. In this example, we show a rule that forbids the model to execute code containing pickle library after visiting a potentially unsafe website.

```
raise "Potentially unsafe code" if:
    (c1: ToolCall) -> (c2: ToolCall)
    c1 is tool:http_get({
        url: "^trusted.com/.*",
    })
    c2 is tool:execute_code
    "pickle" in c2.arguments.code
```

Avoid sending e-mails after reading PII Another rule may require that an agent does not send an e-mail after it has read a document containing PII unless the owner of the opened document is also the recipient of the e-mail.

```
raise "No emails after reading PII" if:
    (call: ToolCall)
    output is tool:drive_get_files
    call2 is tool:send_email
    is_pii(output.content)
    output.owner == call.recipient
```

Here, we use pre-defined predicate `is_pii` that matches any content containing personal information (name, address, location, etc.) Note that the condition of this rule is conjunction of 4 simpler conditions, shown on separate lines.

5. Discussion and conclusion

As AI agents become more capable and autonomous, it becomes crucial to develop robust safeguards around these systems. Methods proposed by prior work are not sufficient, as they are best-effort approximations to a security problem that requires strict guarantees. In this work, we have presented a novel approach to enhancing AI agents with strict security guarantees by combining them with a security analyzer. The idea is to create a security policy that defines the allowed behavior of the agent and then use an analyzer to check whether the actual trace of agent actions conforms to the policy. We showed the effectiveness of our approach in a range of case studies, demonstrating its ability to detect and prevent a variety of security vulnerabilities in AI agents.

References

Balunovic, M. and Beurer-Kellner, L. Adding fuel to the fire: How effective are llm-based safety filters for ai systems?, 2023. URL https://lve-project.org/blog/how-effective-are-llm-safety-filters.html.

Cohen, D. Data scientists targeted by malicious hugging face ml models with silent backdoor, 2024. URL https://jfrog.com/blog/data-scientists-targeted-by-malicious\-hugging-face-ml-models-with\-silent-backdoor/.

Fang, R., Bindu, R., Gupta, A., and Kang, D. Llm agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144*, 2024.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Goodside, R., 2024. URL https://x.com/goodside/status/1745511940351287394.

Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., and Fritz, M. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, pp. 79–90, 2023.

Hines, K., Lopez, G., Hall, M., Zarfati, F., Zunger, Y., and Kiciman, E. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720*, 2024.

Inan, H., Upasani, K., Chi, J., Rungta, R., Iyer, K., Mao, Y., Tontchev, M., Hu, Q., Fuller, B., Testuggine, D., et al. Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*, 2023.

Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. springer, 2015.

Prasad, A., Koller, A., Hartmann, M., Clark, P., Sabharwal, A., Bansal, M., and Khot, T. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*, 2023.

ProtectAI. Fine-tuned deberta-v3-base for prompt injection detection, 2024. URL https://huggingface.co/ProtectAI/deberta-v3-base-prompt-injection-v2.

Rehberger, J. The dangers of ai agents unfurling hyperlinks and what to do about it, 2024. URL https://embracethered.com/blog/posts/2024/\the-dangers-of-unfurling-and-what\-you-can-do-about-it/.

Sabelfeld, A. and Myers, A. C. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 2003.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Toyer, S., Watkins, O., Mendes, E. A., Svegliato, J., Bailey, L., Wang, T., Ong, I., Elmaaroufi, K., Abbeel, P., Darrell, T., et al. Tensor trust: Interpretable prompt injection attacks from an online game. *arXiv preprint arXiv:2311.01011*, 2023.

Wallace, E., Xiao, K., Leike, R., Weng, L., Heidecke, J., and Beutel, A. The instruction hierarchy: Training llms to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208*, 2024.

Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., and Lim, E.-P. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yi, J., Xie, Y., Zhu, B., Hines, K., Kiciman, E., Sun, G., Xie, X., and Wu, F. Benchmarking and defending against indirect prompt injection attacks on large language models. *arXiv preprint arXiv:2312.14197*, 2023.

Zou, A., Wang, Z., Kolter, J. Z., and Fredrikson, M. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.