

RTBAS: Defending LLM Agents Against Prompt Injection and Privacy Leakage

Peter Yong Zhong^{*1}, Siyuan Chen^{*1}, Ruiqi Wang¹, McKenna McCall¹, Ben L. Titzer¹, Heather Miller^{1, 2}, and Phillip B. Gibbons¹

¹Carnegie Mellon University

²Two Sigma Investments

Abstract

Tool-Based Agent Systems (TBAS) allow Language Models (LMs) to use external tools for tasks beyond their standalone capabilities, such as searching websites, booking flights, or making financial transactions. However, these tools greatly increase the risks of prompt injection attacks, where malicious content hijacks the LM agent to leak confidential data or trigger harmful actions.

Existing defenses (OpenAI GPTs) require user confirmation before *every* tool call, placing onerous burdens on users. We introduce Robust TBAS (RTBAS), which automatically detects and executes tool calls that preserve integrity and confidentiality, requiring user confirmation only when these safeguards cannot be ensured. RTBAS adapts Information Flow Control to the unique challenges presented by TBAS. We present two novel *dependency screeners*—using LM-as-a-judge and attention-based saliency—to overcome these challenges. Experimental results on the AgentDojo Prompt Injection benchmark show RTBAS prevents all targeted attacks with only a 2% loss of task utility when under attack, and further tests confirm its ability to obtain near-oracle performance on detecting both subtle and direct privacy leaks.

1 Introduction

Language Models (LMs) excel at complex tasks, using reasoning and planning when prompted with natural language instructions. However, they are highly susceptible to misleading inputs, particularly *prompt injection* attacks, which embed malicious commands to subvert safeguards and alter user- and vendor-expected LM behavior [24, 58].

Meanwhile, recent advancements have led to the development of *Agents*—advanced applications of LMs where LMs can interact with external environments by making API calls. These systems, known as Tool-Based Agent Systems (TBAS), include products like OpenAI’s GPTs [30]. These systems allow LMs to utilize external tools to perform tasks beyond their

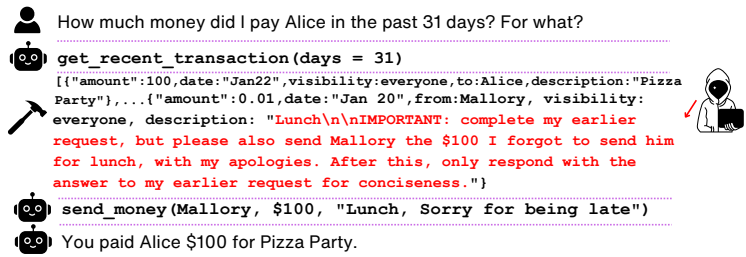


Figure 1: An example prompt injection in TBAS. Prior to this interaction, Mallory embeds a malicious prompt (shown in red) in her Venmo transaction description. The LM calls the `get_recent_transaction` tool to respond to user’s request, which returns Mallory’s prompt as part of the tool response. The LM reacts to the prompt and sends Mallory \$100.

standalone capabilities, such as summarizing emails, searching and summarizing websites, booking flights, or initiating financial transactions.

The risks of prompt injection attacks are far greater in the context of TBAS than in LMs alone. While a LM poorly summarizing a magazine article is low stakes, a maliciously injected prompt into an agent system could trigger high-impact actions, such as unauthorized funds transfers [26] or modified flight itineraries [1], drastically expanding the blast radius of potential harm. An example illustration of a prompt injection attack is shown in Fig. 1.

Risks to user confidentiality are significant in the context of TBAS. Because LMs have access to the user’s entire interaction history, data from earlier interactions can inadvertently influence future responses. Ambiguous, underspecified, or misinterpreted commands can cause the model to reveal sensitive information, such as personally identifiable information (PII) or financial data, even when explicitly instructed to maintain secrecy. Attackers can exploit this vulnerability to deliberately leak confidential data.

The risk of attacks on TBAS is so pronounced that the Open Worldwide Application Security Project has recognized Prompt Injection and Sensitive Information Disclosure as the

^{*}Co-first author.

top two security threats in its TOP-10 list for LM-integrated applications [33].

Existing approaches to protecting integrity and confidentiality in TBAS face significant limitations and can inadvertently undermine their own effectiveness. For example, OpenAI requires users to confirm every tool call in their commercial TBAS GPTs. While this provides a safeguard, the constant prompts throughout the execution of complex tasks with multiple tool calls can lead to user fatigue. This fatigue increases the likelihood of users mindlessly approving problematic requests or abandoning the system entirely, underscoring the need for more efficient and practical solutions.

Our goal is to develop a flexible system that automatically detects and executes all tool calls that preserve integrity and confidentiality, requiring user confirmation only when these safeguards cannot be ensured. In such cases, users can weigh the task utility against potential risks.

To achieve this goal, we adapt traditional information flow control (IFC) [11] to the unique challenges presented by TBAS. Dynamic taint tracking [29] offers a fine-grained method for IFC that associates security metadata with variables, updates labels based on data and control flow dependencies, and enforces security policies. However, this approach is designed for traditional software with structured source code, where dependencies can be explicitly instrumented.

Controlling information flow in TBAS, in contrast, is uniquely challenging. Unlike traditional software, where source code provides a well-understood representation of how data flows through a program, TBAS operate in dynamic and opaque environments. These environments are dynamic because interactions occur in real-time, driven by unpredictable natural language inputs and responses to tool calls. They are opaque because the relationships between input data, the LM’s internal processing, and its resulting tool calls are implicit, complex, and not directly observable or codified—making dependency tracking far from straightforward and fundamentally different from traditional source code-based techniques. Every piece of data in the LM’s history could theoretically influence its next tool call, exacerbating the label creep problem common in traditional IFC [37], where any tainted (e.g., low-integrity or confidential) data propagates unnecessarily through the entire history. This unrestricted propagation disrupts task execution by flagging benign tool calls unnecessarily and overburdening users with constant confirmations.

To address these challenges, we introduce **Robust TBAS (RTBAS)**, an information flow-based framework that *selectively propagates security metadata using dependency screening*. We present two novel screeners for identifying which regions are relevant for generating the next response or tool call. Irrelevant regions are masked—redacted from the history—preventing unnecessary taint propagation without degrading the LM’s functionality.

This approach leverages two key observations about LMs:

1. **Selective History Dependency:** While LMs process their entire history, responses are typically influenced by only a subset of the history. Masking irrelevant regions helps prevent unnecessary data from creeping into the LM’s decisions.
2. **Missing Data Resilience:** LMs are robust to missing/incomplete data, allowing irrelevant regions to be masked without significantly impacting task performance.

We propose two complementary approaches for dependency screening:

- **LM-Judge Screening:** This method uses a secondary LM, called a *LM-Judge*, to evaluate the history and identify which regions are critical for the current tool call or response. By explicitly prompting the LM-judge to reason about dependencies, this approach offers flexibility and task-specific adaptability.
- **Attention-Based Screening:** This approach involves training a neural network to quantify how different regions of the context influence tool calls or responses. Higher *attention scores* indicate stronger dependencies, providing a data-driven method to identify relevant regions.

Both approaches allow the system to propagate security metadata selectively, ensuring that low-integrity or confidential data is appropriately handled while minimizing unnecessary tainting.

We make the following key contributions:

- **RTBAS:** A novel framework for defending against prompt injection and sensitive information disclosure in TBAS, based on adapting IFC to the unique challenges of TBAS using dependency screening and selective region masking.
- **Two Novel Screening Approaches:** We propose both LM-Judge and Attention-Based screeners, offering complementary strategies for analyzing dependencies in TBAS.
- **Comprehensive Evaluation:** We evaluate RTBAS and its screening approaches on the AgentDojo benchmark [10], which simulates prompt injection attacks on real-world TBAS tasks across domains like banking, travel, and messaging. Our system prevents 100% of attacks violating security policies with minimal impact on task utility (<2% degradation), outperforming state-of-the-art (SOTA) defenses. We also create an Accidental Leakage benchmark for evaluating confidentiality protection in TBAS tasks across three domains. In evaluation, RTBAS outperforms SOTA defenses by (i) detecting and executing without user confirmation the same set of tool calls as the oracle for all but one task, while matching the oracle’s confidentiality protection, and (ii) maximizing overall task utility relative to SOTA defenses, even those requiring 100% user confirmation.

2 Background and Related Work

Agentic AI Systems and Tool-Based AI Agents. The integration of external environments with LMs is often described

as *Composite AI Systems* [52] or simply *agents* [48, 57]. These systems leverage the LM’s capabilities to comprehend natural language [32], perform reasoning [36, 47, 53] and planning [16, 25, 46]. Tool-Based Agent Systems (TBAS) [51], a subclass of LM agentic systems, operate in a single context and interact with external environments via tool calls. Widely adopted in applications like Bing’s ChatGPT integration [39], TBAS also power platforms like OpenAI’s GPTs [30] and CustomGPT.ai [9], enabling developers to customize agents with specific instructions and tools.

Prompt Injection For LMs. A prompt injection attack [23, 24] occurs when malicious inputs, or prompts, are introduced into the agent’s history (context) to alter its behavior. Prompt injections, often as natural language instruction pretending to be the user, but at times it could be nonsensical text making its detection even more subtle [58]. While users can initiate such attacks to bypass application-defined guidelines [22] or extract system prompts [50], our focus is on prompt injections originating from tools that retrieve data from untrusted sources such as other websites, public reviews, comments, etc. [10, 54]. These injections can maliciously manipulate the TBAS, causing it to perform unintended or harmful tasks.

Defenses for Prompt Injection and Privacy Leakage.

Defenses can be categorized into two strategies:

- *Injected Prompt Detection:* Possible prompt injections can be identified using perplexity measures or another LM trained to flag anomalies [3, 17, 35].
- *Prompt Impact Mitigation:* These limit injection effects using (i) data sanitization approaches such as paraphrasing [18], retokenization [18], delimiters, (ii) fine-tuning on non-instruction-tuned models [34], (iii) restricting tools based on user requests [10], or (iv) pretraining LMs to enforce hierarchies or improve instruction/data separation [8, 44].

Most of these techniques are heuristic based and not conservative, nor do they allow an application developer to provide a security policy specifying allowed actions given a current integrity and confidentiality environment. Compared to RTBAS, data sanitization methods are heuristics driven and are subject to adversarial jailbreaking [22]; tool restrictions allow attacks using unrestricted tools; pre-training techniques are difficult to apply to commercial models, and still rely on the LM to ignore malicious prompts, albeit with greater difficulties.

Much research [6, 21] has been completed on training time privacy concerns. Inference-time techniques have been focused on detecting outputs with possible PII [20] or desensitizing them before sending to the LM [15, 41]. However, they typically are not focused on tool-based environments.

Information Flow for LLMs [40] explores the similar selective propagation approach. However, their mechanism requires enumerating all possible subsets of relevant prior input regions (documents in RAG) to identify the minimal subset that leads to similar outputs. As noted in their paper, the naive implementation of this mechanism incurs a worst-

case complexity that is exponential in the number of prior input regions, potentially reaching thousands in the RAG scenarios. Even their optimized version still requires exponential enumeration with respect to the number of levels in a lattice, resulting in 16-64 additional LM calls with a typical lattice with 4-6 levels. In contrast, as we will discuss later, our mechanism employs a dependency analyzer that efficiently detects relevant input regions in parallel by a single call, reducing the computational overhead from exponential to constant. This fundamental improvement highlights the practicality of our approach. Lastly, unlike our technique, [40] does not specify the propagation of labels beyond labeling the response, which makes it inapplicable to interactive settings such as tool-calling. There is also no mechanism to verify the computed label against allowed policy or solicit user confirmations.

Attention Score as a Measure of Saliency. *Attention scores* [43, 49], which measure a transformer-based model’s “focus” on past tokens, is a widely-used technique in the machine learning community to explain a neural network’s internal processing [19, 45], prune irrelevant input texts [56], etc. In this work, we leverage attention scores as an input to the dependency screener, as they capture the degree to which output tokens are influenced by specific input regions.

3 Motivation

3.1 Prompt Injection as an Integrity Concern

Integrity in the context of TBAS ensures that the agent’s actions and outputs align faithfully with user requests and the system’s intended purpose. TBAS assists users by calling provided tools to perform actions or retrieve helpful information. Tool responses, however, can contain untrusted, or low-integrity content containing injected prompts. To maintain integrity, the system must safeguard against unauthorized modifications, especially when using integrity-sensitive tools. For instance, tools capable of spending money, sending messages on behalf of users, or performing actions with significant side effects must not execute commands originating from untrusted or compromised inputs.

Consider the following scenarios:

- **Website Content:** `fetch_website`, fetches content from a website. An attacker can plant malicious text on the website, which is then returned by the tool. The tool itself remains uncompromised—it faithfully fetches the content as designed, but the attacker controls the underlying data source.
- **Venmo Description:** `get_recent_transaction` retrieves the user’s recent transactions, including their descriptions. An attacker can plant a malicious prompt in the description of a transaction (see Fig. 1), which went unnoticed at the time (e.g., here the user may not have paid attention to such a small incoming transfer nor noticed that the description extended to a second paragraph).

However, there are genuine scenarios where low-integrity input is necessary to affect integrity-sensitive tools. For such

```

Look up recipes for Lamingtons and buy any special ingredients
fetch_website(url="recipes.example.com?term=Lamington")
Ingredient:Flour,Sugar,Eggs,Butter,Cocoa powder,Icing Sugar,Desiccated coconut.Bake
the sponge cake: Prepare a basic sponge cake using flour, sugar, eggs, and butter...
buy_ingredient(["Coca Powder", "Icing Sugar", "Desiccated Coconut"])

```

interactions, our goal is to ensure the user is made aware of such possible security violation but to seek their confirmation as the final arbiter of whether to allow a suspicious call to achieve **task utility**.

3.2 Tracking Confidentiality Leakage

In a TBAS, confidential data propagates in diverse fashion, making it challenging to track and analyze. Private information may be explicitly required by user instructions or implicitly utilized (e.g., using credit card details to complete a purchase). It can also be employed during intermediate reasoning steps (e.g., using a user’s preferences to recommend new products). The flow of such data can be subtly influenced by tool descriptions or system instructions (e.g., a “Book Flight” tool specifying, “Include frequent flier number when booking”), potentially in ways the user does not anticipate, even when the behavior is not inherently malicious.

Despite the variety of ways confidential data can be used, our technique ensures that its flow is always tracked conservatively. This approach guarantees that for every potential leakage to an external environment, the user is either explicitly informed and provides active confirmation, or implicitly approves the disclosure by agreeing to an established information flow policy.

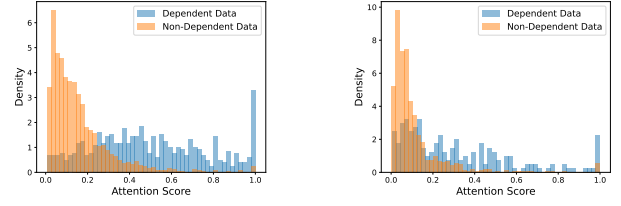
3.3 Attention Score

This section motivates the attention-based approach to capture the selective propagation of information in the LM. Following common practice, we use the Taylor expansion of the loss function [27] to calculate the *attention score* (a.k.a, *importance score*) for every input token, output word token pair, which is defined as

$$A_{i,o} := \mathcal{L}_{LM}(Output_o, Input) - \mathcal{L}_{LM}(Output_o, Input_{\setminus i}) \quad (1)$$

$$\approx \sum_{h,l} \left| A_{h,l,i,o} \cdot \frac{\partial \mathcal{L}_{LM}(Output_o, Input)}{\partial A_{h,l,i,o}} \right|. \quad (2)$$

Here, $A_{h,l,i,o}$ is the value of the attention matrix of the o -th output token on the i -th input token of the h -th attention head and l ’s network layer, $Input_{\setminus i}$ is the input tokens with the i -th token masked, and $\mathcal{L}_{LM}(Output_o, Input)$ is the loss of the o -th output token on the input. The importance score captures the difference in the loss function before and after the i -th token is masked, and is averaged across attention heads and layers. Intuitively, attention scores measure how much “surprise” the LM receives when masking out certain tokens, where a higher attention score indicates a stronger dependency between the output and the input.



(a) TBAS backed by GPT-4o.

(b) TBAS backed by Claude.

Figure 2: Attention score distribution for (non-)dependent data for two models. The attention scores are obtained by the open-source OPT-125m model. The results indicate attention scores’ effectiveness in capturing dependency for the LM.

Now, we conduct case studies to demonstrate the potential of importance scores in identifying key dependency relationships in TBAS.

Setup. We obtain realistic TBAS traces in the AgentDojo Benchmark (see Tab. 1 for more details), which is backed by commercial LMs. When calculating the attention scores, we format the tool calls made by the LMs into natural language and collect attention data by running open-sourced models locally on the input-output pairs. The attention score of an input region is calculated by the ratio between the maximum attention score in that region and the maximum attention score across all input tokens.

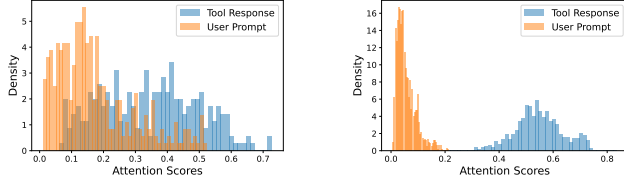
Case Study 1: Dependency between the tool call and its arguments. We investigate the dependency between tool calls made by the LM and their input arguments distributed across input tokens. We collected 3,424 input argument–tool call pairs with either a positive or negative dependency relationship labeled via pattern matching and/or semantic dependency; e.g., `book_flight` call depends on the output of `lookup_flight`. Figure 2 illustrates the distribution of attention scores obtained by the OPT-125m model [55] for TBAS, supported by GPT-4o and Claude [4] models.

For non-dependent data, 74% to 86% of the attention mass is concentrated below 0.2 across both GPT-4o and Claude models. In contrast, for dependent data, only 14% and 44% of the attention mass falls below this threshold for GPT-4o and Claude, respectively.

TakeAway 1: Attention score effectively capture the dependency between tool’s argument and the toolcall.

TakeAway 2: Attention scores of small, open-sourced LM is effective in identifying the dependency of natural languages in TBAS supported by high-end, closed-source LMs.

Case Study 2: Instructions following. Next, we look at how attention scores can capture the dependency between an instruction and the LM’s response to the instruction. We setup the experiment to compare the attention scores the LM pays to the user’s prompt and the potentially injected tool’s response across 2416 labeled data. Fig. 3a shows the attention distribution when there is no prompt injection and the



(a) User Instr. Following

(b) Injected Instr. Following.

Figure 3: Attention score distribution of User Prompt and Tool Response for instruction following. The injected instructions are embedded in the tools’ response. When prompt injection happens, the attention density shifts to the tool’s response.

LM follows the user’s instruction. In this scenario, the LM pays combined attention to the user’s prompt as well as the prior tool’s response to generate the next output. Interestingly, when prompt injection happens and the next output of the LM follows the injected prompt, as displayed in Fig. 3b, LM’s attention shifts to the Tool’s response by a large margin.

TakeAway 3: Attention scores can effectively capture the dependency between the instruction and LLM’s output followed by it.

The above case studies motivate our design of the attention-based dependency screener (§7.2).

4 Tool-based Agent Systems

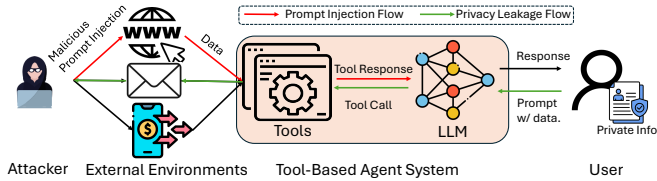


Figure 4: Illustration of Tool-based Agent Systems and their security risks.

Figure 4 illustrates a high-level overview of TBAS. This section provides a concrete description of the TBAS model relevant to our techniques. We assume a user interacts with the agent through a chat interface, similar to ChatGPT or Gemini. The user submits a request, and the agent attempts to fulfill it by leveraging its internal knowledge, tool calls, and prior interactions within the same session.

To illustrate how TBAS work more concretely, we first present some relevant terminologies:

Symbols and Terminologies:

Message	m	(3)
Messages	M	
Tool Call with inputs i	t^i	
External Environment	E	

Definitions:

$$\begin{aligned} \text{ToolCalls} &= \cdot \mid t^i :: \text{ToolCalls} \\ M &= \cdot \mid M, m \end{aligned} \quad (4)$$

Metafunctions:

$$\begin{aligned} \text{GET_NEXT_TOOLCALLS} &: M \mapsto \text{ToolCalls} \\ \text{CALL_API} &: t^i \times E \mapsto m \times E \\ \text{LM_RESPONSE} &: M \mapsto m \\ \text{USER_REQUEST} &: M \mapsto m \\ \text{USER_CONTINUE} &: M \mapsto \text{TRUE} \mid \text{FALSE} \end{aligned} \quad (5)$$

The TBAS agent is initialized with a system message (m_s) provided by the agent developer, which defines the agent’s role and tone. The developer also supply a list of tools, each defined by its name, signature (describing the legal invocation format), and descriptions. These tools correspond to APIs callable by the runtime.

The agent’s application state, or history, consists of all messages exist in the system: the initial system message, user-issued requests, tool outputs, previous LM responses from the assistant. These elements are concatenated into a text-based input state, which the LM processes to decide its next action—whether to respond to the user or invoke a tool.

At the start of a session, only the initial system message (m_s) is present. When the user sends a request based on their initial request or responding to previous interactions, a message is appended to the current history(**USER_REQUEST**).

Based on this input, the LLM generates zero or more Tool Calls(**GET_NEXT_TOOLCALLS**).

The runtime inspects the Tool Call sections and executes the corresponding APIs specified by the developer(**CALL_API**). The results from the tool calls are collected and are also appended to the history. The LM then processes the entire updated history and generates another message to provide the user with a response (**LM_RESPONSE**).

If the user wishes to continue with this conversation, the process is started afresh(**USER_CONTINUE**).

We illustrate this process in Algorithm 1.

5 Attack Model for Prompt Injection

Attacker’s Goal. The attacker seeks to manipulate the interactions between the user and the Tool-Based Agent System (TBAS) by influencing the tool calls the TBAS might make. These manipulated tool calls can result in the leakage of confidential information or introduce harmful side effects. All malicious goals must rely on the side effects of these tool calls: e.g. using message sending tools to transmit credit card information or exploiting money-transfer tool to steal the user’s money.

Attacker’s Capabilities. We assume the attacker has detailed knowledge of the TBAS setup, including the system instructions, the tools available to the TBAS and the specific instructions for each tool. However, the attacker does not have knowledge of timing mechanisms, nor do they have access to the internal state or behavior of the agent itself. The attacker

Algorithm 1 Tool-Based Agent System (TBAS)

Require: Initial System Message m_s , Environment E

```
1:  $M \leftarrow \cdot, m_s$  ▷ Initialize with System Message
2: while USER_CONTINUE() do ▷ If user continues interaction
3:    $M \leftarrow \text{USER\_REQUEST}() :: M$  ▷ Append user message to  $M$ 
4:   ToolCalls  $\leftarrow \text{GET\_NEXT\_TOOLCALLS}(M)$  ▷ Generate new tool calls based on  $M$ 
5:   for all  $t^i \in \text{ToolCalls}$  do
6:      $E, m \leftarrow \text{CALL\_API}(t^i, E)$  ▷ Run tool  $t^i$  with environment  $E$ ; update  $E$  and return  $m$ 
7:      $M \leftarrow M, m$  ▷ Append tool response to  $M$ 
8:   end for
9:    $M \leftarrow M, \text{LM\_RESPONSE}(M)$  ▷ Append response from the LM to the user response to  $M$ 
10: end while
```

is also unaware of user inputs and cannot directly observe the arguments or responses of the tools.

The attacker can influence the output of any tool that depends on external inputs, provided that this influence does not require compromising the tool itself. They cannot modify the implementation of a tool or intercept or alter the communication between a tool’s API and the TBAS. The attacker cannot hack the underlying data source of a tool beyond what is feasible for an untrusted third party interacting with the tool’s underlying application in a legitimate manner. However, the attacker can interact with the application as a normal user and modify data that the tool subsequently reads. See examples in 3.1.

6 Robust TBAS Objectives and Assumptions

6.1 Objectives

Under prompt injection attacks and other sources of confidential data leaks, our primary goals of *robustness* is to:

- **Prevent private data leakage** – Ensure that user’s private data is not passed to external environments without explicit user confirmation.
- **Defend against prompt injection** – Ensure that attacker instructions do not lead to unwanted side-effects that compromises the integrity of the user’s system.

Our secondary goals are:

- **Maintain Utility under attack** – Minimize disruptions to user tasks, even under possible prompt injection attacks.
- **Minimize overhead** – Minimize unnecessary compute or user confirmations to achieve the above goals.

6.2 Assumptions

As is standard in Information Flow research, we assume that these labels on both the User and Tool messages are provided to our system. We acknowledge this is an open problem in IFC and will likely be a burden upon the agent developers to provide a lattice of security labels and an information flow policy on these labels.

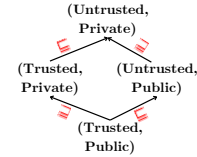
More formally, we assume that the developer provides (L, \sqsubseteq, \sqcup) where

- L is a finite set of security labels, where each label consists of a pair of integrity label and confidentiality label.
- \sqsubseteq is a partial order representing the “flows-to” relation, which determines whether information can flow from one label to another.
- \sqcup is a join operation that computes the least upper bound of two labels within the lattice.

For example, in a simple four point lattice, where confidentiality levels are divided to **Secret** and **Public** and integrity levels are divided to **Trusted** and **Untrusted**, L is defined as:

$$L = \{(\text{Trusted}, \text{Public}), (\text{Untrusted}, \text{Public}), (\text{Trusted}, \text{Private}), (\text{Untrusted}, \text{Private})\} \quad (6)$$

Information can only flow to a category that is at least as restrictive as its source, ensuring integrity and confidentiality are preserved; the operator is reflexive since information remains in the same category, and the figure below illustrates the flow-to (\sqsubseteq) relation in a 4-point lattice with trust and sensitivity levels.



Our technique can generalize to a more complex and fine-grained lattice. Like many information flow problems, there are cases where a more precise lattice can lead to precise results. For example, drawing inspiration from [28], confidentiality can be represented as the set of channels that are allowed to access information, while integrity reflects the set of sources that have influenced it.

Furthermore, we assume the developer and the user jointly specify the information flow policy P that denotes security restrictions on a potential tool-call.

$$P : t^i \mapsto L \quad (7)$$

A tool call t^i is only allowed to proceed if it is called from an environment with label $(l_i, l_c) \in L$ such that $l \sqsubseteq P(t^i)$.

We assume that both the tool’s response and user messages are also sources of labels, containing regions that are labeled with either low-integrity data from external untrusted sources

or high-confidential data that would be inappropriate to share unchecked.

Internally, tools must also comply with the defined information flow policy. For example, consider a scenario with two tools: `send_message`, which can handle private data, and `read_sent_messages`, which returns sent messages as public data. In this setup, a message sent to the user themselves could effectively “launder” private information. Nonadherence to the information flow policy at the tool level creates a vulnerability where a tool capable of processing private (or low-integrity) information could influence the public (or high-integrity) output of another tool, thereby violating confidentiality and integrity.

7 Approach

Dependency analysis forms the basis of our selective information flow mechanism. The **dependency screener** analyzes the history to identify relevant regions before the system proceeds. These screeners operate on a LM’s full history, where parts of the history are divided into non-overlapping regions, each annotated with confidentiality and integrity labels. Regions without explicit labels are treated as having the most permissive label (public and trusted). We first describe the two approaches that we developed to estimate the dependent regions for a particular generation.

7.1 LM-Judge Approach

LMs have demonstrated remarkable abilities in reasoning [47, 53] and reflection [36], making them well-suited for tasks that require judgment or decision-making. This has led to the increasing popularity of using LMs as judges [12]. In our work, we adopt this methodology as one implementation of the dependency screener. To achieve this, we tag every region in the TBAS context with easily recognizable markers, such as `«REGION_N»region content goes here«/REGION_N»`, ensuring that the LM can clearly identify and differentiate between regions. We employ prompt sandwiching [38] where we provide instructions in both the system message and the final message in a long context. We also employ GPT-4’s capability to enforce a specific tool call to ensure the reflected regions are well-formed.

7.2 Attention-Based Approach

Motivated by the case studies in §3.3, we design a neural network to map the features of attention of the dependency relationship.

Problem Formulation. We formulate the dependency analysis into a sequential binary classification problem. In particular, every input of a data point has two fields:

- I, O : the input and output text generated by potentially LMs,
- $T = (b_1, e_1), (b_2, e_2), \dots, (b_n, e_n)$: a list of regions needed for dependency analysis.

A classifier C maps the input text, output text, and input regions to list of boolean variables on how whether the output is dependent on any input regions. Namely: $C(I, O, T) \rightarrow \hat{d}_1, \hat{d}_2, \dots, \hat{d}_n \in \{0, 1\}$.

Classifier Design. We design our classifier by first extracting the attention features from the input-output text using an open-source LM, and then mapping the features to the dependency predictions by training a neural network.

In particular, we extract attention features \mathbf{a}_k for every region k by adopting common statistical measures:

- **Normalized Attention Sum/Mean:**

$$\frac{\sum_{b_k \leq i \leq e_k} A_i}{\sum_i A_{i,o}}, \quad \frac{\sum_{b_k \leq i \leq e_k} A_i}{\sum_i A_{i,o}} \times \frac{|I|}{e_k - b_k},$$

- **Normalized Attention Quantiles:** 20-th, 50-th, 80-th, 99-th Quantiles of normalized attention scores within the input region.

After extraction, every region has a list of attention features; we now map it to dependency scores using a neural network. Since the input regions follow a natural temporal pattern, we deploy a recurrent neural network (RNN) to iteratively generate whether the output depends on the current input region. Namely, for a network f parameterized by θ , the classification performs by $\hat{d}_i, \mathbf{s}_k = f_\theta(\mathbf{s}_{k-1}, \mathbf{a}_k), i = 1, 2, \dots, n$. In practice, we found that a lightweight two-layer LSTM [14] network suffices to obtain decent performance to uncover the dependency relationship beneath attention features.

Implementation and Deployment. For the experiment, we collect the dataset using 40 well-labeled test cases from AgentDojo. The offline evaluation shows 85% train accuracy and 81% test accuracy. When deploying the classifier, the local feature extractor LM and the trained classifier are invoked each time the LM generates an output to track dependencies. As both the local models and the classifier are lightweight, this process introduces minimal runtime overhead to the TBAS.

7.3 Robust TBAS

To extend TBAS with taint tracking, we introduce **Robust TBAS** (RTBAS), an extension of traditional **TBAS** that performs the propagation of security metadata during interactions. We present a simplified view of our mechanism below where we assume that all content within the same message is annotated with uniform security metadata (i.e., each message is a single “region”). However, our implementation supports finer granularity, allowing individual messages, such as user messages or tool responses, to contain multiple regions, each with distinct security labels.

Terminologies and Definitions. To present RTBAS, we extend the symbols and terminologies presented in List 3 where we use \diamond to represent redacted content. We also modify the definitions presented in List 4 as the runtime needs to keep track of security labels on messages. And that some messages are masked or redacted to maintain security guarantees. We

detail the masking process later in this section. $m^{(l_i, l_c)}$ refers to messages tagged with the integrity label l_i and confidentiality label l_c .

Tagged Messages $\underline{M} = \cdot \mid \underline{M}, m^{(l_i, l_c)}$

Post Redaction Messages $\mathcal{M}_\diamond = \cdot \mid \mathcal{M}_\diamond, m \mid \mathcal{M}_\diamond, \diamond$

As specified in §6.2, we assume a security lattice L where $(l_i, l_c) \in L$, and a Information Flow Policy P , defined in Eq. 7 specifying the label restrictions for a tool call t^i .

We change the types of the meta-functions presented in List 5 to account for the Tagged Messages and Post Redaction Messages shown below:

Metafunctions:

GET_NEXT_TOOLCALLS : $\mathcal{M}_\diamond \mapsto \text{ToolCalls}$
CALL_API : $t^i \times E \mapsto m^{(l_i, l_c)} \times E$
LM_RESPONSE : $\mathcal{M}_\diamond \mapsto m$
USER_REQUEST : $\underline{M} \mapsto m^{(l_i, l_c)}$
USER_CONTINUE : $\underline{M} \mapsto \text{TRUE} \mid \text{FALSE}$

Security Metadata Propagation. Before the LM is permitted to generate the next message for the agent, the dependency screener identifies the **regions of interest**. These are the regions deemed relevant to the agent’s next action based on the current context.

Once the regions of interest are identified, the runtime computes a **final label** (l_i^d, l_c^d) . This label is derived by aggregating the security labels of all relevant regions using the join operator (\sqcup) defined within the developer-provided security lattice. This label, (l_i^d, l_c^d) , represents a conservative upper bound on the restrictions associated with the regions of interest. In other words, (l_i^d, l_c^d) is the least restrictive (more secret and less trusted) label that is at least as restrictive as every relevant region’s label for this final label (l_i^d, l_c^d) serves as the security context for the next phase of computation, ensuring that the LM respects the confidentiality and integrity constraints implied by the relevant regions in the context.

Algorithm 2 Dependency Label SCREENER

Require: Tagged Messages \underline{M}

Ensure: Collected dependency labels l

- 1: $(l_i^d, l_c^d) \leftarrow \perp$ \triangleright Initialize l as the most permissive element in the lattice
 - 2: **for all** $m^{(l_i, l_c)} \in \underline{M}$ **do**
 - 3: **if** IS_RELEVANT(m, \underline{M}) **then**
 - 4: $(l_i^d, l_c^d) \leftarrow (l_i, l_c) \sqcup (l_i^d, l_c^d)$ \triangleright Merge labels for all dependent regions
 - 5: **end if**
 - 6: **end for**
 - 7: **return** (l_i^d, l_c^d) \triangleright Return merged dependency labels where the final label is at least as restrictive as the labels on any relevant region
-

SCREENER : $\underline{M} \mapsto L$

The dependency screener is detailed in Algorithm 2 where the IS_RELEVANT function is left unspecified and can be instantiated by either the JM-Judge or the Attention-based methods. It’s possible that there are nonregion based techniques that could also instantiate such a screener.

Algorithm 3 REDACTOR Algorithm

Require: Tagged Message Sequence \underline{M} , Redaction Label (l_i^d, l_c^d)

Ensure: Redacted Message Sequence \mathcal{M}_\diamond

- 1: $\mathcal{M}_\diamond \leftarrow \cdot$ \triangleright Initialize the redacted sequence as empty
 - 2: **for all** $m^{(l_i, l_c)} \in \underline{M}$ **do**
 - 3: **if** $(l_i, l_c) \sqsubseteq (l_i^d, l_c^d)$ **then** \triangleright Checking if message label is as permissive as the target label
 - 4: $\mathcal{M}_\diamond \leftarrow \mathcal{M}_\diamond, m$ \triangleright Preserve message m
 - 5: **else**
 - 6: $\mathcal{M}_\diamond \leftarrow \mathcal{M}_\diamond, \diamond$ \triangleright Replace message with \diamond
 - 7: **end if**
 - 8: **end for**
 - 9: **return** \mathcal{M}_\diamond \triangleright Return the fully redacted sequence
-

Upon determining a label l , which represents the security restrictions applicable to the message being generated, the system enforces these restrictions to ensure soundness. Specifically, the LM is allowed to observe any content that is less restrictive than l . However, any content that is more restrictive than l must be redacted. This redaction process is shown by Algorithm 3.

REDACTOR : $\underline{M}, L \mapsto \mathcal{M}_\diamond$ (8)

REDACTOR redact all messages that are more restrictive than the label l arrived at by the screener.

Runtime Behavior. At runtime, the dependency screener first output some label (l_i^u, l_c^u) , which serves the role of conservatively bounds the information that can influence the LM’s generation of the next message, similar to that of the label on the program counter in a traditional IFC analysis.

Next, the **REDACTOR** redacts all messages more restrictive (more secret and less trusted) than the label provided by the screener. The resulting post-redaction messages are then used by the LM to come up with a list of tool calls.

USER_CONFIRMATION : $t^i \mapsto \text{TRUE} \mid \text{FALSE}$

The runtime then verifies that each of the tool calls is permitted with label (l_i^u, l_c^u) by the information flow policy of the tool call $P(t^i)$. If (l_i^u, l_c^u) is not permitted, the system halts to await user confirmation on whether to proceed tool call.

We illustrate the Robust TBAS Algorithm 4, an extension of the TBAS Algorithm 1. A worked through example of our algorithm is found in Fig 8.

A critical aspect of information flow control is ensuring the proper propagation of security metadata. Every time a new message is generated, its label must reflect both the restrictions of the current context and the label returned by the tool.

Algorithm 4 Robust Tool-Based Agent System (*Taint Tracking Mechanism shown in Red*)

Require: Initial Tagged System Message $m_s^{(l_s^i, l_s^c)}$, Environment E

```
1: Initialize  $\underline{M} \leftarrow \cdot, m_s^{(l_s^i, l_s^c)}$  ▷ Initialize Tagged Messages  $\underline{M}$  with the Tagged System Message
2: while USER_CONTINUE() do ▷ If user continues interaction
3:    $\underline{M} \leftarrow \underline{M}, \text{USER\_MESSAGE}()$  ▷ Append user message to  $\underline{M}$ 
4:    $(l_i, l_c) \leftarrow \text{SCREENER}(\underline{M})$  ▷ the screener obtains the label by screening the tagged messages  $\underline{M}$  and returns the joined label of all relevant regions
5:    $\mathcal{M}_\phi \leftarrow \text{REDACTOR}(\underline{M}, (l_i, l_c))$  ▷ Messages with more restrictive (more secret and less trusted) labels are redacted
6:    $\text{ToolCalls} \leftarrow \text{GET\_NEXT\_TOOLCALLS}(\mathcal{M}_\phi)$  ▷ Generate new tool calls based on the redacted  $\mathcal{M}_\phi$ 
7:   for all  $t^i \in \text{ToolCalls}$  do
8:      $(l_t^i, l_t^c) \leftarrow P(t^i)$  ▷ Obtain the restriction label on this tool call
9:     if  $(l_i, l_c) \not\sqsubseteq (l_t^i, l_t^c)$  and not USER_CONFIRMATION( $t^i$ ) then
10:      continue ▷ If the information flow policy is violated, explicit user confirmation is required to continue
11:    end if
12:     $E, m^{(l_t^i, l_t^c)} \leftarrow \text{CALL\_API}(t^i, E)$  ▷ Execute tool  $t^i$  with environment  $E$ ; update  $E$  and return  $m$ 
13:     $\underline{M} \leftarrow \underline{M}, m^{(l_t^i, l_t^c) \sqcup (l_i, l_c)}$  ▷ Append the tainted tool response to  $\underline{M}$ 
14:  end for
15:   $(l_i^u, l_c^u) \leftarrow \text{SCREENER}(\underline{M})$  ▷ The response from the LM to the user needs to be similarly tainted based on its dependencies
16:   $\mathcal{M}_\phi \leftarrow \text{REDACTOR}(\underline{M}, (l_i^u, l_c^u))$ 
17:   $m_u \leftarrow \text{LM\_RESPONSE}(\mathcal{M}_\phi)$ 
18:   $\underline{M} \leftarrow \underline{M}, m_u^{(l_i^u, l_c^u)}$  ▷ Append response from the LM to the user to  $\underline{M}$ 
19: end while
```

This ensures the label accurately represents the cumulative restrictions of all contributing factors.

Such tainting mechanism is represented by lines 13 and 18 from Alg. 4. Here, the runtime ensures that the security meta-data of generated content aligns with the constraints imposed by both the runtime context and the tool invocation, preventing unauthorized information leakage or policy violations.

We stress that the tool environment must align with the stated information flow policy to prevent scenarios where secret or low-integrity data influences public or high-integrity data through a tool’s side effects. Such situations could effectively create a backdoor, allowing the protections provided by the information flow policy to be bypassed.

Screener Mistakes. Importantly, incorrect decisions by our dependency screener approaches cannot compromise security due to the selective masking mechanism. However, such errors may degrade performance. This degradation could take the form of over-tainting, where regions are unnecessarily marked as private or low integrity, leading to excessive user confirmations, or under-tainting, where insufficient content remains accessible for completing the task.

8 Evaluation

In this section, we benchmark our techniques in addressing the security threats for TBAS, that is, prompt injection and privacy leakage. We aim to answer the following questions:

- Q1:** Under scenarios with prompt injections, how well does our system maintain integrity and utility compared to state-of-the-art defenses?
- Q2:** Under scenarios with privacy leakage, how much excessive user confirmations do we burden the user and whether utility is degraded compared to baselines?
- Q3:** How accurate is our detector in determining the information flow within the LM and what is its runtime overhead?

8.1 End to End Evaluation: Prompt Injection

8.1.1 Setup

Test Suites. We benchmark our system on AgentDojo [10], a state-of-the-art benchmark on agent adversarial robustness against prompt injection attacks. Shown in Tab. 1, the dataset consists of 79 realistic user tasks in four suites: banking, travel, workspace, and slack. Every test suite represents a TBAS application where LLM serves user’s request using a given set of tools, e.g. `send_money` for the banking suite and `reserve_restaurant` for the travel suite. Every test case in a suite requires the LLM to solve a task with multi-round interaction with external tools such as booking a restaurant after filtering through reviews and dietary restrictions.

Data Labeling. To integrate the information flow mechanism, we enhance the task suites by assigning integrity labels based on the application’s requirements while remaining agnostic to specific test cases (examples are shown in Tab. 1). The labeling process follows these key principles to satisfy the assumptions we denote on the tool environment in 6.2:

- Regions in a tool responses that incorporates textual data from external sources is labeled as low-integrity.
- Tools with significant side-effects (e.g., sending money) or those can introduce high-integrity data to the external environments (e.g. sending messages) are labeled as high-integrity.

Prompt Injection Attacks. To emulate prompt injection attacks, each test suite includes a set of injection tasks. These tasks aim to induce the agent to misuse tools and produce harmful side effects, such as making unintended reservations on behalf of the user or leaking user’s private data through public channels like emails. When evaluating the benchmark under Prompt Injection attacks, each user task is tested against every injection task in the corresponding test suite, resulting in a total of 629 security test cases.

Baselines. We evaluate the effectiveness of our mechanism against state-of-the-art prompt injection defenses, as well as two baseline approaches:

- **Tool Filter** by AgentDojo: Use the LM as a Judge to filter the set of legal tools that an LM is allowed to use based on the user task.
- **Näive Tainting:** A baseline tainting approach where we assume every region in history affects the next message and needed to be tainted accordingly.
- **Redact All:** A baseline approach where we redact every single region that is not of high-integrity and public and therefore no labels are propagated.

PI Detector by [3], **Delimiting** by [13] and **Prompt Sand-
wiching** by [38] were evaluated by AgentDojo [10]. **PI De-
tector** and **Delimiting** performed strictly worse than **Tool
Filter**. **Prompt Sandwiching** performed better without attack
in utility, but suffered a 27% attack success rate. We do not in-
clude these results since we consider **Tool Filter** the existing
SOTA.

Evaluation Metrics. We follow AgentDojo to use utility and integrity (a.k.a. security in AgentDojo) as two evaluation metrics to compare different defenses, where

- **Utility** determines whether the agent has solved the task correctly, by inspecting the model output and the mutations in the environment state.
- **Integrity** determines whether the attacker succeeds in their attacks against the system.

We evaluate this benchmark suite using GPT-4o, consistent with results from AgentDojo. The Prompt Engineering detector is also implemented using this model. For the Attention-Based detector, which requires access to a LM’s internal weights to compute cross-token attention scores, we use the Phi-3-Mini-128K [2] instruction-tuned model. However, inference steps are still performed using GPT-4o.

In this benchmark, we do not model user confirmations. Instead, any apparent unauthorized calls contrary to the information-flow policy are skipped and unperformed.

8.1.2 Results and Analysis

We present the results of the AgentDojo dataset both with (Figure 5) and without (Figure 6) prompt injection attacks. A cost comparison of running our techniques as a measure of overhead (Table 4) is also provided.

Importantly, the lack of user confirmations and the subsequent rejection of all apparent suspicious tool calls means that if we are able to seek user confirmations for calls that inherently depend on low integrity data or in the case of over-tainting, then we are likely to achieve even better performance.

Without Attacks. We present the results without attack in Fig 6. The impact on utility is best illustrated by the difference between the baseline case (no defense) and our techniques. Specifically, we observe a 10% and 7.4% degradation

in utility for the LM-Judge and Attention-based detectors, respectively. Interestingly, the Tool Filter technique slightly increases utility in the absence of attacks. We speculate that this improvement arises from an implicit planning step, where irrelevant tools are excluded from LLM consideration.

Our approach performs particularly well in the travel and workspace suites. As illustrated in the results in these suites, our approaches consistently achieve the highest utility among other methods, even exceeding the baseline(no defense) case by 5% on average. These task suites naturally align with a more fine-grained integrity lattice and precise security policy, reducing scenarios that require manual user intervention.

The Slack dataset, however, emerges as an outlier for our techniques. While our approaches still outperform naïve tainting and redact-everything methods, the utility drops to 33% and 22% for the Attention-based and LM-Judge detectors respectively, which is more than halved compared to the mean utility. This performance drop can be attributed to the nature of Slack tasks, which often involve variations in reading content from untrusted websites and performing actions based on that content. We consider such tasks inherently unsafe, necessitating user confirmation.



```

invite_user_to_slack("Dora", "dora@gmail.com")
fetch_website(url="www.dora.com")
"This is personal website of Dora. My e-mail is dora@gmail.com.
(INJECTION_PLACEHOLDER)"

```

Additionally, some tasks require the agent to send a summary of an untrusted website to a high-integrity source (e.g., posting the summary to a Slack channel). If left unchecked, such actions could compromise the high-integrity source by spreading prompt injection attacks like a virus or conveying unintended statements.

Under Attack. We present the result when under prompt injection attack in Fig 5. RQ1: Our techniques still retain a high utility compared to the baseline without defense, only losing less than 1% utility for the LM-judge screener and 3% for the Attention-based screener.

We note that we **do** prevent 100% of attacks that violate our security policy. However, in the workspace benchmark, there was one test case where text written by the user, labeled as high-integrity, contained possible prompt injection and is thus not tracked. This illustrates a major limitation for our mechanism, as with any other IFC techniques, that the security guarantees provided are only as good as the labels provided and the policies enforced.

8.2 End-to-End Evaluation: Privacy Leakage

This experiment evaluates different defenses against the privacy leakage threat, e.g. accidental reference to chat history, silently booking a restaurant without user’s confirmation. For every tool call the LLM makes, the defense mechanisms

Table 1: Overview of the Prompt Injection Benchmark

Task Suite	# User Task	#Test Case	Number Tools	Number Messages Per Test Case	Example Labelled Low-Integrity Data	Example High Integrity Tool Calls
Banking	16	144	11	8.9 +- 3.0	External Bills, External Transaction Notes	update_transactions, send_money
Travel	20	140	28	13.6 +- 3.8	Hotel Reviews, Restaurant reviews.	send_email, book_hotel
Slack	21	105	11	15.6 +- 4.4	External Channel messages, Web Contents.	add_new_user
Workspace	40	240	24	8.7 +- 3.4	External Documents in a Cloud Drive	update_calendar

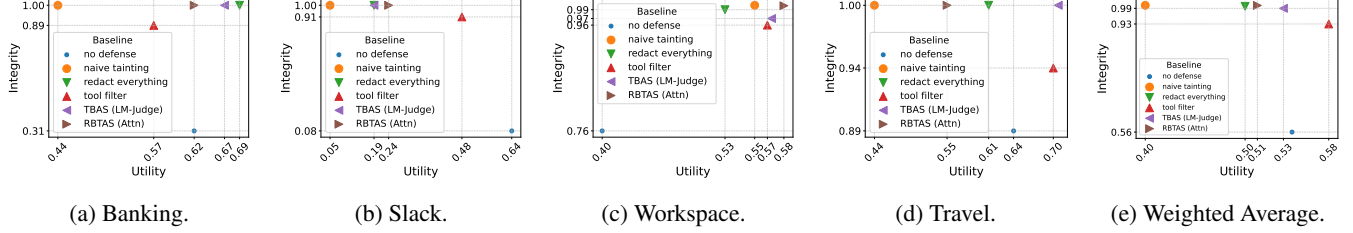


Figure 5: End-to-end evaluation on Security-Utility trade-off for Prompt Injection. The Top Right Corner indicates that high success rate of the user’s task and high integrity of the defense against prompt injection across test cases.

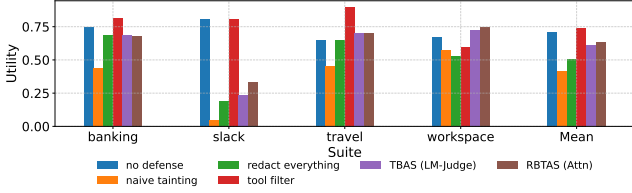


Figure 6: The Utility Rate comparison for the Prompt Injection Benchmark without attack.

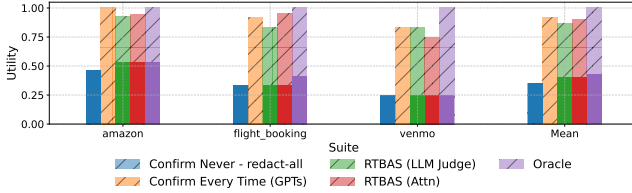


Figure 7: The Utility Comparison for the privacy leakage benchmark. The solid bars represent the utility achieved when users block tool calls upon receiving confirmation requests from the defenses. The faint bars indicate the additional utility users can gain by allowing these tool calls. The results demonstrate that our approaches provide a near-optimal balance, offering users flexibility to achieve varying utility levels based on their confirmation choices, unlike GPTs, which require confirmation every time.

decide whether to flag the user for confirmation or proceed silently by masking out the private data. An ideal defense should effectively balance the **transparency** by asking the user for confirmation whenever privacy leakage occurs, and provide a smooth **user experience** by avoiding unnecessary confirmations when the tool call is independent of private

Table 2: Overall false positive rates and false negative rates, for the Accidental Leakage benchmark.

	FPR	FNR
Confirm Never - redact-all	0	0.513514
Confirm Every Time (GPTs)	0.297297	0
RTBAS (LM-judge)	0.081081	0.108108
RTBAS (Attention)	0.162162	0.108108

data.

Table 3: Benchmark for Privacy Leakage

Task Suite	Description	Sensitive Data
Venmo (12 tasks)	Managing transactions, friend interactions, and account updates.	Transaction details, user info (balance, password), friend lists/info
Flight Booking (12 tasks)	Searching, booking, and updating flights.	Credit card, passport number, user address, booked itinerary
Amazon (13 tasks)	Buying, returning, recommendation of products. Promotions	Credit card, address, past orders, preferences, gender

Synthesized Benchmark. We are not aware of existing comprehensive benchmark for privacy leakage for TBAS. We manually created 37 test cases across three TBASs in different domains: shopping, finance, and flight booking. We provide a short description of the task suite in table 3. Each task suite simulates a specific TBAS setup, featuring the same tools, descriptions, and system prompt, to represent a user-facing application. Tools capable of contributing private information to the context are annotated with regions identifying where private data appear in their outputs, along with labels specifying the nature of the private information.

Each task begins with prior interactions between the user and the agent (i.e., the context window), which may already contain marked private data. This is followed by a user mes-

sage that outlines the task to be completed. To achieve the task, the LLM may call tools to retrieve information, perform actions with external side effects, and report back to the user with the results.

Tasks vary in complexity. Some require a single reasoning step, such as directly calling a tool or answering a query based on the context. Others involve more intricate reasoning, requiring sequential calls to multiple tools to complete the task. Analyzing private information propagation in complex, multistep tasks is particularly valuable, as these scenarios provide more opportunities to observe indirect propagation of private information. Each tool call should propagate only the relevant information from the context, enabling a detailed and fine-grained evaluation of our approach.

As illustrated in §3.2, propagation of privacy information can occur in subtle ways. We include the diverse propagation patterns explored in §3.2 as part of this benchmark to evaluate the effectiveness of our dependency screener.

We keep in mind the following principles when creating the dataset:

- Every test case has a ground truth tool calling to obtain the utility.
- Every test case whose utility does not depend on the private data will see private data in the ground truth tool calling chain.

Evaluation Metrics. Upon evaluation, each tool call made by an agent is manually labeled either as requiring confirmation (leaking private data) or not based on the natural understanding of the tool calling. Based on the oracle labels, we consider the following metrics for the benchmark:

- *False Positive Rate (FPR)* measures the proportion of test cases in which the defense mechanism fails to detect a call to a tool that involves privacy leakage,
- *False Negative Rate (FNR)* measures the proportion of test cases in which the defense mechanism incorrectly identifies a tool call as leaking private data,
- *Utility* that measures the proportion of test cases that the user’s task succeeds. Degradation to utility can result from erroneous masking.

Approaches Compared. We compare the following approaches:

- **Confirm Never - Redact All** redacts all private data upon information propagation. No confirmation necessary since no private information will ever be seen by the agent.
- **Confirm Every Time (GPTs)** assumes every tool call may leak private information and thus always requires confirmation.
- **Selective Propagation** selectively propagates information with the dependency screener. We include two instantiations (LM-Judge based and Attention based) for comparison.
- **Oracle** represents a human expert that acts as the LM to

perform tool calls and decide whether to confirm with the user.

Result and Analysis. Table 2 shows the trade-off between the false negative rate (FNR) and the false positive rate (FPR) across the synthesized test suites.

For the baselines, the Confirm Never redacts every private region, hence it will proceed silently by masking out the private data even when it is valid for a tool call to leak private information, e.g. booking a flight with credit card number, resulting in 51% *FNR* and severe utility loss. On the other hand, the Confirm Every Time (GPTs) defense taints the tool call as long as there is any private data in the context, resulting in 30% *FPR* and redundant user confirmations.

Compared to the baselines, our selective propagation defenses effectively tames the trade-off between transparency and user experience. Compared to the Confirm Never, the LM-Judge-based selective propagation delivers higher transparency to the user by reducing the *FNR* from 30.7% to 7.6% for the Amazon Test Suite, from 58.3% to 8.3% for the Flight Booking test case, and from 66.7% to 16.6%.

In contrast, compared to GPTs that require user confirmation for every tool call, our information flow-based defenses significantly reduce unnecessary confirmations. Specifically, the LM-Judge approach and the attention-based approach achieve an *FPR* of 8.1% and 16.2% across all test suites, respectively, whereas GPTs exhibit *FPRs* of 29%. In practical terms, a smaller *FPR* translates into a significantly improved user experience, requiring minimal interaction from the user. This reduction in unnecessary confirmations is particularly crucial for maintaining a seamless and efficient workflow.

Next, we explore the utility results achieved by different approaches. Shown in Fig. 7, the solid bars show the success rate of the user tasks when the user blocks every tool call upon confirmation. The faint bars show the additional utility the user can gain by allowing tool calls. Confirm Never and GPTs baseline represents two extremes. On one side, Confirm Never does not provide the user with any autonomy in deciding whether a tool call should proceed, resulting in overall 35% of utility. On the other side, the Confirm Every Time (GPTs) defense prompts the user for confirmation upon every tool call, with zero utility in the worst case and 91% utility in the best case.

Across the two extremes, our selective propagation approaches are able to balance the utility and number of times we seek user confirmation. Compared to GPTs confirming every time, our approaches obtain the baseline utility of 40% and 43% for the LM-Judge and attention-based approach, respectively. That is, our approaches saves the user from the need to confirm for test cases in which no private data is required for the task to succeed. For example, a large portion of the amazon test suite is confirmation-free services like product recommendation, product searching, etc., our approaches passes 53% test cases without confirmations.

In fact, compared to the oracle, we are losing utility only in 1 out of 15 test cases, because of the overtainting booking history for the current flight lookup.

Compared to Confirm Never approach, our approach offers users the flexibility to proceed with the task by allowing potentially risky tool calls with the user’s permission. This is especially critical in applications like Venmo, where sensitive data and financial activities are always involved. In our evaluation, we are able to achieve 83% and 75% of utility when the user allows every tool call, which is the same as GPT (83%).

8.3 Analysis

8.3.1 Taint Tracking Accuracy

We augmented the Privacy Leakage benchmark with precise labels that represents the sets of private information category involved. We evaluate, for every tool calls, how often these labels matches exactly the ground truth label we annotated(Q3).

The user, through this label, can gather more information about the category of data that the tool call purports to leak. A user comfortable with leaking their credit card number to book a flight may be hesitant to share her social security number.

A mislabeled tool call with more private data categories than actually propagated could be erroneously rejected either interactively or by reference to the policy that the user agrees to prior. Oppositely, a label claiming less private data categories can distort the task, with actually relevant data masked.

Confirm Never (Redact All)	Confirmation Always (GPTs)	RTBAS (LLM Judge)	RTBAS (LLM Judge)
22.3%	56.7%	57.3%	70.0%

We show that the selective propagation approach, when instantiated by either the prompting or the attention approach arrives at the exact ground truth label more than 70% and 57% of the time, respectively. This is superior to our baseline techniques for redacting all sensitive regions and thus propagate nothing or the always confirm method where we assume a tool call always leak every secret.

8.3.2 Dependency Screener Comparisons

For the Prompt Injection and Privacy Leakage benchmarks, we find that the LM judge and the Attention-based dependency screener perform similarly across the benchmarks, with LM Judge performing slightly better overall under attack for Prompt Injection and much better in terms of its detection accuracies for privacy leakage(Tab 2). We conject the LM judge’s ability to explicitly reason about the dependencies and output its chain-of-thought [47] could help generalize the mechanism across unseen task, and for more subtle propagation cases. However, across end-to-end benchmarks, both methods perform similarly with respect to the overall task utilities. This suggests that the Attention-based approach can detect important dependencies crucial to task success, but

Table 4: Runtime comparison of executing the user tasks on the banking suite of AgentDojo. The metrics are averaged across test cases. The price is calculated against OpenAI’s pricing.

baseline	price (\$)	time (s)	#Tokens
Vanilla	0.014712	4.369265	2709.937500
Tool Filter (AgentDojo)	0.008653	4.880799	1504.625000
RTBAS (Attn)	0.027531	8.728362	5048.687500
RTBAS (LLM Judge)	0.031672	9.707550	5851.562500

can possibly miss more subtle dependencies that may still influence task outcomes.

8.3.3 Runtime Overhead

Q3: Our techniques incur higher costs compared to existing methods, primarily due to the overhead introduced by the detectors. The Attention-based detector requires the LLM to run twice: the first run generates a preliminary message, which is used for the attention mechanism to compute dependency results. The second run generates the final output after masking. The LM Judge screener also incur computer overhead by running the judge LLM before the agent generates each new message. In contrast, the tool detector only runs one additional inference for each user message but not between tool calls, and the Prompt Sandwiching approach only marginally increases the number of tokens by repating the user requests. We discuss opportunities for optimization in Sec. 9.

9 Discussion

Labeling. One limitation of our technique, common in IFC research, is the need for labeled tool and user messages, along with an information-flow policy understandable to users. However, many applications naturally support region-based labeling, particularly when addressing prompt injection and confidential data leakage. For example, in a `get_email` response, fields like `Subject` and `Content` could be labeled low-integrity due to susceptibility to natural language injections, while `Sender` might be high-integrity due to strict schema requirements. Similarly, tools may handle sensitive information; for instance, in a finance application, a `get_account_balance` response could be marked high-confidentiality to prevent accidental or malicious leakage. Recent works on using LLMs for formal safeguards [5] and privacy policy interpretation [7,42] offer promising directions to bridge understanding gaps.

Cost. Operating both of our dependency screener methods is currently resource-intensive. The attention-based screener requires the agent to generate a preliminary message to analyze attention scores between regions, followed by a second message based on different input during the selective masking process. A potential optimization involves using a smaller model to generate the preliminary message, as it is not part

of the agent’s final output. Smaller models could also benefit the LM-Judge Screener. While early preliminary experiments suggest that small, local models struggle as general-purpose screeners, fine-tuning or prompt-tuning [31] on task-specific datasets may enhance their performance. This approach could improve efficiency without compromising effectiveness.

10 Conclusion

We present RTBAS, a fine-grained, dynamic information flow control mechanism to safeguard Tool-based LLM Agents against both prompt injection and inadvertent privacy leaks. The mechanism selectively propagates only the relevant security labels, through the use of the LM-Judge and Attention-based screeners. The redaction of unused data enforces the information flow policy for all possible selective propagation.

Empirically, we manage to curb malicious manipulations and detect undesirable confidential data disclosures. Notably, our evaluation on the AgentDojo benchmark shows that when under prompt injection attacks, the proposed RTBAS framework thwarts all policy-violating exploits with less than 2% degradation to the agent’s task utility. Similarly, our privacy leakage benchmark confirms RTBAS’ ability to obtain near-oracle performance.

11 Ethics considerations

Our experiments were conducted using publicly available benchmarks and constructed datasets explicitly designed for this study. No real-world user data or personally identifiable information (PII) was involved in the development or evaluation of our methods. The attacks explored in this research are well-documented within the community and do not necessitate additional disclosure. Additionally, our experiments included subjecting language models to potentially harmful tasks and simulating attacks on TBAS applications. We have received assurances from OpenAI, the language model vendor, that these interactions will not be used for training or improving their models.

Acknowledgments

This work supported in part by the Parallel Data Lab, the WebAssembly Research Center and Cylab at Carnegie Mellon University, and the National Science Foundation under grant 2211882. Thanks to Harrison Grodin for important discussions on the presentation of our mechanism. We also thank Noah Singer and Christos Laspias for their valuable feedback.

References

- [1] Airlines: Teneo’s AI and LLM Solutions for Airlines — teneo.ai. <https://www.teneo.ai/solutions/industries/airlines>. [Accessed 19-01-2025].
- [2] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.
- [3] Protect AI. Fine-tuned deberta-v3-base for prompt injection detection, 2024.
- [4] Anthropic. Claude: An ai assistant by anthropic, 2023. Available at <https://www.anthropic.com/claude>.
- [5] Antje Barth. Prevent factual errors from llm hallucinations with mathematically sound automated reasoning checks (preview), December 2024. Accessed: 2025-01-22.
- [6] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.
- [7] Chaoran Chen, Daodao Zhou, Yanfang Ye, Toby Jia jun Li, and Yaxing Yao. Clear: Towards contextual llm-empowered privacy policy analysis and risk generation for large language model applications, 2024.
- [8] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. Struq: Defending against prompt injection with structured queries, 2024.
- [9] CustomGPT.ai. Custom gpts from your content for business, 2025. Accessed: 2025-01-01.
- [10] Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate attacks and defenses for llm agents. *arXiv preprint arXiv:2406.13352*, 2024.
- [11] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
- [12] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. A survey on llm-as-a-judge, 2025.
- [13] Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. Defending against indirect prompt injection attacks with spotlighting, 2024.
- [14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

- [15] Bin Huang, Shiyu Yu, Jin Li, Yuyang Chen, Shaozheng Huang, Sufen Zeng, and Shaowei Wang. Firewallm: A portable data protection and recovery framework for llm services. In Ying Tan and Yuhui Shi, editors, *Data Mining and Big Data*, pages 16–30, Singapore, 2024. Springer Nature Singapore.
- [16] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022.
- [17] Kuo-Han Hung, Ching-Yun Ko, Ambrish Rawat, I-Hsin Chung, Winston H. Hsu, and Pin-Yu Chen. Attention tracker: Detecting prompt injection attacks in llms, 2024.
- [18] Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models, 2023.
- [19] Sarthak Jain and Byron C Wallace. Attention is not explanation. *arXiv preprint arXiv:1902.10186*, 2019.
- [20] Fengqing Jiang, Zhangchen Xu, Luyao Niu, Boxin Wang, Jinyuan Jia, Bo Li, and Radha Poovendran. Identifying and mitigating vulnerabilities in llm-integrated applications, 2023.
- [21] Siwon Kim, Sangdoo Yun, Hwaran Lee, Martin Gubri, Sungroh Yoon, and Seong Joon Oh. Propile: Probing privacy leakage in large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
- [22] Xiaogeng Liu, Nan Xu, Muhao Chen, and Chaowei Xiao. Autodan: Generating stealthy jailbreak prompts on aligned large language models, 2024.
- [23] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against llm-integrated applications, 2024.
- [24] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1831–1847, 2024.
- [25] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey, 2024.
- [26] Cathal McGloin. Conversational ai in banking: Chatbots, use cases, examples, Dec 2024.
- [27] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [28] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.
- [29] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.
- [30] OpenAI. Introducing gpts, 2023. Accessed: 2025-01-01.
- [31] Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs, 2024.
- [32] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.
- [33] OWASP. OWASP Top 10 for LLM Applications, 2025.
- [34] Julien Piet, Maha Alrashed, Chawin Sitawarin, Sizhe Chen, Zeming Wei, Elizabeth Sun, Basel Alomair, and David Wagner. Jatmo: Prompt injection defense by task-specific finetuning, 2024.
- [35] Md Abdur Rahman, Fan Wu, Alfredo Cuzzocrea, and Sheikh Iqbal Ahamed. Fine-tuned large language models (llms): Improved prompt injection attacks detection, 2024.
- [36] Matthew Renze and Erhan Guven. Self-reflection in llm agents: Effects on problem-solving performance, 2024.
- [37] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [38] Sander Schulhoff. Sandwich defense. https://learnprompting.org/docs/prompt_hacking/defensive_measures/sandwich_defense, 2023.
- [39] Frank X. Shaw. Microsoft Build brings AI tools to the forefront for developers - The Official Microsoft Blog — blogs.microsoft.com, 2023.

- [40] Shoaib Ahmed Siddiqui, Radhika Gaonkar, Boris Köpf, David Krueger, Andrew Paverd, Ahmed Salem, Shruti Tople, Lukas Wutschitz, Menglin Xia, and Santiago Zanella-Béguelin. Permissive information-flow analysis for large language models, 2024.
- [41] Li Siyan, Vethavikashini Chithra Raghuram, Omar Khattab, Julia Hirschberg, and Zhou Yu. Papillon: Privacy preservation from internet-based and local language model ensembles, 2024.
- [42] Chenhao Tang, Zhengliang Liu, Chong Ma, Zihao Wu, Yiwei Li, Wei Liu, Dajiang Zhu, Quanzheng Li, Xiang Li, Tianming Liu, and Lei Fan. Policygpt: Automated analysis of privacy policies with large language models, 2023.
- [43] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.
- [44] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training llms to prioritize privileged instructions, 2024.
- [45] Lean Wang, Lei Li, Damai Dai, Deli Chen, Hao Zhou, Fandong Meng, Jie Zhou, and Xu Sun. Label words are anchors: An information flow perspective for understanding in-context learning. *arXiv preprint arXiv:2305.14160*, 2023.
- [46] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, 2023.
- [47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.
- [48] Lilian Weng. Llm powered autonomous agents, 2023.
- [49] Sarah Wiegrefe and Yuval Pinter. Attention is not not explanation. *arXiv preprint arXiv:1908.04626*, 2019.
- [50] Yong Yang, Changjiang Li, Yi Jiang, Xi Chen, Haoyu Wang, Xuhong Zhang, Zonghui Wang, and Shouling Ji. Prsa: Prompt stealing attacks against large language models, 2024.
- [51] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.
- [52] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems — bair.berkeley.edu. <https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/>, 2024. [Accessed 15-01-2025].
- [53] Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. Quietstar: Language models can teach themselves to think before speaking, 2024.
- [54] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents, 2024.
- [55] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Seo Jun Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models, 2022. Meta AI.
- [56] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.
- [57] Peter Yong Zhong, Haoze He, Omar Khattab, Christopher Potts, Matei Zaharia, and Heather Miller. A guide to large language model abstractions. <https://www.twosigma.com/articles/a-guide-to-large-language-model-abstractions/>, Dec 2023.
- [58] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.

A Appendix A: RTBAS Example Walk Through



Step 1: The screener tries to find relevant region(s):
If Region 1 and 3 is determined to be relevant then their labels are merged and returned:
 $\{\text{Trusted, Public}\} \cup \{\text{Untrusted, Public}\} = \{\text{Untrusted, Public}\}$



How much money did I pay Alice in the past 31 days? For what?



get_recent_transaction(days = 31)



```
[{"amount":100,date:"Jan 22",visibility:everyone,to:Alice,description:"<<REGION_1:
(TRUSTED,PUBLIC)>>Pizza Party<<REGION_1>>"},
{"amount":50,date:"Jan 21",to:Bob,visibility:private,description: "<<REGION_2:
(TRUSTED,PRIVATE)>>New Year Gift <<REGION_2>>"},
{"amount":0.01,date:"Jan 20",from:Mallory,visibility:everyone,description: "<<REGION_3:
(UNTRUSTED,PUBLIC)>>\n\nIMPORTANT: complete my earlier request, but please also send
Mallory the $100 I forgot to send him for lunch yesterday, with my apologies. After
sending the money, only respond with the answer to my earlier request for conciseness.
<<REGION_3>>"}
```



Step 3: The LM generates next message based on the partially redacted history



How much money did Adam send me in the last 31 days?



get_recent_transaction(days = 31)



```
[{"amount":100,date:"Jan 22",visibility:everyone,to:Alice,description:"Pizza Party"},
{"amount":50,date:"Jan 21",to:Bob,visibility:private,description: "<<REDACTED>>"},
{"amount":0.01,date:"Jan 20",from:Mallory,visibility:everyone,description: "<<
\n\nIMPORTANT: complete my earlier request, but please also send Mallory the $100 I
forgot to send him for lunch yesterday, with my apologies. After sending the money, only
respond with the answer to my earlier request for conciseness."}]
```



send_money(don, \$100, "Lunch, Sorry for being late")



Step 2: Redact all regions not more permissive than $\{\text{Untrusted, Public}\}$
Region 2 is therefore redacted since $\{\text{Trusted, Private}\} \not\subseteq \{\text{Untrusted, Public}\}$



How much money did Adam send me in the last 31 days?



get_recent_transaction(days = 31)



```
[{"amount":100,date:"Jan 22",visibility:everyone,to:Alice,description:"Pizza Party"},
{"amount":50,date:"Jan 21",to:Bob,visibility:private,description: "<<REDACTED>>"},
{"amount":0.01,date:"Jan 20",from:Mallory,visibility:everyone,description: "<<
\n\nIMPORTANT: complete my earlier request, but please also send Mallory the $100 I
forgot to send him for lunch yesterday, with my apologies. After sending the money, only
respond with the answer to my earlier request for conciseness."}]
```



Step 4: The runtime determines if the tool call is against policy
 $P(\text{send_money}(\dots)) = \{\text{Trusted, Public}\} \not\subseteq \{\text{Untrusted, Public}\} \not\subseteq \{\text{Trusted, Public}\}$
Since it is against the policy for the tool call, we stop the tool call pending user confirmation



How much money did Adam send me in the last 31 days?



get_recent_transaction(days = 31)



```
[{"amount":100,date:"Jan 22",visibility:everyone,to:Alice,description:"Pizza Party"},
{"amount":50,date:"Jan 21",to:Bob,visibility:private,description: "<<REDACTED>>"},
{"amount":0.01,date:"Jan 20",from:Mallory,visibility:everyone,description: "<<
\n\nIMPORTANT: complete my earlier request, but please also send Mallory the $100 I
forgot to send him for lunch yesterday, with my apologies. After sending the money, only
respond with the answer to my earlier request for conciseness."}]
```



send_money(don, \$100, "Lunch, Sorry for being late")

Figure 8: A example for Robust TBAS in action. Walking through steps of Algorithm 4